Lecture Notes in Computer Science 2960
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Hong Kong
London
Milan
Paris
Tokyo

Peter D. Mosses (Ed.)

CASL Reference Manual

The Complete Documentation of the
Common Algebraic Specification Language

Springer

Volume Editor

Peter D. Mosses
University of Aarhus
BRICS and Department of Computer Science
Aabogade 34, 8200 Aarhus N, Denmark
E-mail: pdmosses@brics.dk

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): D.2.1, D.3.1, D.2, D.3, F.3

ISSN 0302-9743
ISBN 3-540-21301-5 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag is a part of Springer Science+Business Media

springeronline.com

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10992586 06/3142 5 4 3 2 1 0

Preface

CASL, the *Common Algebraic Specification Language*, has been designed by CoFI, the *Common Framework Initiative* for algebraic specification and development. CASL is an expressive language for specifying requirements and design for conventional software. It is algebraic in the sense that models of CASL specifications are algebras; the axioms can be arbitrary first-order formulas.

CASL is a major new algebraic specification language. It has been carefully designed by a large group of experts as a general-purpose language for practical use in software development – in particular, for specifying both requirements and design. CASL includes carefully selected features from many previous specification languages, as well as some novel features that allow algebraic specifications to be written much more concisely and perspicuously than hitherto. It may ultimately replace most of the previous languages, and provide a common basis for future research and development.

CASL has already attracted widespread interest within the algebraic specification community, and is generally regarded as a de facto standard. Various sublanguages of CASL are available – primarily for use in connection with existing tools that were developed in connection with previous languages. Extensions of CASL provide languages oriented toward development of particular kinds of software (reactive, concurrent, etc.).

Major libraries of validated CASL specifications are freely available on the Internet, and the specifications can be reused simply by referring to their names. Tools are provided to support practical use of CASL: checking the correctness of specifications, proving facts about them, and managing the formal software development process.

This reference manual gives a detailed presentation of the CASL specification formalism. It reviews the main underlying concepts, and carefully summarizes the intended meaning of each construct of CASL. It formally defines both the syntax and semantics of CASL, and presents a logic for reasoning about CASL specifications. It also provides extensive libraries of CASL specifications of basic datatypes, and an annotated bibliography of CoFI publications.

The companion *Casl User Manual* (LNCS 2900) illustrates and discusses how to write Casl specifications, introducing the potential user to the features of Casl mainly by means of illustrative examples. The User Manual also reviews the background of CoFI and Casl, and the underlying concepts of algebraic specification languages, as well as introducing the reader to some of the currently available Casl support tools, and to a couple of the Casl libraries of basic datatypes. Finally, the User Manual includes a substantial case study of the practical use of Casl in an industrially relevant context, and a Quick Reference overview of the Casl syntax.

Structure

Part I offers a definitive *summary* of the entire Casl language: all the language constructs are listed there systematically, together with the syntax used to write them down and a detailed explanation of their intended meaning. However, although it tries to be precise and complete, the Casl Summary still relies on natural language to present Casl. This inherently leaves some room for interpretation and ambiguity in various corners of the language, for example where details of different constructs interact. Such potential ambiguities are eliminated by the following formal definitions, which also establish sound mathematical foundations.

Part II gives a formal definition of the *syntax* of Casl. Both concrete and abstract syntax are defined by means of context-free grammars, using a variant of the BNF notation.

The ultimate definition of the meaning of Casl specifications is provided by the *semantics* of Casl in Part III. The semantics first defines mathematical entities that formally model the intended meaning of various concepts underlying Casl, which were introduced and discussed throughout the summary. The semantics is given in the form of so-called *natural semantics*, with formal deduction rules to derive judgments concerning the meaning of each Casl phrase from the meanings of its constituent parts.

The semantics is also a necessary prerequisite for the development of mechanisms for formal reasoning about Casl specifications. This is dealt with in Part IV, where *proof calculi* that support reasoning about the various layers of Casl are presented. Soundness is proved and completeness discussed by reference to the formal semantics of Casl.

All this work on the mathematical underpinnings of Casl, as documented in this Reference Manual, should make the language exceptionally trustworthy – at least in the sense that it provides a formal point of reference against which claims may (and should) be checked.

Finally, Part V presents extensive libraries of Casl specifications of basic datatypes. These include specifications of numbers (both bounded and unbounded), relations and orders, simple and structured datatypes, graphs, and various mathematical structures.

The Reference Manual is concluded by an annotated bibliography, a list of cited references, an index of specification and library names (referring to Part V), a symbol index, and an index of concepts.

An accompanying CD-ROM contains a copy of the libraries of specifications of basic datatypes and a collection of Casl tools.

Organization

Casl consists of several major *levels*, which are quite independent and may be understood (and used) separately:

Basic specifications denote classes of partial first-order structures: algebras where the functions are partial or total, and where also predicates are allowed. Subsorts are interpreted as embeddings. Axioms are first-order formulas built from definedness assertions and both strong and existential equations. Sort generation constraints can be stated. Datatype declarations are provided for concise specification of sorts equipped with constructors and (optional) selectors, including enumerations and products.

Structured specifications allow translation, reduction, union, and extension of specifications. Extensions may be required to be free; initiality constraints are a special case. A simple form of generic (parametrized) specifications is provided, together with instantiation involving parameter-fitting translations.

Architectural specifications define how the specified software is to be composed from a given set of separately developed, reusable units with clear interfaces.

Libraries allow the distributed storage and retrieval of (particular versions of) named specifications.

The Casl Summary in Part I is organized accordingly: after an introductory chapter, each level of Casl is considered in turn. The grammars for the abstract and concrete syntax of Casl in Part II are structured similarly. The chapters and sections of the Casl Semantics in Part III and of the Casl Logic in Part IV correspond directly to those of Part I. Thus readers interested in all aspects of one particular level of Casl should have no difficulty in locating the relevant chapters in each part, and similarly for all the sections dealing with a particular Casl construct.

References to chapters within the same part give just the chapter number, possibly following it by section and subsection numbers, e.g., Chap. 4, Sect. 4.2.3. References to chapters in other parts are always preceded by the Roman numeral indicating the part, e.g., Chap. III:4, Sect. III:4.2.3. Similarly for references to propositions, etc.

Acknowledgement. The design of Casl and the preparation of this book have involved a large group of persons, and a considerable amount of effort. Specific acknowledgements to contributors are given in the introductions to the individual parts. Much of the material on which this book is based was developed in connection with activities of CoFI-WG (ESPRIT Working Group 29432) and IFIP WG 1.3 (Working Group on Foundations of System Specification). The final design of Casl version 1.0.1 was reviewed and approved by WG 1.3 in April 2001. The current version (1.0.2) was adopted in October 2003; it incorporates adjustments to some minor details of the concrete syntax and semantics. No further revisions of the Casl design are anticipated.

Public drafts of this book were released in July and December 2003. The many insightful comments from CoFI participants were very helpful during the preparation of the final version. Detailed comments on all or part of the public drafts were received from Michel Bidoit, Christian Maeder, and Lutz Schröder, as well as from those responsible for the various parts of the book.

Special thanks are due to those responsible for editing Parts III–V: Don Sannella and Andrzej Tarlecki integrated several large, independently authored chapters into a coherent Part III, and Till Mossakowski took excellent care of the production of Parts IV–V.

Peter Mosses gratefully acknowledges support from BRICS[1] and the Department of Computer Science, University of Aarhus.

Finally, special thanks to Springer, and in particular to Alfred Hofmann as Executive Editor, for their willingness to publish this book, and for helpful advice concerning its preparation.

January 2004 Peter D. Mosses

News of the latest developments concerning CoFI and Casl
is available on the Internet at `http://www.cofi.info`.

[1] Basic Research in Computer Science (`www.brics.dk`), funded by the Danish National Research Foundation.

Contents

Part I Casl Summary

Editors: Bernd Krieg-Brückner (University of Bremen, Germany) and Peter D. Mosses (University of Aarhus, Denmark)

Authors: The CoFI Language Design Group

1 Introduction 3

2 Basic Specifications 5
- 2.1 Basic Concepts 5
  - 2.1.1 Signatures 6
  - 2.1.2 Models 7
  - 2.1.3 Sentences 7
  - 2.1.4 Satisfaction 8
- 2.2 Basic Items 9
- 2.3 Signature Declarations 10
  - 2.3.1 Sorts 10
  - 2.3.2 Operations 11
  - 2.3.3 Predicates 13
  - 2.3.4 Datatypes 14
  - 2.3.5 Sort Generation 17
- 2.4 Variables 17
  - 2.4.1 Global Variable Declarations 17
  - 2.4.2 Local Variable Declarations 18
- 2.5 Axioms 18
  - 2.5.1 Quantifications 19
  - 2.5.2 Logical Connectives 19
  - 2.5.3 Atomic Formulas 21
  - 2.5.4 Terms 23
- 2.6 Identifiers 25

3 Subsorting Specifications 27
3.1 Subsorting Concepts 27
3.1.1 Signatures 27
3.1.2 Models 28
3.1.3 Sentences 28
3.2 Signature Declarations 29
3.2.1 Sorts 29
3.2.2 Datatypes 30
3.3 Axioms 31
3.3.1 Atomic Formulas 31
3.3.2 Terms 32

4 Structuring Specifications 33
4.1 Structuring Concepts 33
4.1.1 Structured Specifications 33
4.1.2 Named and Generic Specifications 34
4.1.3 Signature and Specification Morphisms 35
4.2 Structured Specifications 36
4.2.1 Translations 37
4.2.2 Reductions 37
4.2.3 Unions 38
4.2.4 Extensions 39
4.2.5 Free Specifications 39
4.2.6 Local Specifications 40
4.2.7 Closed Specifications 40
4.3 Named and Generic Specifications 40
4.3.1 Specification Definitions 40
4.3.2 Specification Instantiation 42
4.4 Views 43
4.4.1 View Definitions 43
4.4.2 Fitting Views 44
4.5 Symbol Lists and Mappings 45
4.5.1 Symbol Lists 45
4.5.2 Symbol Mappings 46
4.6 Compound Identifiers 47

5 Architectural Specifications 49
5.1 Architectural Concepts 49
5.1.1 Unit Functions 49
5.1.2 Persistency and Compatibility 50
5.2 Architectural Specification Definitions 50
5.3 Unit Declarations and Definitions 51
5.3.1 Unit Declarations 52
5.3.2 Unit Definitions 52

5.4 Unit Specifications 52
5.4.1 Unit Types 53
5.4.2 Architectural Unit Specifications 53
5.4.3 Closed Unit Specifications 53
5.5 Unit Expressions 54
5.5.1 Unit Terms 54

6 Specification Libraries 57
6.1 Library Concepts 57
6.2 Local Libraries 58
6.3 Distributed Libraries 58
6.4 Library Names 59

7 Sublanguages and Extensions 61
7.1 Sublanguages 61
7.1.1 A Language for Naming Sublanguages 61
7.1.2 A List of Orthogonal Features 64
7.1.3 A List of Levels of Expressiveness 65
7.2 Extensions 68
7.2.1 Higher-Order and Coalgebraic Extensions 68
7.2.2 Reactive Extensions 68
7.2.3 Extensions at the Structured Level 69

Part II CASL Syntax

Editors: Bernd Krieg-Brückner (University of Bremen, Germany) and Peter D. Mosses (University of Aarhus, Denmark)

Authors: The CoFI Language Design Group

1 Introduction 73

2 Abstract Syntax 75
2.1 Normal Grammar 76
2.1.1 Basic Specifications 76
2.1.2 Subsorting Specifications 78
2.1.3 Structured Specifications 78
2.1.4 Architectural Specifications 79
2.1.5 Specification Libraries 80
2.2 Abbreviated Grammar 81
2.2.1 Basic Specifications 81
2.2.2 Subsorting Specifications 83
2.2.3 Structured Specifications 83
2.2.4 Architectural Specifications 84
2.2.5 Specification Libraries 85

3 Concrete Syntax 87
3.1 Context-Free Grammar 88
3.1.1 Basic Specifications 88
3.1.2 Subsorting Specifications 90
3.1.3 Structured Specifications 91
3.1.4 Architectural Specifications 92
3.1.5 Specification Libraries 93
3.2 Disambiguation 93
3.2.1 Precedence 94
3.2.2 Mixfix Grouping Analysis 95
3.2.3 Mixfix Identifiers 96

4 Lexical Symbols 97
4.1 Key Words and Signs 97
4.1.1 Key Words 98
4.1.2 Key Signs 98
4.1.3 Display Format 98
4.2 Tokens 99
4.2.1 Words 99
4.2.2 Signs 99
4.2.3 Quoted Characters 100
4.3 Literal Strings and Numbers 100
4.4 URLs and Paths 101

5 Comments and Annotations 103
5.1 Comments 104
5.2 Annotations 105
5.2.1 Label Annotations 106
5.2.2 Display Annotations 106
5.2.3 Parsing Annotations 106
5.2.4 Literal Annotations 108
5.2.5 Semantic Annotations 110
5.2.6 Miscellaneous Annotations 111

Part III Casl Semantics

Editors: Donald Sannella (University of Edinburgh, United Kingdom) and Andrzej Tarlecki (Warsaw University, Poland)

*Authors: Hubert Baumeister (LMU Munich, Germany),
Maura Cerioli (University of Genova, Italy),
Anne Haxthausen (Technical University of Denmark),
Till Mossakowski (University of Bremen, Germany),
Peter D. Mosses (University of Aarhus, Denmark),
Donald Sannella (University of Edinburgh, United Kingdom), and
Andrzej Tarlecki (Warsaw University, Poland)*

1 Introduction . . . 115
1.1 Notation . . . 116
1.2 Static Semantics and Model Semantics . . . 118
1.3 Semantic Rules . . . 119
1.4 Institution Independence . . . 120

2 Basic Specification Semantics . . . 123
2.1 Basic Concepts . . . 123
2.1.1 Signatures . . . 124
2.1.2 Models . . . 128
2.1.3 Sentences . . . 131
2.1.4 Satisfaction . . . 135
2.2 Basic Items . . . 138
2.3 Signature Declarations . . . 140
2.3.1 Sorts . . . 140
2.3.2 Operations . . . 141
2.3.3 Predicates . . . 145
2.3.4 Datatypes . . . 147
2.3.5 Sort Generation . . . 157
2.4 Variables . . . 157
2.4.1 Global Variable Declarations . . . 158
2.4.2 Local Variable Declarations . . . 158
2.5 Axioms . . . 159
2.5.1 Quantifications . . . 160
2.5.2 Logical Connectives . . . 160
2.5.3 Atomic Formulas . . . 162
2.5.4 Terms . . . 165
2.6 Identifiers . . . 168

3 Subsorting Specification Semantics 169
3.1 Subsorting Concepts 169
3.1.1 Signatures 169
3.1.2 Models 173
3.1.3 Sentences 174
3.2 Signature Declarations 175
3.2.1 Sorts 175
3.2.2 Datatypes 177
3.3 Axioms 184
3.3.1 Atomic Formulas 184
3.3.2 Terms 187

4 Structured Specification Semantics 189
4.1 Structuring Concepts 189
4.1.1 Institution Independence and the CASL Institution 190
4.1.2 Derived Notions 193
4.1.3 Signature Morphisms 196
4.1.4 Extended Signatures 200
4.1.5 Institution Independent Structuring Concepts 201
4.2 Structured Specifications 204
4.2.1 Translations 205
4.2.2 Reductions 206
4.2.3 Unions 207
4.2.4 Extensions 208
4.2.5 Free Specifications 209
4.2.6 Local Specifications 210
4.2.7 Closed Specifications 210
4.3 Named and Generic Specifications 211
4.3.1 Specification Definitions 211
4.3.2 Specification Instantiation 214
4.4 Views 216
4.4.1 View Definitions 216
4.4.2 Fitting Views 218
4.5 Symbol Lists and Mappings 220
4.5.1 Symbol Lists 221
4.5.2 Symbol Mappings 222
4.6 Compound Identifiers 223

5 Architectural Specification Semantics 227
5.1 Architectural Concepts 228
5.2 Architectural Specification Definitions 232
5.3 Unit Declarations and Definitions 234
5.3.1 Unit Declarations 235
5.3.2 Unit Definitions 236

5.4 Unit Specifications 237
5.4.1 Unit Types 238
5.4.2 Architectural Unit Specifications 239
5.4.3 Closed Unit Specifications 239
5.5 Unit Expressions 240
5.5.1 Unit Terms 242
5.6 Extended Static Semantics 247
5.6.1 Architectural Concepts 248
5.6.2 Architectural Specification Definitions 251
5.6.3 Unit Declarations and Definitions 253
5.6.4 Unit Specifications 255
5.6.5 Unit Expressions 255
5.6.6 Discussion 262

6 Specification Library Semantics 265
6.1 Library Concepts 266
6.2 Local Libraries 268
6.3 Distributed Libraries 270
6.4 Library Names 271

Part IV Casl Logic

Editor: Till Mossakowski (University of Bremen, Germany)

*Authors: Till Mossakowski (University of Bremen, Germany),
Piotr Hoffman (Warsaw University, Poland),
Serge Autexier (DFKI Saarbrücken, Germany), and
Dieter Hutter (DFKI Saarbrücken, Germany)*

1 Introduction 275
1.1 Institution Independence 276
1.2 Style of the Proof Calculi 277
1.3 Soundness and Completeness 277

2 Basic Specification Calculus 279

3 Subsorting Specification Calculus 287

4 Structured Specification Calculus 289
4.1 Institution Independence 290
4.2 Development Graphs 293
4.3 Translating Development Graphs along Institution Comorphisms 297

4.4 Proof Rules for Development Graphs 298
4.4.1 Hiding Decomposition Rules 299
4.4.2 Conservativity Rules 303
4.4.3 Simple Structural Rules 307
4.5 Soundness and Completeness 308
4.6 Checking Conservativity and Freeness 310
4.7 Translation from Structured Specifications to Development Graphs 311
4.7.1 Concepts for the Verification Semantics 312
4.7.2 Structured Specifications 317
4.7.3 Named and Generic Specifications 320
4.7.4 Views 322
4.7.5 Adequacy of the Translation 324

5 Architectural Specification Calculus 329
5.1 Semantics 330
5.1.1 Static and Model Semantics 330
5.1.2 Extended Static Semantics 334
5.2 Soundness and Completeness of the Extended Static Semantics 338
5.2.1 Concepts 338
5.2.2 Proof 341
5.3 The Proof Calculus 347
5.3.1 Definition of the Proof Calculus 348
5.3.2 Soundness and Completeness 353

6 Specification Library Calculus 357

Part V CASL Libraries

Authors: Markus Roggenbach (University of Wales Swansea, United Kingdom), Till Mossakowski (University of Bremen, Germany), and Lutz Schröder (University of Bremen, Germany)

1 Introduction 363
1.1 A Short Overview of the Specified Datatypes 364
1.2 The Library Basic/Numbers 365
1.3 The Library Basic/RelationsAndOrders 368
1.4 The Library Basic/Algebra_I 369
1.5 The Library Basic/SimpleDatatypes 370
1.6 The Library Basic/StructuredDatatypes 370
1.7 The Library Basic/Graphs 372
1.8 The Library Basic/Algebra_II 374
1.9 The Library Basic/LinearAlgebra_I 375
1.10 The Library Basic/LinearAlgebra_II 377
1.11 The Library Basic/MachineNumbers 377

2 Library Basic/Numbers 379

3 Library Basic/RelationsAndOrders 387

4 Library Basic/Algebra_I 393

5 Library Basic/SimpleDatatypes 401

6 Library Basic/StructuredDatatypes 405

7 Library Basic/Graphs 421

8 Library Basic/Algebra_II 431

9 Library Basic/LinearAlgebra_I 439

10 Library Basic/LinearAlgebra_II 449

11 Library Basic/MachineNumbers 453

12 Dependency Graphs of the Libraries 459

Appendices

Annotated Bibliography 469

References 487

Index of Library and Specification Names 491

Abstract Syntax Sorts and Constructors 495

Symbol Index 501

Concept Index 511

Part I

CASL Summary

The CoFI Language Design Group

Editors: Bernd Krieg-Brückner and Peter D. Mosses

P.D. Mosses (Ed.): CASL Reference Manual, LNCS 2960, pp. 1–69, 2004.

1

Introduction

This part of the Casl Reference Manual gives a detailed summary of the syntax and intended semantics of Casl. Readers are assumed to be already familiar with the main concepts of algebraic specifications.

Chapter 2 summarizes *many-sorted basic specifications* in Casl, which denote classes of many-sorted partial first-order structures: algebras where the functions are partial or total, and where also predicates are allowed. Axioms are first-order formulas built from equations and definedness assertions. Sort generation constraints can be stated. Datatype declarations are provided for concise specification of sorts together with constructors and (optional) selectors.

Chapter 3 summarizes *subsorted basic specifications*, which extend many-sorted specifications with a simple treatment of subsorts, interpreting subsort inclusion as embedding.

Chapter 4 summarizes *structured specifications*, which allow translation, reduction, union, and extension of specifications. Extensions may be required to be free; initiality constraints are a special case. A simple form of generic specifications is provided, together with instantiation involving parameter-fitting translations and views.

Chapter 5 summarizes *architectural specifications*, which define how the specified software is to be composed from a given set of separately-developed, reusable units with clear interfaces.

Chapter 6 summarizes *specification libraries*, which allow the (distributed) storage and retrieval of named specifications.

Finally, Chap. 7 (by Till Mossakowski) summarizes various *sublanguages* and *extensions* of Casl.

In general, each chapter first summarizes the main *semantic concepts* underlying the kind of specification concerned, then it presents the (abstract and concrete) syntax of the associated Casl *language constructs* and indicates their intended semantics. See Part II of this reference manual for complete grammars for the abstract and concrete syntax of Casl, and Part III for the formal semantics of Casl.

This summary does not attempt to motivate the design choices that have been taken; a rationale for a preliminary design has been published separately [49], as has a full exposition of architectural specifications [6]. See also [1] for a concise overview of Casl, and [5] for a tutorial introduction.

Acknowledgement. The CoFI Language Design Group was formed at the founding meeting of the Common Framework Initiative in Oslo, September 1995. Language design working meetings were held in Paris (November 1995), Munich (January 1996), Oxford (March 1996), Paris (May 1996), Munich (July 1996), Edinburgh (November 1996), Paris (January and April 1997), Amsterdam (September 1997), Bremen (January 1998), Lisbon (April 1998), Amsterdam (April 1999), Berlin (April 2000), and Genova (April 2001). The earlier meetings were mostly hosted by Michel Bidoit, Bernd Krieg-Brückner, Don Sannella, and Martin Wirsing; the later meetings were co-located with major conferences. Notes recording the discussions and decisions at the meetings were produced by Christine Choppy.

The following persons contributed to the design of Casl– some of them over many years, others only occasionally – by studying the issues and making suggestions: Egidio Astesiano, Hubert Baumeister, Jan Bergstra, Gilles Bernot, Didier Bert, Mohammed Bettaz, Michel Bidoit, Mark van den Brand, Maria Victoria Cengarle, Maura Cerioli, Christine Choppy, Pietro Cenciarelli, Ole-Johan Dahl, Hans-Dieter Ehrich, Hartmut Ehrig, José Fiadeiro, Marie-Claude Gaudel, Chris George, Joseph Goguen, Radu Grosu, Magne Haveraaen, Anne Haxthausen, Jim Horning, Hélène Kirchner, Kolyang, Hans-Jörg Kreowski, Bernd Krieg-Brückner, Pierre Lescanne, Christoph Lüth, Tom Maibaum, Grant Malcolm, Karl Meinke, Till Mossakowski, Peter Mosses, Peter Padawitz, Fernando Orejas, Olaf Owe, Gianna Reggio, Horst Reichel, Markus Roggenbach, Erik Saaman, Don Sannella, Giuseppe Scollo, Amilcar Sernadas, Andrzej Tarlecki, Christophe Tronche, Eelco Visser, Frédéric Voisin, Eric Wagner, Michał Walicki, Bjarke Wedemeijer, Martin Wirsing, Uwe Wolter, and Alexandre Zamulin.

The acronym Casl for the Common Algebraic Specification Language was proposed by Christine Choppy.

The design of the abstract syntax and semantics of Casl was much influenced by the work of the CoFI Semantics Group, mainly consisting of Hubert Baumeister, Maura Cerioli, Anne Haxthausen, Till Mossakowski, Don Sannella, and Andrzej Tarlecki. (See Part II regarding the design of the concrete syntax.)

The IFIP WG1.3 Referees' Report on Casl reviewed the initial design proposal for Casl (version 0.97, May 1997); the Casl Designers' final response to the referees indicated how the points raised in the report had influenced the final design (version 1.0.1-DRAFT, June 2000, approved and released as version 1.0.1 in March 2001)[1]. The IFIP WG1.3 reviewers consisted of Hartmut Ehrig (Coordinator), José Meseguer, Ugo Montanari, Fernando Orejas, Peter Padawitz, Francesco Parisi-Presicce, Martin Wirsing, and Uwe Wolter.

The coordinator of the Language Design task group during the design of Casl was Bernd Krieg-Brückner.

[1] The original design documents and the reviews are available from the CoFI Archives [16].

2

Basic Specifications

Basic specifications in Casl allow declaration of sorts, subsorts, operations (both total and partial), and predicates, and the use of formulas of first-order logic for stating axioms. Subsorts can be defined by formulas. Sorts can be constrained to include only generated values. Both loose and free datatypes with constructor and (optionally) selector operations can be declared concisely.

Section 2.1 introduces the concepts underlying *many-sorted* basic specifications, and the remaining sections cover the language constructs provided by Casl for use in such specifications: Sect. 2.2 describes the overall structure of basic specifications; Sect. 2.3 introduces declarations of sorts, operations, and predicates; Sect. 2.4 deals with variable declarations; Sect. 2.5 summarizes the formulas and terms used in axioms; and Sect. 2.6 indicates the form of identifiers. The concepts and Casl constructs concerned with subsorts are summarized separately, in Chap. 3.

2.1 Basic Concepts

First, before considering the particular concepts underlying basic specifications in Casl, here is a brief reminder of how specification frameworks in general may be formalized in terms of so-called *institutions* [20] (some category-theoretic details are omitted) and *proof systems*.

A *basic specification framework* may be characterized by:

- a class **Sig** of *signatures* Σ, each determining the set of *symbols* $|\Sigma|$ whose intended interpretation is to be specified, with *morphisms* between signatures;
- a class $\mathbf{Mod}(\Sigma)$ of *models*, with *homomorphisms* between them, for each signature Σ;
- a set $\mathbf{Sen}(\Sigma)$ of *sentences* (or *axioms*), for each signature Σ;
- a relation $\models$ of *satisfaction*, between models and sentences over the same signature; and
- a *proof system*, for inferring sentences from sets of sentences.

A *basic specification* consists of a signature Σ together with a set of sentences from $\mathbf{Sen}(\Sigma)$. The signature provided for a particular declaration or sentence in a specification is called its *local environment.* It may be a restriction of the entire signature of the specification, e.g., determined by an order of *presentation* for the signature declarations and the sentences with *linear visibility*, where symbols may not be used before they have been declared; or it may be the entire signature, reflecting *non-linear visibility.*

The (loose) *semantics* of a basic specification is the class of those models in $\mathbf{Mod}(\Sigma)$ which satisfy all the specified sentences. A specification is said to be *consistent* when there are some models that satisfy all the sentences, and *inconsistent* when there are no such models. A sentence is a *consequence* of a basic specification if it is satisfied in all the models of the specification.

A *signature morphism* $\sigma : \Sigma \to \Sigma'$ determines a *translation* function $\mathbf{Sen}(\sigma)$ on sentences, mapping $\mathbf{Sen}(\Sigma)$ to $\mathbf{Sen}(\Sigma')$, and a *reduct* function $\mathbf{Mod}(\sigma)$ on models, mapping $\mathbf{Mod}(\Sigma')$ to $\mathbf{Mod}(\Sigma)$ [1]. Satisfaction is required to be preserved by translation: for all $S \in \mathbf{Sen}(\Sigma), M' \in \mathbf{Mod}(\Sigma')$,

$$\mathbf{Mod}(\sigma)(M') \models S \iff M' \models \mathbf{Sen}(\sigma)(S).$$

The proof system is required to be sound, i.e., sentences inferred from a specification are always consequences; moreover, inference is to be preserved by translation.

Sentences of basic specifications may include *constraints* that restrict the class of models, e.g., to reachable ones.

The rest of this chapter considers many-sorted basic specifications of the Casl specification framework, and indicates the underlying signatures, models, and sentences[2]. Then the syntax of the language constructs used for expressing many-sorted basic specifications is described. Consideration of the extra features concerned with subsorts is deferred to Chap. 3. The abstract syntax of any well-formed basic specification determines a signature and a set of sentences, the models of which provide the semantics of the basic specification.

2.1.1 Signatures

A *many-sorted signature* $\Sigma = (S, TF, PF, P)$ consists of:

- a set S of *sorts*;
- sets $TF_{w,s}$, $PF_{w,s}$, of *total function symbols*, respectively *partial function symbols*, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each *function profile* (w, s) consisting of a sequence of *argument sorts* $w \in S^*$ and a *result sort* $s \in S$ (*constants* are treated as functions with no arguments);
- sets P_w of *predicate* symbols, for each *predicate profile* consisting of a sequence of argument sorts $w \in S^*$.

[1] In fact **Sig** is a category, and **Sen**(.) and **Mod**(.) are functors. The categorial aspects of the semantics of Casl are emphasized in its formal semantics in Part III.

[2] A particular proof system for Casl is provided in Part IV.

Constants and functions are also referred to as *operations*, following the traditions of algebraic specification.

Note that symbols used to identify sorts, operations, and predicates may be *overloaded*, occurring in more than one of the above sets. To ensure that there is no ambiguity in sentences at this level, however, function symbols f and predicate symbols p are always *qualified* by profiles when used, written $f_{w,s}$ and p_w respectively. (The language described later in this chapter allows the omission of such qualifications when these are unambiguously determined by the context.)

A *many-sorted signature morphism* $\sigma : (S, \mathit{TF}, \mathit{PF}, P) \rightarrow (S', \mathit{TF}', \mathit{PF}', P')$ consists of a mapping from S to S', and for each $w \in S^*, s \in S$, a mapping between the corresponding sets of function, resp. predicate symbols. A partial function symbol may be mapped also to a total function symbol, but not vice versa.

2.1.2 Models

For a many-sorted signature $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$ a *many-sorted model* $M \in \mathbf{Mod}(\Sigma)$ is a *many-sorted first-order structure* consisting of a *many-sorted partial algebra*:

- a non-empty *carrier set* s^M for each sort $s \in S$ (let w^M denote the Cartesian product $s_1^M \times \cdots \times s_n^M$ when $w = s_1 \ldots s_n$),
- a *partial function* f^M from w^M to s^M for each function symbol $f \in \mathit{TF}_{w,s}$ or $f \in \mathit{PF}_{w,s}$, the function being required to be total in the former case,

together with:

- a *predicate* $p^M \subseteq w^M$ for each predicate symbol $p \in P_w$.

A (weak) *many-sorted homomorphism* h from M_1 to M_2, with $M_1, M_2 \in \mathbf{Mod}(S, \mathit{TF}, \mathit{PF}, P)$, consists of a function $h_s : s^{M_1} \rightarrow s^{M_2}$ for each $s \in S$ preserving not only the values of functions but also their definedness, and preserving the truth of predicates [14].

Any signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ determines the *many-sorted reduct* of each model $M' \in \mathbf{Mod}(\Sigma')$ to a model $M \in \mathbf{Mod}(\Sigma)$, defined by interpreting symbols of Σ in M in the same way that their images under σ are interpreted in M'.

2.1.3 Sentences

The *many-sorted terms* on a signature $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$ and a set of sorted, non-overloaded variables X are built from:

- variables from X;
- applications of qualified function symbols in $\mathit{TF} \cup \mathit{PF}$ to argument terms of appropriate sorts.

We refer to such terms as *fully-qualified terms*, to avoid confusion with the terms of the language considered later in this chapter, which allow the omission of qualifications and explicit sorts when these are unambiguously determined by the context.

For a many-sorted signature $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$ the *many-sorted sentences* in $\mathbf{Sen}(\Sigma)$ are the usual closed many-sorted first-order logic formulas, built from atomic formulas using quantification (over sorted variables) and logical connectives. An inner quantification over a variable makes a hole in the scope of an outer quantification over the same variable, regardless of the sorts of the variables. Implication may be taken as primitive (in the presence of an always-false formula), the other connectives being regarded as derived.

The *atomic formulas* are:

- applications of qualified predicate symbols $p \in P$ to argument terms of appropriate sorts;
- assertions about the definedness of fully-qualified terms;
- existential and strong equations between fully-qualified terms of the same sort.

Definedness assertions may be derived from existential equations or regarded as applications of fixed, always-true predicates. Strong equations may be derived from existential equations, using implication and conjunction; existential equations may be derived from conjunctions of strong equations and definedness assertions, or regarded as applications of fixed predicates.

The sentences $\mathbf{Sen}(\Sigma)$ also include *sort-generation constraints*. Let $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$. A sort-generation constraint consists of (S', F') with $S' \subseteq S$ and $F' \subseteq \mathit{TF} \cup \mathit{PF}$ [3].

2.1.4 Satisfaction

The satisfaction of a sentence in a structure M is determined as usual by the holding of its atomic formulas w.r.t. assignments of (defined) values to all the variables that occur in them, the values assigned to variables of sort s being in s^M. The value of a term w.r.t. a variable assignment may be undefined, due to the application of a partial function during the evaluation of the term. Note, however, that the satisfaction of sentences is two-valued (as is the holding of open formulas with respect to variable assignments).

The application of a predicate symbol p to a sequence of argument terms holds in M iff the values of all the terms are defined and give a tuple belonging to p^M. A definedness assertion concerning a term holds iff the value of the term is defined (thus it corresponds to the application of a constantly-true unary predicate to the term). An existential equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined.

[3] The translation of such constraints along signature morphisms adds a further component, for technical reasons.

The value of an occurrence of a variable in a term is that provided by the given variable assignment. The value of the application of a function symbol f to a sequence of argument terms is defined only if the values of all the argument terms are defined and give a tuple in the domain of definedness of f^M, and then it is the associated result value.

A sort-generation constraint (S', F') is satisfied in a Σ-model M if the carriers of the sorts in S' are *generated* by the function symbols in F'. That is, every element of each sort in S' is the value of a term built from just these symbols (possibly using variables of sorts not in S', with appropriate assignments of values to them).

The rest of this chapter indicates the abstract and concrete syntax of the constructs of many-sorted basic specifications, and describes their intended interpretation.

2.2 Basic Items

For an introduction to the form of grammar used here to define the abstract syntax of language constructs, see Chap. II:2, which also provides the complete grammar defining the abstract syntax of the entire CASL specification language.

```
BASIC-SPEC ::= basic-spec BASIC-ITEMS*
```

A many-sorted basic specification `BASIC-SPEC` in the CASL language is written simply as a sequence of `BASIC-ITEMS` constructs:

$BI_1 \ldots BI_n$

The empty basic specification is not usually needed, but can be written '**{ }**'.

This language construct determines a basic specification within the underlying many-sorted institution, consisting of a signature and a set of sentences of the form described at the beginning of this chapter. This signature and the class of models over it that satisfy the set of sentences provide the *semantics* of the basic specification. Thus this chapter explains well-formedness of basic specifications, and the way that they determine the underlying signatures and sentences, rather than directly explaining the intended interpretation of the constructs.

While *well-formedness* of specifications in the language can be checked statically, the question of whether the value of a term that occurs in a well-formed specification is necessarily defined in all models may depend on the specified axioms (and it is not decidable in general).

```
BASIC-ITEMS ::= SIG-ITEMS | FREE-DATATYPE | SORT-GEN
              | VAR-ITEMS | LOCAL-VAR-AXIOMS | AXIOM-ITEMS
```

A `BASIC-ITEMS` construct is always a list, written:

plural-keyword $X_1; \ldots\ X_n;$

The *plural-keyword* may also be written in the singular (regardless of the number of items), and the final '`;`' may be omitted.

Each `BASIC-ITEMS` construct determines part of a signature and/or some sentences (except for `VAR-ITEMS`, which merely declares some global variables). The order of the basic items is generally significant: there is *linear visibility* of declared symbols and variables in a list of `BASIC-ITEMS` constructs (except within a list of datatype declarations). Repeated declaration of a symbol is allowed, and does not affect the semantics; some tools may however be able to locate and warn about such duplications, in case they were not intentional.

A list of signature declarations and definitions `SIG-ITEMS` determines part of a signature and possibly some sentences. A `FREE-DATATYPE` construct determines part of a signature together with some sentences. A sort-generation construct `SORT-GEN` determines part of a signature, together with some sentences including a corresponding sort generation constraint. A list of variable declaration items `VAR-ITEMS` determines sorted variables that are implicitly universally quantified in the subsequent axioms of the enclosing basic specification; note that variable declarations do not contribute to the signature of the specification in which they occur. A `LOCAL-VAR-AXIOMS` construct restricts the scope of the variable declarations to the indicated list of axioms. (Variables may also be declared locally in individual axioms, by explicit quantification.) An `AXIOM-ITEMS` construct determines a set of sentences.

2.3 Signature Declarations

```
SIG-ITEMS ::= SORT-ITEMS | OP-ITEMS | PRED-ITEMS | DATATYPE-ITEMS
```

A list `SORT-ITEMS` of sort declarations determines one or more sorts. A list `OP-ITEMS` of operation declarations and/or definitions determines one or more operation symbols, and possibly some sentences; similarly for a list `PRED-ITEMS` of predicate declarations and/or definitions. Operation and predicate symbols may be overloaded, being declared with several different profiles in the same local environment. A list `DATATYPE-ITEMS` of datatype declarations determines one or more sorts together with some constructor and (optional) selector operations, and sentences defining the selector operations on the values given by the constructors with which they are associated.

2.3.1 Sorts

```
SORT-ITEMS ::= sort-items SORT-ITEM+
SORT-ITEM  ::= SORT-DECL
```

A list `SORT-ITEMS` of sort declarations is written:

sorts $SI_1; \ldots SI_n;$

Sort Declarations

```
SORT-DECL ::= sort-decl SORT+
SORT      ::= SORT-ID
```

A sort declaration `SORT-DECL` is written:

$s_1, \ldots, s_n$

It declares each of the sorts in the list s_1, ..., s_n.

2.3.2 Operations

```
OP-ITEMS ::= op-items OP-ITEM+
OP-ITEM  ::= OP-DECL | OP-DEFN
```

A list `OP-ITEMS` of operation declarations and definitions is written:

ops OI_1; ... OI_n;

Operation Declarations

```
OP-DECL ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
OP-NAME ::= ID
```

An operation declaration `OP-DECL` is written:

$f_1, \ldots, f_n : TY, a_1, \ldots, a_m$

When the list a_1, ..., a_m is empty, the declaration is written simply:

$f_1, \ldots, f_n : TY$

It declares each operation name f_1, ..., f_n as a total or partial operation, with profile as specified by the operation type TY, and as having the attributes a_1, ..., a_m (if any). If an operation is declared both as total and as partial with the same profile, the resulting signature only contains the total operation.

Operation Types

```
OP-TYPE         ::= TOTAL-OP-TYPE | PARTIAL-OP-TYPE
TOTAL-OP-TYPE   ::= total-op-type SORT-LIST SORT
PARTIAL-OP-TYPE ::= partial-op-type SORT-LIST SORT
SORT-LIST       ::= sort-list SORT*
```

A total operation type `TOTAL-OP-TYPE` with some argument sorts is written:

$s_1 \times \ldots \times s_n \rightarrow s$

When the list of argument sorts is empty, the type is simply written 's'. The sign displayed as '×' may be input as '×' in ISO Latin-1, or as '`*`' in ASCII. The sign displayed as '→' is input as '`->`'.

A partial operation type `PARTIAL-OP-TYPE` with some argument sorts is written:

$$s_1 \times \ldots \times s_n \to ?\ s$$

When the list of argument sorts is empty, the type is simply written '? s'.

The operation profile determined by the type has argument sorts $s_1, \ldots, s_n$ and result sort s.

Operation Attributes

```
OP-ATTR        ::= BINARY-OP-ATTR | UNIT-OP-ATTR
BINARY-OP-ATTR ::= assoc-op-attr | comm-op-attr | idem-op-attr
UNIT-OP-ATTR   ::= unit-op-attr TERM
```

Operation attributes assert that the operations being declared (which must be binary) have certain common properties, which are characterized by strong equations, universally quantified over variables of the appropriate sort. (This can also be used to add attributes to operations that have previously been declared without them.)

The attribute `assoc-op-attr` is written '*assoc*'. It asserts the *associativity* of an operation f:

$$f(x, f(y, z)) = f(f(x, y), z)$$

The attribute of associativity moreover implies a local parsing annotation (see Sect. II:5.2.3) that allows an infix operation f of the form '`__`t`__`' (or '`__ __`') to be iterated without explicit grouping parentheses.

The attribute `comm-op-attr` is written '*comm*'. It asserts the *commutativity* of an operation f:

$$f(x, y) = f(y, x)$$

The attribute `idem-op-attr` is written '*idem*'. It asserts the *idempotency* of an operation f:

$$f(x, x) = x$$

The attribute `UNIT-OP-ATTR` is written '*unit T*'. It asserts that the value of the term T is the *unit* (left and right) of an operation f:

$$f(T, x) = x \wedge f(x, T) = x$$

In practice, the unit T is normally a constant. In any case, T must not contain any free variables (i.e., variables that are not explicitly declared by enclosing quantifications).

The declaration enclosing an operation attribute is ill-formed unless the operation profile has exactly two argument sorts, both the same, which, except in the case of commutativity, have also to be the same as the result sort.

Operation Definitions

```
OP-DEFN         ::= op-defn OP-NAME OP-HEAD TERM
OP-HEAD         ::= TOTAL-OP-HEAD | PARTIAL-OP-HEAD
TOTAL-OP-HEAD   ::= total-op-head ARG-DECL* SORT
PARTIAL-OP-HEAD ::= partial-op-head ARG-DECL* SORT
ARG-DECL        ::= arg-decl VAR+ SORT
```

A definition `OP-DEFN` of a total operation with some arguments is written:

$$f(v_{11}, \ldots, v_{1m_1} : s_1;\ \ldots;\ v_{n1}, \ldots, v_{nm_n} : s_n) : s = T$$

When the list of arguments is empty, the definition is simply written:

$$f : s = T$$

A definition `OP-DEFN` of a partial operation with some arguments is written:

$$f(v_{11}, \ldots, v_{1m_1} : s_1;\ \ldots;\ v_{n1}, \ldots, v_{nm_n} : s_n) :?\ s = T$$

When the list of arguments is empty, the definition is simply written:

$$f :?\ s = T$$

It declares the operation name f as a total, respectively partial operation, with a profile having argument sorts s_1 (m_1 times), …, s_n (m_n times) and result sort s. It also asserts the strong equation:

$$f(v_{11}, \ldots, v_{nm_n}) = T$$

universally quantified over the declared argument variables (which must be distinct, and are the only free variables allowed in T), or just '$f = T$' when the list of arguments is empty.

In each of the above cases, the operation name f may occur in the term T, and may have *any* interpretation satisfying the equation – not necessarily the least fixed point.

2.3.3 Predicates

```
PRED-ITEMS ::= pred-items PRED-ITEM+
PRED-ITEM  ::= PRED-DECL | PRED-DEFN
PRED-NAME  ::= ID
```

A list `PRED-ITEMS` of predicate declarations and definitions is written:

preds $PI_1;\ \ldots\ PI_n;$

Predicate Declarations

```
PRED-DECL ::= pred-decl PRED-NAME+ PRED-TYPE
```

A predicate declaration `PRED-DECL` is written:

$$p_1, \ldots, p_n : TY$$

It declares each predicate name p_1, …, p_n as a predicate, with profile as specified by the predicate type TY.

Predicate Types

```
PRED-TYPE ::= pred-type SORT-LIST
```

A predicate type `PRED-TYPE` with some argument sorts is written:

$s_1 \times \ldots \times s_n$

The sign displayed as '×' may be input as '×' in ISO Latin-1, or as '`*`' in ASCII. When the list of argument sorts is empty, the type is written '()'.

The predicate profile determined by the type has argument sorts $s_1, \ldots, s_n$.

Predicate Definitions

```
PRED-DEFN ::= pred-defn PRED-NAME PRED-HEAD FORMULA
PRED-HEAD ::= pred-head ARG-DECL*
```

A definition `PRED-DEFN` of a predicate with some arguments is written:

$p(v_{11}, \ldots, v_{1m_1} : s_1; \;\ldots;\; v_{n1}, \ldots, v_{nm_n} : s_n) \Leftrightarrow F$

When the list of arguments is empty, the definition is simply written:

$p \Leftrightarrow F$

The sign displayed as '$\Leftrightarrow$' is input as '`<=>`'.

It declares the predicate name p as a predicate, with a profile having argument sorts s_1 (m_1 times), ..., s_n (m_n times). It also asserts the equivalence:

$p(v_{11}, \ldots, v_{nm_n}) \Leftrightarrow F$

universally quantified over the declared argument variables (which must be distinct, and are the only free variables allowed in F), or just '$p \Leftrightarrow F$' when the list of arguments is empty. The predicate name p may occur in the formula F, and may have *any* interpretation satisfying the equivalence.

2.3.4 Datatypes

```
DATATYPE-ITEMS ::= datatype-items DATATYPE-DECL+
```

A list `DATATYPE-ITEMS` of datatype declarations is written:

types DD_1; ... DD_n;

The order of the datatype declarations is *not* significant: there is *non-linear visibility* of the declared sorts in a list (in contrast to the linear visibility between the `BASIC-ITEMS` of a `BASIC-SPEC`, and between the `SIG-ITEMS` of a `SORT-GEN`).

Datatype Declarations

```
DATATYPE-DECL ::= datatype-decl SORT ALTERNATIVE+
```

A datatype declaration DATATYPE-DECL is written:

$s ::= A_1 \mid \ldots \mid A_n$

It declares the sort s. For each alternative construct $A_1, \ldots, A_n$, it declares the specified constructor and selector operations, and determines sentences asserting the expected relationship between selectors and constructors. All sorts used in an alternative construct must be declared in the local environment (which always includes the sort declared by the datatype declaration itself). A list of datatype declarations must not declare a function symbol both as a constructor and selector with the same profiles.

Note that a datatype declaration allows models where the ranges of the constructors are not disjoint, and where not all values are the results of constructors. This looseness can be eliminated in a general way by use of free extensions in structured specifications (as summarized in Chap. 4), or by use of free datatypes within basic specifications (see below). Unreachable values can be eliminated also by the use of sort generation constraints.

Alternatives

```
ALTERNATIVE       ::= TOTAL-CONSTRUCT | PARTIAL-CONSTRUCT
TOTAL-CONSTRUCT   ::= total-construct   OP-NAME COMPONENTS*
PARTIAL-CONSTRUCT ::= partial-construct OP-NAME COMPONENTS+
```

A total constructor TOTAL-CONSTRUCT with some components is written:

$f(C_1;\ \ldots;\ C_n)$

When the list of components is empty, the constructor is simply written 'f'.

A partial constructor PARTIAL-CONSTRUCT with some components is written:

$f(C_1;\ \ldots;\ C_n)?$

(Partial constructors without components are not expressible in datatype declarations.)

The alternative declares f as an operation. Each component $C_1, \ldots, C_n$ specifies one or more argument sorts for the profile, and possibly some component selectors; the result sort is the sort declared by the enclosing datatype declaration. The selectors within each alternative must be distinct, but need not be distinct from selectors in different alternatives.

Components

```
COMPONENTS     ::= TOTAL-SELECT | PARTIAL-SELECT | SORT
TOTAL-SELECT   ::= total-select   OP-NAME+ SORT
PARTIAL-SELECT ::= partial-select OP-NAME+ SORT
```

A declaration `TOTAL-SELECT` of total selectors is written:

$f_1, \ldots, f_n : s$

A declaration `PARTIAL-SELECT` of partial selectors is written:

$f_1, \ldots, f_n :? s$

The remaining case is a component sort without any selector, simply written 's'.

In the first two cases, the component declaration provides n components: the sort s is taken as an argument sort n times for the constructor operation declared by the enclosing alternative, and it declares f_1, ..., f_n as selector operations for the respective components. In the first case, each selector operation is declared as total, and in the second case, as partial. The component declaration also determines sentences that define the value of each selector on the values given by the constructor of the enclosing alternative.

In the last case, the component declaration provides the sort s only once as an argument sort for the constructor of the enclosing alternative, and it does not declare any selector operation for that component.

Note that when there is more than one alternative construct in a datatype declaration, selectors are usually partial, and should therefore be declared as such; their values on constructs for which they are not declared as selectors are left unspecified.

Free Datatype Declarations

```
FREE-DATATYPE ::= free-datatype DATATYPE-ITEMS
```

A list `FREE-DATATYPE` of free datatype declarations is written:

free types DD_1; ... DD_n;

This construct is only well-formed when all the constructors declared by the datatype declarations are total.

Free datatype declarations declare the same sorts, constructors, and selectors as ordinary datatype declarations. Apart from the sentences that define the values of selectors, the free datatype declarations determine additional sentences requiring that the constructors are injective, that the ranges of constructors of the same sort are disjoint, that all the declared sorts are generated by the constructors, and that the value of applying a selector to a constructor for which it has not been declared is always undefined. The sentences ensure that the models, if any, are the same as for a free extension with the datatype declarations, provided that the following conditions are fulfilled (all conditions refer to fully qualified symbols):

- all the declared sorts are distinct from those in the local environment, and
- each total selector is present in all the alternatives for its argument sort.

When the alternatives of a free datatype declaration are all constants, the declared sort corresponds to an (unordered) enumeration type.

2.3.5 Sort Generation

```
SORT-GEN ::= sort-gen SIG-ITEMS+
```

A sort generation `SORT-GEN` is written:

generated { $SI_1 \ldots SI_n$ **}**;

When the list of `SIG-ITEMS` is a single `DATATYPE-ITEMS` construct, writing the grouping signs is optional:

generated types DD_1; ... DD_n;

(The terminating '`;`' is optional in both cases.)

It determines the same elements of signature and sentences as SI_1, ..., SI_n, together with a corresponding sort generation constraint sentence: all the declared sorts of SI_1, ..., SI_n are required to be generated by the declared operations of SI_1, ..., SI_n – but *excluding* operations declared as *selectors* by datatype declarations. A `SORT-GEN` is ill-formed if it does not declare any sorts.

2.4 Variables

Variables for use in terms may be declared globally, locally, or with explicit quantification. Globally or locally declared variables are implicitly universally quantified in subsequent axioms of the enclosing basic specification. Variables are not included in the declared signature.

Universal quantification over a variable that does not occur free in an axiom is semantically irrelevant, due to the assumption that all carrier sets are non-empty.

2.4.1 Global Variable Declarations

```
VAR-ITEMS ::= var-items VAR-DECL+
```

A list `VAR-ITEMS` of variable declarations is written:

vars VD_1; ... VD_n;

Note that local variable declarations are written in a similar way, but followed directly by a bullet ‘ • ’ instead of the optional semicolon.

```
VAR-DECL ::= var-decl VAR+ SORT
VAR      ::= SIMPLE-ID
```

A variable declaration VAR-DECL is written:

$$v_1, \ldots, v_n : s$$

It declares the variables v_1, …, v_n of sort s for use in subsequent axioms, but it does *not* contribute to the declared signature.

The scope of a global variable declaration is the subsequent axioms of the enclosing basic specification; a later declaration for a variable with the same identifier overrides the earlier declaration (regardless of whether the sorts of the variables are the same). A global declaration of a variable is equivalent to adding a universal quantification on that variable to the subsequent axioms of the enclosing basic specification.

2.4.2 Local Variable Declarations

```
LOCAL-VAR-AXIOMS ::= local-var-axioms VAR-DECL+ AXIOM+
```

A localization LOCAL-VAR-AXIOMS of variable declarations to a list of axioms is written:

$$\forall VD_1; \ldots; \; VD_n \; \bullet \; F_1 \ldots \; \bullet \; F_m;$$

The sign displayed as ‘∀’ is input as ‘forall’. The sign displayed as ‘ • ’ may be input as ‘·’ in ISO Latin-1, or as ‘.’ in ASCII.

It declares variables for local use in the axioms F_1, …, F_m, but it does *not* contribute to the declared signature. A local declaration of a variable is equivalent to adding a universal quantification on that variable to all the indicated axioms.

2.5 Axioms

```
AXIOM-ITEMS ::= axiom-items AXIOM+
AXIOM       ::= FORMULA
```

A list AXIOM-ITEMS of axioms is written:

$$\bullet \; F_1 \ldots \; \bullet \; F_n$$

Each well-formed axiom determines a sentence of the underlying basic specification (closed by universal quantification over all declared variables).

```
FORMULA ::= QUANTIFICATION | CONJUNCTION | DISJUNCTION
          | IMPLICATION | EQUIVALENCE | NEGATION | ATOM
```

A formula is constructed from atomic formulas of the form `ATOM` using quantification and the usual logical connectives.

Keywords in formulas and terms are displayed in the same font as identifiers.

2.5.1 Quantifications

```
QUANTIFICATION ::= quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER     ::= universal | existential | unique-existential
```

A quantification with the `universal` quantifier is written:

$$\forall VD_1; \ldots; VD_n \bullet F$$

The sign displayed as '$\forall$' is input as '`forall`'. The sign displayed as ' $\bullet$ ' may be input as '·' in ISO Latin-1, or as '`.`' in ASCII.

A quantification with the `existential` quantifier is written:

$$\exists VD_1; \ldots; VD_n \bullet F$$

A quantification with the `unique-existential` quantifier is written:

$$\exists! VD_1; \ldots; VD_n \bullet F$$

The sign displayed as '$\exists$' is input as '`exists`'.

The first case is universal quantification, holding when the body F holds for all values of the quantified variables; the second case is existential quantification, holding when the body F holds for some values of the quantified variables; and the last case is unique existential quantification, abbreviating a formula that holds when the body F holds for unique values of the quantified variables.

The formula $\forall VD_1; \ldots; VD_n \bullet F$ is equivalent to $\forall VD_1 \bullet \ldots \forall VD_n \bullet F$; and $\forall v_1, \ldots, v_n : s \bullet F$ is equivalent to $\forall v_1 : s \bullet \ldots \forall v_n : s \bullet F$. Similarly for the other quantifiers. The scope of a variable declaration in a quantification is the component formula F, and an inner declaration for a variable with the same identifier as in an outer declaration overrides the outer declaration (regardless of whether the sorts of the variables are the same). Note that the body of a quantification extends as far as possible.

2.5.2 Logical Connectives

The usual logical connectives are provided. Conjunction and disjunction apply to lists of two or more formulas; they both have weaker precedence than negation. When mixed, they have to be explicitly grouped, using parentheses '(...)'.

Both implication (which may be written in two different ways) and equivalence have weaker precedence than conjunction and disjunction. When the ‘forward’ version of implication is iterated, it is implicitly grouped to the right; the ‘backward’ version is grouped to the left. When these constructs are mixed, they have to be explicitly grouped.

Conjunction

```
CONJUNCTION ::= conjunction FORMULA+
```

A conjunction is written:

$$F_1 \wedge \ldots \wedge F_n$$

The sign displayed as ‘$\wedge$’ is input as ‘`/\`’.

Disjunction

```
DISJUNCTION ::= disjunction FORMULA+
```

A disjunction is written:

$$F_1 \vee \ldots \vee F_n$$

The sign displayed as ‘$\vee$’ is input as ‘`\/`’.

Implication

```
IMPLICATION ::= implication FORMULA FORMULA
```

An implication is written:

$$F_1 \Rightarrow F_2$$

The sign displayed as ‘$\Rightarrow$’ is input as ‘`=>`’. An implication may also be written in reverse order:

$$F_2 \ \mathit{if} \ F_1$$

Equivalence

```
EQUIVALENCE ::= equivalence FORMULA FORMULA
```

An equivalence is written:

$$F_1 \Leftrightarrow F_2$$

The sign displayed as ‘$\Leftrightarrow$’ is input as ‘`<=>`’.

Negation

```
NEGATION ::= negation FORMULA
```

A negation is written:

$\neg F_1$

The sign displayed as '¬' may be input as '¬' in ISO Latin-1, or as '`not`' in ASCII.

2.5.3 Atomic Formulas

```
ATOM ::= TRUTH | PREDICATION | DEFINEDNESS
       | EXISTL-EQUATION | STRONG-EQUATION
```

An *atomic formula* `ATOM` is well-formed (with respect to the local environment and variable declarations) if it is well-sorted and expands to a unique atomic formula for constructing sentences. The notions of when an atomic formula is *well-sorted*, of when a term is *well-sorted for a particular sort*, and of the *expansions* of atomic formulas and terms, are indicated below for the various constructs.

Due to overloading of predicate and/or operation symbols, a well-sorted atomic formula or term may have several expansions, preventing it from being well-formed. Qualifications on operation and predicate symbols may be used to determine the intended expansion and make it well-formed; explicit sorts on arguments and/or results may also help to avoid unintended expansions.

Truth

```
TRUTH ::= true-atom | false-atom
```

The atomic formulas `true-atom` and `false-atom` are written '*true*', resp. '*false*'.

They are always well-sorted, and expand to primitive sentences, such that the sentence for '*true*' always holds, and the sentence for '*false*' never holds.

Predicate Application

```
PREDICATION    ::= predication PRED-SYMB TERMS
PRED-SYMB      ::= PRED-NAME | QUAL-PRED-NAME
QUAL-PRED-NAME ::= qual-pred-name PRED-NAME PRED-TYPE
```

An application of a predicate symbol *PS* to some argument terms is written:

$PS(T_1, \ldots, T_n)$

When PS is a mixfix identifier (cf. Sect. 2.6) consisting of a sequence 't_0__ . . . __t_n' of possibly-empty mixfix tokens t_i separated by place-holders '__', the application may also be written:

$$t_0\, T_1 \ldots T_n\, t_n$$

When the predicate symbol is a constant p with no argument terms, its application is simply written 'p'.

A qualified predicate name `QUAL-PRED-NAME` with type TY is written:

$$(\mathit{pred}\ \ p : TY)$$

An unqualified predicate name `PRED-NAME` is simply written 'p'.

The application of the predicate symbol is well-sorted when there is a declaration of the predicate name (with the argument sorts indicated by the indicated type in the case of a qualified predicate name) such that all the argument terms are well-sorted for the respective argument sorts. It then expands to an application of the qualified predicate name to the fully-qualified expansions of the argument terms for those sorts.

Definedness

```
DEFINEDNESS ::= definedness TERM
```

A definedness formula is written:

$$\mathit{def}\ T$$

It is well-sorted when the term is well-sorted for some sort. It then expands to a definedness assertion on the fully-qualified expansion of the term.

Equations

```
EXISTL-EQUATION ::= existl-equation TERM TERM
STRONG-EQUATION ::= strong-equation TERM TERM
```

An existential equation `EXISTL-EQUATION` is written:

$$T_1 \overset{e}{=} T_2$$

The sign displayed as '$\overset{e}{=}$' is input as '`=e=`'.

A strong equation is written:

$$T_1 = T_2$$

An existential equation holds when the values of the terms are both defined and equal; a strong equation holds also when the values of both terms are undefined (thus the two forms of equation are equivalent when the values of both terms are always defined).

An equation is well-sorted if there is a sort such that both terms are well-sorted for that sort. It then expands to the corresponding existential or strong equation on the fully-qualified expansions of the terms for that sort.

2.5.4 Terms

```
TERM ::= SIMPLE-ID | QUAL-VAR | APPLICATION
       | SORTED-TERM | CONDITIONAL
```

A term is constructed from constants and variables by applications of operations. All names used in terms may be qualified by the intended types, and the intended sort of the term may be specified. Note that the condition of a conditional term is a formula, not a term.

Identifiers

An unqualified simple identifier in a term may be a variable or a constant, depending on the local environment and the variable declarations. Either is well-sorted for the sort specified in its declaration; a variable expands to the (sorted) variable itself, whereas a constant expands to an application of the qualified symbol to the empty list of arguments. Note that when an identifier is declared both as variable and as a constant of the same sort, unqualified use of the identifier always makes the enclosing atomic formula ill-formed.

Qualified Variables

```
QUAL-VAR ::= qual-var VAR SORT
```

A qualified variable `QUAL-VAR` is written:

$(var\ \ v : s)$

It is well-sorted for the sort s if v has been declared accordingly.

Operation Application

```
APPLICATION  ::= application OP-SYMB TERMS
OP-SYMB      ::= OP-NAME | QUAL-OP-NAME
QUAL-OP-NAME ::= qual-op-name OP-NAME OP-TYPE
TERMS        ::= terms TERM*
```

An application of an operation symbol OS to some argument terms is written:

$OS(T_1, \ldots, T_n)$

When OS is a mixfix identifier (cf. Sect. 2.6) consisting of a sequence 't_0__ $\ldots$ __t_n' of possibly-empty mixfix tokens t_i separated by place-holders '__', the application may also be written:

$t_0\, T_1 \ldots T_n\, t_n$

When the operation symbol is a constant c with no argument terms, its application is simply written 'c'.

Declaring different mixfix identifiers that involve some common tokens may lead to ambiguity, with different candidate groupings of the same sequence of tokens and terms. Such ambiguity prevents the enclosing atomic formula from being well-formed, irrespective of the declared profiles of the symbols involved, and generally has to be eliminated by use of explicit grouping parentheses. However, to allow the omission of some parentheses, infix identifiers are given weaker precedence than prefix identifiers, which in turn are given weaker precedence than postfix identifiers. (The mixfix identifier '__ __' is allowed, and regarded as an infix, although this is unlikely to be the case in higher-order extensions of Casl, since there juxtaposition will be reserved for function application.)

In an application, a qualified operation name `QUAL-OP-NAME` with f qualified by the operation type TY is written:

$(op\ \ f : TY)$

When the qualified operation name is a constant c, its application (to no arguments) is written $(op\ \ c : TY)$.

The application is well-sorted for some particular sort when there is a declaration of the operation name (with the argument and result sorts indicated by the type, if specified) such that all the argument terms are well-sorted for the respective argument sorts, and the result sort is the required sort. It then expands to an application of the qualified operation name to the fully-qualified expansions of the argument terms for those sorts.

Sorted Terms

```
SORTED-TERM ::= sorted-term TERM SORT
```

A sorted term is written:

$T : s$

It is well-sorted for some sort if the component term T is well-sorted for the specified sort s. It then expands to those of the fully-qualified expansions of the component term that have the specified sort.

Conditional Terms

```
CONDITIONAL ::= conditional TERM FORMULA TERM
```

A conditional term is written:

$T_1\ when\ F\ else\ T_2$

It is well-sorted for some sort when both T_1 and T_2 are well-sorted for that sort and F is a well-formed formula. The enclosing *atomic* formula ‘$A[T_1$ *when* F *else* $T_2]$’ expands to ‘$(A[T_1]$ *if* $F) \wedge (A[T_2]$ *if* $\neg F)$’. When several conditional terms occur in the same atomic formula, the expansions are made in a fixed but arbitrary order (all orders yield equivalent formulas).

2.6 Identifiers

```
SIMPLE-ID ::= WORDS
SORT-ID   ::= WORDS
ID        ::= id TOKEN
```

The internal structure of identifiers, used to identify sorts, operations, predicates, and variables, is insignificant in the abstract syntax of basic specifications. (SORT-ID and ID are extended with compound identifiers, whose structure is significant, in connection with generic specifications in Sect. 4.6.)

```
TOKEN     ::= WORDS | DOT-WORDS | SIGNS | DIGIT | QUOTED-CHAR
```

In concrete syntax, an identifier may be written as a single *token*: either a sequence of words – each consisting of letters, digits, and primes ('), the first word starting with a letter) separated by single underscores (_) and possibly prefixed by a dot (.) – or a sequence of other printable ISO Latin-1 characters (excluding ( ) ; , ‘ " %). Keywords, and various other sequences that could be confused with separators, are not allowed as tokens in the input syntax (however, *display annotations* may be used to produce them when formatting identifiers, cf. Sect. II:5.2.2).

```
ID        ::= id MIX-TOKEN+
MIX-TOKEN ::= TOKEN | PLACE | BRACED-ID | BRACKET-ID | EMPTY-BRS
BRACED-ID ::= braced-id ID
BRACKET-ID::= bracket-id ID
EMPTY-BRS ::= empty-braces | empty-brackets
```

An identifier may also be a *mixfix identifier*[4] ‘$t_0 \_\_ \ldots \_\_ t_n$’, consisting of a sequence of possibly-empty mixfix tokens t_i interspersed with *place-holders*, each place-holder being written as a *pair* of underscores ‘__’. Mixfix identifiers allow the use of *mixfix notation* for application of operations and predicates to argument terms in concrete syntax. A mixfix identifier such as f__ is a different symbol from f. An application of the (unqualified) symbol $f\_\_$ to x may be written as f x, f(x), or f__(x); an application of f to x may only be written as f(x). ‘Invisible’ identifiers, consisting entirely of two or more place-holders (separated by spaces), are allowed.

[4] Mixfix notation is so-called because it generalizes infix, prefix, and postfix notation to allow arbitrary mixing of argument positions and identifier tokens.

Braces ‘`{`’, ‘`}`’ and square brackets ‘`[`’, ‘`]`’ are allowed as mixfix tokens in mixfix identifiers; however, any occurrences of these characters in a declared identifier must be balanced – e.g., ‘`{[__}]`’ and ‘`{__]`’ are *not* allowed.

An identifier may be used simultaneously to identify different kinds of entities (sorts, operations, and predicates) in the same local environment; its intended interpretation is determined by the context.

3

Subsorting Specifications

Section 3.1 introduces the signatures, models, and sentences characterizing basic specifications with subsorts, extending what was provided for many-sorted specifications in Chap. 2. The notion of satisfaction for subsorted specifications is essentially as for many-sorted specifications. The rest of the chapter indicates the abstract and concrete syntax of the constructs of *subsorted* basic specifications, and describes their intended interpretation, extending Chap. 2. Section 3.2 covers the declaration and definition of subsorts, and Sect. 3.3 introduces subsort membership tests and casts for use in axioms.

3.1 Subsorting Concepts

The intuition behind the treatment of subsorts adopted here is to represent subsort inclusion by embedding (which is not required to be the identity), commuting, as usual in order-sorted approaches, with overloaded operation symbols. In the language described later in this chapter, however, no conditions such as 'regularity' are imposed on signatures. Instead, terms and sentences that can be given different parses (up to the commutativity between embedding and overloaded symbols) are simply rejected as ill-formed.

3.1.1 Signatures

A *subsorted signature* $\Sigma = (S, TF, PF, P, \leq)$ consists of a many-sorted signature (S, TF, PF, P) together with a pre-order $\leq$ of *subsort embedding* on the set S of sorts. The pre-order $\leq$ is extended pointwise to sequences of sorts.

For a subsorted signature, we define *overloading relations* for operation and predicate symbols. Two qualified operation symbols f_{w_1,s_1} and f_{w_2,s_2} are in the *overloading relation* (written $f_{w_1,s_1} \sim_F f_{w_2,s_2}$) iff there exists a $w \in S^*$ and $s \in S$ such that $w \leq w_1, w_2$ and $s_1, s_2 \leq s$. Similarly, two qualified predicate symbols p_{w_1} and p_{w_2} are in the overloading relation (written $p_{w_1} \sim_P p_{w_2}$) iff there exists a $w \in S^*$ such that $w \leq w_1, w_2$. We say that two profiles of a symbol are in the overloading relation if the corresponding qualified symbols are in the overloading relation.

Note that two profiles of an overloaded constant declared with different sorts are in the overloading relation iff the two sorts have a common supersort.

A *subsorted signature morphism* $\sigma : \Sigma \to \Sigma'$ is a many-sorted signature morphism that preserves the subsort relation and the overloading relations.

With each subsorted signature $\Sigma = (S, \mathit{TF}, \mathit{PF}, P, \leq)$ a many-sorted signature $\Sigma^{\#}$ is associated, extending $(S, \mathit{TF}, \mathit{PF}, P)$ for each pair of sorts $s \leq s'$ by a total embedding operation (from s into s'), a partial projection operation (from s' onto s), and a membership predicate (testing whether values in s' are embeddings of values in s). The symbols used for embedding, projection, and membership are chosen to be distinct from all symbols that can be explicitly declared in specifications.

Any subsorted signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ expands to a many-sorted signature morphism $\sigma^{\#} : \Sigma_1^{\#} \to \Sigma_2^{\#}$, preserving the symbols used for embedding, projection, and membership.

3.1.2 Models

For a subsorted signature Σ the *subsorted models* are ordinary many-sorted models for $\Sigma^{\#}$ that satisfy the following properties (which can be formalized as a set of conditional axioms):

- Embedding operations are total and injective; projection operations are partial, and injective when defined.
- The embedding of a sort into itself is the identity function.
- All compositions of embedding operations between the same two sorts are equal functions.
- Embedding followed by projection is the identity function; projection followed by embedding is included in the identity function.
- Membership in a subsort holds just when the projection to the subsort is defined.
- Embedding is compatible with those operations and predicates that are in the overloading relations.

3.1.3 Sentences

For a subsorted signature Σ, the *subsorted sentences* are the ordinary many-sorted sentences (as defined in Chap. 2) for the associated many-sorted signature $\Sigma^{\#}$.

A well-formed subsorted basic specification `BASIC-SPEC` of the Casl language determines a basic specification of the underlying subsorted institution, consisting of a subsorted signature and a set of sentences of the form described above. This signature and the class of models over it that satisfy the set of sentences provide the semantics of the basic specification.

3.2 Signature Declarations

No further alternatives for `SIG-ITEMS` are needed.

3.2.1 Sorts

```
SORT-ITEM ::= ... | SUBSORT-DECL | ISO-DECL | SUBSORT-DEFN
```

When a subsort declaration `SUBSORT-DECL`, isomorphism declaration `ISO-DECL`, or subsort definition `SUBSORT-DEFN` occurs in a sort generation construct, the embedding operations between the subsort(s) and the supersort are treated as declared operations with regard to the generation of sorts and to free datatype declarations.

Subsort Declarations

```
SUBSORT-DECL ::= subsort-decl SORT+ SORT
```

A subsort declaration `SUBSORT-DECL` is written:

$$s_1, \ldots, s_n < s$$

It declares all the sorts s_1, …, s_n, and s, as well as the embedding of each s_i as a subsort of s. The s_i must be distinct from s.

Introducing an embedding relation between two sorts may cause operation symbols to become related by the overloading relation, so that values of terms become equated when the terms are identical up to embedding.

Isomorphism Declarations

```
ISO-DECL ::= iso-decl SORT+
```

An isomorphism declaration `ISO-DECL` is written:

$$s_1 = \ldots = s_n$$

It declares all the sorts s_1, …, s_n, as well as their embeddings as subsorts of each other. Thus the carriers for the sorts s_i are required to be isomorphic. The s_i must be distinct.

Subsort Definitions

```
SUBSORT-DEFN ::= subsort-defn SORT VAR SORT FORMULA
```

A subsort definition `SUBSORT-DEFN` is written:

$$s = \{v : s' \bullet F\}$$

The sign displayed as ‘ • ’ may be input as ‘·’ in ISO Latin-1, or as ‘`.`’ in ASCII. It provides an explicit specification of the values of the subsort s of s', in contrast to the implicit specification provided by using subsort declarations and overloaded operation symbols.

The subsort definition declares the sort s; it declares the embedding of s as a subsort of s', which must already be declared in the local environment; and it asserts that the values of s are precisely (the projection of) those values of the variable v from s' for which the formula F holds.

The scope of the variable v is restricted to the formula F. Any other variables occurring in F must be explicitly declared by enclosing quantifications.

Note that the terms of sort s' cannot generally be used as terms of sort s. But they can be explicitly projected to s, using a cast, cf. Sect. 3.3.2.

Defined subsorts may be separately related using subsort (or isomorphism) declarations – implication or equivalence between their defining formulas does *not* give rise to any subsort relationship between them.

3.2.2 Datatypes

Datatype declarations are unchanged, except for a new kind of `ALTERNATIVE`:

Alternatives

```
ALTERNATIVE ::= ... | SUBSORTS
SUBSORTS    ::= subsorts SORT+
```

A subsorts alternative is written:

sorts $s_1, \ldots, s_n$

As with sort declarations, the plural keyword may be written in the singular (regardless of the number of sorts).

The sorts s_i, which must be already declared in the local environment, are declared to be embedded as subsorts of the sort declared by the enclosing datatype declaration. (‘*sorts* $s_1, \ldots, s_n$’ and ‘*sort* s_1 | ... | *sort* s_n’ are equivalent.)

When each alternative of a free datatype declaration is a subsorts alternative, the declared sort corresponds to the disjoint union of the listed sorts, provided that these have no common subsorts. The models of a free datatype declaration, if any, are the same as for a free extension with the datatype declarations, provided that the following conditions are fulfilled (apart from those listed concerning free datatype declarations in Sect. 2.3.4):

- all the sorts that are embedded in the declared sort by the alternatives have no common subsorts; and moreover,
- consider the set of qualified constructor and selector symbols declared by the free datatype: no element of this set is in the overloading relation with any other element, nor with the qualified operation symbols from the local environment.

3.3 Axioms

The only further new constructs introduced in connection with subsorts are atomic formulas for subsort membership, and terms for casting to subsorts.

3.3.1 Atomic Formulas

```
ATOM ::= ... | MEMBERSHIP
```

As for many-sorted specifications, an atomic formula is well-formed (with respect to the current declarations) if it is well-sorted and expands to a unique atomic formula for constructing sentences of the underlying institution – but now for subsorted specifications, uniqueness is required only up to an *equivalence* on atomic formulas and terms. This equivalence is the least one including fully-qualified terms that are the same up to profiles of operation symbols in the overloading relation $\sim_F$ and embedding, and fully-qualified atomic formulas that are the same up to the profiles of predicate symbols in the overloading relation $\sim_P$ and embedding.

The notions of when an atomic formula or term is *well-sorted* and of its *expansion* are indicated below for the various subsorting constructs. Due not only to overloading of predicate and/or operation symbols, but also to implicit embeddings from subsorts into supersorts, a well-sorted atomic formula may have several non-equivalent expansions, preventing it from being well-formed. Qualifications on operation and predicate symbols, or explicit sorts on terms, may be used to determine the intended expansion (up to the equivalence indicated above) and make the enclosing formula well-formed.

Membership

```
MEMBERSHIP ::= membership TERM SORT
```

A membership formula is written:

$T \in s$

The sign displayed as '$\in$' is input as '`in`'.

It is well-sorted if the term T is well-sorted for a supersort s' of the specified sort s. It expands to an application of the pre-declared predicate symbol for testing s' values for membership in the embedding of s.

3.3.2 Terms

```
TERM ::= ... | CAST
```

Casts

```
CAST ::= cast TERM SORT
```

A cast term is written:

T *as* s

It is well-sorted if the term T is well-sorted for a supersort s' of s. It expands to an application of the pre-declared operation symbol for projecting s' to s.

Term formation is also extended by letting a well-sorted term of a subsort s be regarded as a well-sorted term of a supersort s' as well, which provides implicit embedding. It expands to the explicit application of the pre-declared operation symbol for embedding s into s'. (There are no implicit projections.) Also a sorted-term $T : s'$ expands to an explicit application of an embedding, provided that the apparent sort s of the component term T is a subsort of the specified sort s'.

4

Structuring Specifications

Section 4.1 reviews the concepts underlying the structuring constructs provided by Casl. The rest of the chapter indicates their abstract and concrete syntax, and describes their intended interpretation, extending what was provided for basic (many-sorted and subsorted) specifications in the preceding chapters. Section 4.2 covers structured specifications: renaming, hiding, union, extension, and free extension. Section 4.3 introduces named and generic specifications. Section 4.4 indicates how to define and use views, with Sect. 4.5 addressing the use of symbol lists and mappings in connection with views. Finally, Sect. 4.6 introduces compound identifiers.

4.1 Structuring Concepts

Recall that a basic specification, as described in Chaps. 2 and 3, consists essentially of a signature Σ (declaring symbols) and a set of sentences (axioms or constraints) over Σ. The semantics of a well-formed basic specification is the specified signature Σ together with the class of all Σ-models that satisfy the specified sentences.

Section 4.1.1 considers structured specifications, which allow basic specifications to be divided into parts, and the relationship between them to be exhibited. Section 4.1.2 is concerned with named specifications, which allow reuse of such parts; generic specifications have also parameters that can be fitted to argument specifications by so-called views – which can themselves be named and generic. Section 4.1.3 explains how the signature and specification morphisms that are involved in structuring are determined by symbol sets and mappings.

4.1.1 Structured Specifications

A *structured specification* is formed by combining specifications in various ways, starting from basic specifications. For instance, specifications may be *united*; a specification may be *extended* with further signature items and/or sentences; parts of a signature may be *hidden*; the signature may be *translated* to use different symbols (with corresponding translation of the sentences) by a signature morphism; and models may be restricted to *free extensions* (*initial* models are a special case of free extensions). The abstract syntax of constructs in the Casl language for presenting such structured specifications is described later in this chapter.

The structuring concepts and constructs and their semantics do not depend on a specific framework of basic specifications. This means that the Casl language design for basic specifications is orthogonal to that of structured specifications. Therefore, Casl basic specifications, as summarized in the preceding chapters, can be restricted to sublanguages (cf. Sect. 7.1) or extended (cf. Sect. 7.2) in various ways without the need to reconsider or to change structured specifications[1].

The semantics of a well-formed structured specification is of the same form as that of a basic specification: a signature Σ together with a class of Σ-models. Thus the structure of a specification is *not* reflected in its models: it is used only to present the specification in a modular style. (Specification of the *architecture* of models in the CoFI framework is addressed in Chap. 5.)

Within a structured specification, the *current signature* may vary. For instance, when two specifications are united, the signature valid in the one is generally different from that valid in the other. The association between symbols and their declarations as given by the current signature is called the *local environment*.

4.1.2 Named and Generic Specifications

Parts of structured specifications, in contrast to arbitrary parts of basic specifications, are potentially reusable – either verbatim, or with the adjustment of some *parameters*. Specifications may be *named*, so that the reuse of a specification may be replaced by a *reference* to it through its name. (Libraries of named specifications are explained in Chap. 6.) The current association between names and the specifications that they reference is called the *global environment*. Named specifications are implicitly *closed*, not depending on a local environment of declared symbols. A reference to the name of a specification is equivalent to the referenced specification itself, provided that the closedness is explicitly ensured.

A named specification may declare some *parameters*, the union of which is extended by a *body*; it is then called *generic*. A reference to a generic specification should *instantiate* it by providing, for each parameter, an *argument*

[1] The occasional reference to the subsort and overloading relations in this chapter may simply be ignored (or the relations may be replaced by the identity relation) when the framework for basic specifications is restricted so as not to include these features.

specification together with a *fitting morphism* from the parameter to the argument specification. Fitting may also be achieved by (explicit) use of named *views* between the parameter and argument specifications. The union of the arguments, together with the translation of the generic specification by an expansion of the fitting morphism, corresponds to a so-called pushout construction – taking into account any explicit *imports* of the generic specification, which allow symbols used in the body to be declared also by arguments.

4.1.3 Signature and Specification Morphisms

The semantics of structured specifications involve signature morphisms and the corresponding reducts on models. For instance, hiding some symbols in a specification corresponds to a signature morphism that injects the non-hidden symbols into the original signature; the models, after hiding the symbols, are the reducts of the original models along this morphism. Translation goes the other way: the reducts of models over the translated signature back along the morphism give the original models.

The semantics of views involves also *specification morphisms*, which are signature morphisms between particular specifications such that the reduct of each model of the target specification is a model of the source specification.

Given a signature Σ with symbols $|\Sigma|$, *symbol sets* and *symbol mappings* determine signature morphisms as follows:

- A subset of the symbols in $|\Sigma|$ determines the inclusion of the *smallest* subsignature of Σ that contains these symbols. (When an operation or predicate symbol is included, all the sorts in its profile have to be included too.)

  It also determines the inclusion of the *largest* subsignature of Σ that does *not* contain any of these symbols. (When a sort is not included, no operation or predicate symbol with that sort in its profile can be included either.)
- A mapping of symbols in $|\Sigma|$ determines the signature morphism from Σ that extends this mapping with identity maps for all the remaining names in $|\Sigma|$. In the case that the signature morphism does not exist, the enclosing construct is ill-formed.
- Given another signature Σ', a mapping of symbols in $|\Sigma|$ to symbols in $|\Sigma'|$ determines the unique signature morphism from Σ to Σ' that extends the given mapping, and then is the identity, as far as possible, on common names of Σ and Σ'. (Mapping an operation or predicate symbol implies mapping the sorts in the profile consistently.) In the case that the signature morphism does not exist or is not unique, the enclosing construct is ill-formed.

4.2 Structured Specifications

The summary below indicates when structured specifications are well-formed, and how their signatures and classes of models are determined by those of their component specifications. The interpretation is essentially based on model classes – a 'flattening' reduction to sets of sentences is not possible, in general (due to the presence of constructs such as hiding and freeness).

A structured specification can only be well-formed when all its component specifications are well-formed.

```
SPEC ::= BASIC-SPEC | TRANSLATION | REDUCTION
       | UNION | EXTENSION | FREE-SPEC | LOCAL-SPEC
       | CLOSED-SPEC
```

A *translation* allows the symbols declared by a specification to be renamed; it may also be used to require that some symbols have been declared, e.g., when referencing a named specification. A *reduction* allows symbols to be hidden; for convenience, the remaining 'revealed' symbols may be simultaneously renamed. A *union* combines specifications such that when the declaration of a particular symbol is common to some of the combined specifications, its interpretation in a model has to be a common one too – this is called the 'same name, same thing' principle. An *extension* may *enrich* models by declaring new symbols and asserting their properties, and/or *specialize* the interpretation of already-declared symbols. A *free specification* is used to restrict interpretations to *free extensions*, with initiality as a special case. A *local specification* is used to specify *auxiliary* symbols for local use, hiding them afterwards. A *closed specification* ensures that the local environment provided to a specification is empty.

When the above constructs are combined in the same specification, the grouping is determined unambiguously by precedence rules: translations and reductions have the highest precedence, then come local specifications, then unions, and finally extensions have the lowest precedence. (Free specifications generally involve explicit grouping, and their relative precedence to the other constructs is irrelevant.) A different grouping may always be obtained by use of grouping braces: '**{** ... **}**'.

A specification `SPEC` may occur in a context (e.g., when it is being named) where it is required to be *self-contained* or *closed*, not depending on the local environment at all. In that case, it determines a signature and a class of models straightforwardly.

In structured specifications, however, a specification `SPEC` may also occur in a context where it is to *extend* other specifications, providing itself only part of a signature. Then its interpretation determines an extended signature Σ', given a signature Σ (the local environment), together with a model class over Σ' (when defined), given a model class over Σ. The signature and model class for the self-contained case above can be obtained by supplying the empty signature and the model class of the empty specification, respectively.

Translations and reductions in a `SPEC` are not allowed to affect symbols that are already in the local environment that is being extended. The other structuring constructs generalize straightforwardly from self-contained specifications to extensions.

4.2.1 Translations

```
TRANSLATION ::= translation SPEC RENAMING
RENAMING    ::= renaming  SYMB-MAP-ITEMS+
```

A translation is written:

SP **with** *SM*

Symbol mappings *SM* are described in Sect. 4.5.

The symbols mapped by *SM* must be among those declared by *SP*. The signature Σ given by *SP* and the mapping *SM* then determine a signature morphism to a signature Σ', as explained in Sect. 4.1.3. The morphism must not affect the symbols already declared in the local environment, which is passed unchanged to *SP*.

The class of models of the translation consists exactly of those models over Σ' whose reducts along the morphism are models of *SP*.

If a partial operation symbol is renamed into a total one, this is only well-formed in the case that the resulting operation symbol is already total due to another component of the renaming.

When the symbol mapping *SM* is simply a list of identity maps (which may be abbreviated to a simple list of symbols) the only effect of the translation on the semantics of *SP* is to require that the symbols listed are indeed included in the signature given by *SP*, otherwise the translation is not well-formed.

4.2.2 Reductions

```
REDUCTION   ::= reduction SPEC RESTRICTION
RESTRICTION ::= HIDDEN | REVEALED
HIDDEN      ::= hidden SYMB-ITEMS+
REVEALED    ::= revealed SYMB-MAP-ITEMS+
```

A hiding reduction is written:

SP **hide** *SL*

A revealing reduction is written:

SP **reveal** *SM*

Symbol lists *SL* and symbol mappings *SM* are described in Sect. 4.5.

The symbols listed by *SL*, or mapped by *SM*, must be among those declared by *SP*.

In the case of a hiding reduction, the signature Σ given by SP and the set of symbols listed by SL determine the inclusion of the largest subsignature Σ' of Σ that does *not* contain any of the listed symbols, as explained in Sect. 4.1.3. Note that hiding a sort entails hiding all the operations and predicate symbols whose profiles involve that sort.

In the case of a revealing reduction, the signature Σ given by SP and the set of symbols mapped by SM determine the inclusion of the smallest subsignature Σ' of Σ that contains all of the listed symbols, as explained in Sect. 4.1.3. Note that revealing an operation or predicate symbol entails revealing the sorts involved in its profile.

In both cases, the subsort embedding relation is inherited from that declared by SP, and a model class $\mathcal{M}$ is given by the reducts of the models of SP along the inclusion of Σ' in Σ.

In the case of a hiding reduction, its model class is simply $\mathcal{M}$. In the case of a revealing reduction, however, the signature Σ' and the mapping SM of (all) the symbols in it determine a signature morphism to a signature Σ'', as explained in Sect. 4.1.3. The class of models of the reduction then consists exactly of those models over Σ'' whose reducts along this morphism are in $\mathcal{M}$.

A reduction must not affect the symbols already declared in the local environment, which is passed unchanged to SP.

4.2.3 Unions

```
UNION ::= union SPEC+
```

A union is written:

SP_1 **and** ... **and** SP_n

When the current local environment is empty, each SP_i must determine a complete signature Σ_i. The signature of the union is obtained by the ordinary union of the Σ_i (not their disjoint union). Thus all (non-localized) occurrences of a symbol in the SP_i are interpreted uniformly (rather than being regarded as homonyms for potentially different entities). This is the ‘same name, same thing’ principle. If the same name is declared both as a total and as a partial operation with the same profile (in different signatures), the operation becomes total in the union.

The models are those models of the union signature for which the reduct along the signature inclusion morphism from SP_i is a model of SP_i, for each $i = 1, \ldots, n$.

When the current local environment is non-empty, each SP_i must determine an extension from it to a complete signature Σ_i; then the resulting signature is determined as above. Similarly, models of the local environment are extended to models of the SP_i; then the resulting models are determined as above. This provides the required partial functions from signatures to signatures, and from model classes to model classes.

4.2.4 Extensions

```
EXTENSION ::= extension SPEC+
```

An extension is written:

SP_1 **then** ... **then** SP_n

When the current local environment is empty, SP_1 must determine a complete signature Σ_1; otherwise, it must determine an extension from the local environment to a complete signature Σ_1. For $i = 2, \ldots, n$ each SP_i must determine an extension from Σ_{i-1} to a complete signature Σ_i. The signature determined by the entire extension is then Σ_n.

Similarly, SP_1 determines a class of models $\mathcal{M}_1$ over Σ_1. For $i = 2, \ldots, n$ each SP_i determines the class $\mathcal{M}_i$ of those models over Σ_i which satisfy the conditions imposed by SP_i and whose reducts to Σ_{i-1} are in $\mathcal{M}_{i-1}$. The class of models determined by the entire extension is then $\mathcal{M}_n$.

An annotation '`%cons`' after the occurrence of '**then**' that precedes SP_i indicates that the corresponding extension is conservative, i.e., every model in $\mathcal{M}_{i-1}$ is the reduct of some model in $\mathcal{M}_i$. Similarly, an annotation '`%mono`' indicates that the corresponding extension is monomorphic, i.e., every model in $\mathcal{M}_{i-1}$ is the reduct of a model in $\mathcal{M}_i$ which is unique up to isomorphism. An annotation '`%def`' indicates that the corresponding extension is definitional, i.e., every model in $\mathcal{M}_{i-1}$ is the reduct of a unique model in $\mathcal{M}_i$. Finally, an annotation '`%implies`' indicates that the corresponding extension just adds implied properties, i.e., the model classes $\mathcal{M}_{i-1}$ and $\mathcal{M}_i$ are the same (this requires that their signatures are equal, too).

4.2.5 Free Specifications

```
FREE-SPEC ::= free-spec SPEC
```

A free specification `FREE-SPEC` is written:

free **{** SP **}**

Recall that the specification written:

free types DD_1; ... DD_n;

is parsed as a free datatype declaration construct of a basic specification (cf. Sect. 2.3.4), but in fact it usually has the same interpretation as the free structured specification written:

free **{** **types** DD_1; ... DD_n; **}**

This equivalence holds at least in the framework for basic specifications summarized in Chaps. 2 and 3, under some minor restrictions.

When the current local environment is empty, SP must determine a complete signature Σ; otherwise, it must determine an extension from the local environment to a complete signature Σ. In both cases, Σ is the signature determined by the free specification.

When the current local environment is empty, the free specification determines the class of initial models of *SP*; otherwise, it determines the class of models that are free extensions for *SP* of their own reducts to models of the current local environment.

4.2.6 Local Specifications

```
LOCAL-SPEC ::= local-spec SPEC SPEC
```

A local specification `LOCAL-SPEC` is written:

local SP_1 **within** SP_2

It is equivalent to writing:

{ SP_1 **then** SP_2 **} hide** $SY_1, \ldots, SY_n$

where $SY_1, \ldots, SY_n$ are all the symbols declared by SP_1 that are not already in the current local environment. Thus the symbols $SY_1, \ldots, SY_n$ are only for local use in (SP_1 and) SP_2. The hiding must not affect symbols that are declared only in SP_2 (thus operation or predicate symbols declared in SP_2 should not have sorts declared by SP_1 in their profiles).

4.2.7 Closed Specifications

```
CLOSED-SPEC ::= closed-spec SPEC
```

A closed specification `CLOSED-SPEC` is written:

closed { *SP* **}**

It determines the same signature and class of models as *SP* determines in the empty local environment, thus ensuring the closedness of *SP*.

4.3 Named and Generic Specifications

Specifications are named by specification definitions, and referenced by use of the name. A named specification may also have some parameters, which have to be instantiated when referencing the specification.

4.3.1 Specification Definitions

```
SPEC-DEFN   ::= spec-defn SPEC-NAME GENERICITY SPEC
GENERICITY ::= genericity PARAMS IMPORTED
PARAMS      ::= params SPEC*
IMPORTED    ::= imported SPEC*
```

A generic specification definition `SPEC-DEFN` with some parameters and some imports is written:

> **spec** SN $[\mathit{SP}_1]$ ... $[\mathit{SP}_n]$ **given** $\mathit{SP}''_1, \ldots, \mathit{SP}''_m =$
>
> $\quad \mathit{SP}$
>
> **end**

When the list of imports $\mathit{SP}''_1, \ldots, \mathit{SP}''_m$ is empty, the definition is written:

> **spec** SN $[\mathit{SP}_1]$ ... $[\mathit{SP}_n] =$
>
> $\quad \mathit{SP}$
>
> **end**

When the list of parameters $\mathit{SP}_1, \ldots, \mathit{SP}_n$ is empty, the definition merely names a specification and is simply written:

> **spec** $\mathit{SN} =$
>
> $\quad \mathit{SP}$
>
> **end**

The terminating '**end**' keyword is optional.

It defines the name SN to refer to the specification that has parameter specifications $\mathit{SP}_1, \ldots, \mathit{SP}_n$ (if any), import specifications $\mathit{SP}''_1, \ldots, \mathit{SP}''_m$ (if any), and body specification SP. This extends the global environment (which must not already include a definition for SN).

The well-formedness and semantics of a generic specification are essentially as for the imports, extended by the union of the parameter specifications, extended by the body:

$$\{\ \mathit{SP}''_1 \textbf{ and } \ldots \textbf{and } \mathit{SP}''_m\ \} \textbf{ then } \{\ \mathit{SP}_1 \textbf{ and } \ldots \textbf{and } \mathit{SP}_n\ \} \textbf{ then } \mathit{SP}$$

The local environment given to the defined specification is empty, i.e., the above specification is implicitly closed. The difference between declaring parameters and leaving them implicit in an extension is that each parameter has to be provided with a fitting argument specification in all references to the specification name SN. The declared parameters show just which parts of the generic specification are *intended* to vary between different references to it. The imports, in contrast, are fixed, and common to the parameters, body, and arguments.

When a declared parameter happens to be merely a specification name, it always must refer to an *existing* specification definition in the global environment – it does *not* declare a local name for an argument specification.

```
SPEC-NAME ::= SIMPLE-ID
```

A specification name `SPEC-NAME` is normally displayed in a SMALL-CAPS font, and input in mixed upper and lower case.

4.3.2 Specification Instantiation

```
SPEC      ::= ... | SPEC-INST
SPEC-INST ::= spec-inst SPEC-NAME FIT-ARG*
```

An instantiation `SPEC-INST` of a generic specification with some fitting argument specifications is written

$SN[FA_1]\ldots[FA_n]$

When the list of fitting arguments $FA_1, \ldots, FA_n$ is empty, the instantiation is merely a reference to the name of a specification that has no declared parameters at all, and it is simply written '*SN*'. Note that the grouping braces '**{ }**', normally required when writing free (or closed) specifications, may always be omitted around instantiations.

The instantiation refers to the specification named *SN* in the global environment, providing a fitting argument FA_i for each declared parameter (in the same order).

```
FIT-ARG  ::= FIT-SPEC
FIT-SPEC ::= fit-spec SPEC SYMB-MAP-ITEMS*
```

A fitting argument specification `FIT-SPEC` is written:

SP'_i **fit** SM_i

When SM_i is empty, the fitting argument specification is simply written SP'_i. Symbol mappings *SM* are described in Sects. 4.5 and 4.6.

The signature Σ_i given by the parameter specification SP_i, the signature Σ'_i given by the corresponding argument specification, and the symbol mapping SM_i determine a signature morphism from Σ_i to Σ'_i, as explained in Sect. 4.1.3. The fitting argument is well-formed only when the signature morphism is defined, i.e., the fitting argument morphism is well-defined. Note that mapping an operation or predicate symbol generally implies non-identity mapping of the sorts in the profile.

When there is more than one parameter, the separate fitting argument morphisms have to be *compatible*, and their union has to yield a single morphism from the union of the parameters to the union of the arguments. Thus any common parts of declared parameters have to be instantiated in the same way, and it is pointless to declare the same parameter twice in a generic specification. (Generic specifications that require two similar but independent parameters can be expressed by using a translation to distinguish between the symbols in the signatures of the two parameters.)

Each fitting argument FA_i is regarded as an extension of the union of the imports (the current local environment is ignored). The fitting argument morphism has to be identity on all symbols declared by the imports SP''_1, $\ldots$, SP''_m of the generic specification, if there are any. Any symbol declared explicitly in the parameter (and not only in the import) must be mapped to a symbol declared explicitly in the argument specification.

Let SP' be the extension of the imports by the generic parameters and then by the body of the specification named SN:

$$\{\ \mathit{SP}''_1\ \textbf{and}\ \ldots\textbf{and}\ \mathit{SP}''_m\ \}\ \textbf{then}\ \{\ \mathit{SP}_1\ \textbf{and}\ \ldots\textbf{and}\ \mathit{SP}_n\ \}\ \textbf{then}\ \mathit{SP}$$

Let FM be the morphism yielded by the fitting arguments $\mathit{FA}_1, \ldots, \mathit{FA}_n$, extended to a morphism applicable to the signature of SP' as explained in Sects. 4.5 and 4.6 (and written as a list of symbol maps). Then the semantics of the well-formed instantiation $\mathit{SN}[\mathit{FA}_1]\ldots[\mathit{FA}_n]$ is the same as that of the specification:

$$\{\ \mathit{SP}'\ \textbf{with}\ \mathit{FM}\ \}\ \textbf{and}\ \mathit{SP}'_1\ \textbf{and}\ \ldots\textbf{and}\ \mathit{SP}'_n$$

where each SP'_i is the specification of the corresponding fitting argument FA_i. Each model of an argument FA_i (these are models of SP'_i reduced by the signature morphism determined by SM_i) is required to be a model of the corresponding parameter SP_i, otherwise the instantiation is undefined. The instantiation is not well-formed if the result signature is not a pushout of the body and argument signatures: if the translated body

$$\{\ \mathit{SP}'\ \textbf{with}\ \mathit{FM}\ \}$$

and the union of the argument specifications

$$\mathit{SP}'_1\ \textbf{and}\ \ldots\textbf{and}\ \mathit{SP}'_n$$

share any symbols, these symbols have to be translations (along FM) of symbols that share in the extension of the imports by the parameters

$$\{\ \mathit{SP}''_1\ \textbf{and}\ \ldots\textbf{and}\ \mathit{SP}''_m\ \}\ \textbf{then}\ \{\ \mathit{SP}_1\ \textbf{and}\ \ldots\textbf{and}\ \mathit{SP}_n\ \}$$

Here, two sorts share if they are identical, and two function or predicate symbols share if they are in the overloading relation.

4.4 Views

Views between specifications are named by view definitions, and referenced by use of the name. A named view may also have some parameters, which have to be instantiated when referencing the view.

4.4.1 View Definitions

```
VIEW-DEFN ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE SYMB-MAP-ITEMS*
VIEW-TYPE ::= view-type SPEC SPEC
```

A view definition `VIEW-DEFN` with some parameters and some imports is written:

view VN $[SP_1]$ ... $[SP_n]$ **given** SP''_1, ..., SP''_m : SP **to** $SP' =$
    SM
end

A view definition `VIEW-DEFN` with some parameters is written:

view VN $[SP_1]$ ... $[SP_n]$: SP **to** $SP' =$
    SM
end

When the list of parameters is empty, the view definition is simply written:

view VN : SP **to** $SP' =$
    SM
end

The terminating '**end**' keyword is optional.

It declares the view name VN to have the type of specification morphisms from SP to SP', parameter specifications SP_1, ..., SP_n (if any), import specifications SP''_1, ..., SP''_m (if any), and defines it by the symbol mapping SM. Symbol mappings SM are described in Sects. 4.5 and 4.6.

SP gets the empty local environment. The well-formedness conditions for SP' are as if SP' were the body of a generic specification with parameters SP_1, ..., SP_n and import specifications SP''_1, ..., SP''_m. The view definition is well-formed only if the signature morphism determined by the symbol mapping SM, as explained in Sect. 4.1.3, is defined. The view definition extends the global environment (which must not already include a definition for VN).

Parameters in a view definition allow the same view to be instantiated with different fitting arguments, giving compositions of the morphism defined by the view with other fitting morphisms. The source SP of the view is *not* in the scope of the view parameters SP_1, ..., SP_n, and view instantiation affects only the target of the generic view.

It is required that the reduct by the specification morphism of each model of the target

{ SP''_1 **and** ... **and** SP''_m **} then {** SP_1 **and** ... **and** SP_n **} then** SP'

is a model of the source SP; otherwise the semantics is undefined.

```
VIEW-NAME ::= SIMPLE-ID
```

A view name `VIEW-NAME` is normally displayed in a Small-Caps font, and input in mixed upper and lower case.

4.4.2 Fitting Views

```
FIT-ARG  ::= ... | FIT-VIEW
FIT-VIEW ::= fit-view VIEW-NAME
```

A reference to a non-generic fitting argument view `FIT-VIEW` is simply written:

view *VN*

It refers to the current global environment, and is well-formed as an argument for a parameter SP_i only when the global environment includes a view definition for *VN* of type from *SP* to SP', such that the signatures of *SP* and of SP_i are the same. The view definition then provides the fitting morphism from the parameter SP_i to the argument specification given by the target SP' of the view.

If the generic specification being instantiated has imports, the fitting morphism is then the union of the specification morphism given by the view and the identity morphism on the imports. The argument specification is the union of the target of the view and the imports.

Each model of *SP* is required to be a model of SP_i, otherwise the instantiation is undefined.

```
FIT-VIEW ::= ... | fit-view VIEW-NAME FIT-ARG+
```

A fitting argument view `FIT-VIEW` involving the instantiation of a generic view to fitting arguments is written:

view *VN* $[FA_1] \ldots [FA_n]$

It refers to the current global environment, and is well-formed only when the global environment includes a generic view definition for *VN* with parameters that can be instantiated by the indicated fitting arguments FA_1, ..., FA_n to give a view of type from *SP* to SP', such that the signatures of *SP* and of SP_i are the same. As with non-generic views, each model of *SP* is required to be a model of SP_i, otherwise the instantiation is undefined. The instantiation of a generic view with some fitting arguments is not well-formed if the instantiation of the target SP' of the view with the same fitting arguments is not well-formed.

4.5 Symbol Lists and Mappings

Symbol lists are used in hiding reductions. Symbol mappings are used in translations, revealing reductions, fitting arguments, and views.

4.5.1 Symbol Lists

```
SYMB-ITEMS ::= symb-items SYMB-KIND SYMB+
SYMB-KIND  ::= implicit | sorts-kind | ops-kind | preds-kind
SYMB       ::= ID | QUAL-ID
QUAL-ID    ::= qual-id ID TYPE
TYPE       ::= OP-TYPE | PRED-TYPE
```

A list of symbols `SYMB-ITEMS` with implicit kinds `SYMB-KIND` is written simply:

$SY_1, \ldots, SY_n$

Overloaded operation symbols and predicate symbols may be disambiguated by explicit qualification; when SY_i is not qualified, the effect is as if all (sort, operation, or predicate) symbols declared with the name SY_i in the current local environment are listed.

Optionally, the list may be sectioned into sub-lists by inserting the keywords '**sorts**', '**ops**', '**preds**' (or their singular forms), which explicitly indicate that the subsequent symbols are of the corresponding kind:

sorts s_1, …, **ops** f_1, …, **preds** p_1, …

As with signature declarations in basic specifications, there is no restriction on the order of the various sections of the list.

A qualified identifier `QUAL-ID` is written

$I : TY$

where TY is an operation type or a predicate type. When TY is a single sort s, it is interpreted as a constant operation type or unary predicate type, as determined by the latest keyword, or, when there is none, unambiguously by the local environment.

The list determines a set of qualified symbols, obtained from the listed symbols with reference to a given signature; the order in which symbols are listed is not significant (except regarding their position in relation to any specified kinds).

Note that in the symbol list 'I_1**,** … **,**$I_n : TY$' it is *only* the last identifier, I_n, which is qualified; to qualify all the identifiers, the list has to be written thus:

$I_1{:}TY$, …, $I_n{:}TY$

4.5.2 Symbol Mappings

```
SYMB-MAP-ITEMS ::= symb-map-items SYMB-KIND SYMB-OR-MAP+
SYMB-OR-MAP    ::= SYMB | SYMB-MAP
SYMB-MAP       ::= symb-map SYMB SYMB
```

A list of symbol maps `SYMB-MAP-ITEMS` with implicit kinds `SYMB-KIND` is written simply:

$SY_1 \mapsto SY_1', \ldots, SY_n \mapsto SY_n'$

The sign displayed as '$\mapsto$' is input as '`|->`'.

$SY_i \mapsto SY_i'$ denotes the map that takes the symbol SY_i to the symbol SY_i'. The mapped symbols in the list must be distinct. Overloaded operation symbols and predicate symbols may be disambiguated by explicit qualification;

when SY_i is not qualified, the effect is as if all (sort, operation, or predicate) symbols declared with the name SY_i (other than those explicitly mapped as fully qualified symbols) in the current environment are mapped uniformly to SY_i'.

Optionally, the list may be sectioned into sub-lists by inserting the keywords '**sorts**', '**ops**', '**preds**' (or their singular forms), which explicitly indicate that the subsequent symbols are of the corresponding kind:

$$\textbf{sorts } s_1 \mapsto s_1', \ldots, \textbf{ ops } f_1 \mapsto f_1', \ldots, \textbf{ preds } p_1 \mapsto p_1', \ldots$$

As with signature declarations in basic specifications, there is no restriction on the order of the various sections of the list.

An identity map '$SY_i \mapsto SY_i$' may be simply written 'SY_i'. Thus a symbol list may be regarded as a special case of a symbol mapping.

The list determines a set of qualified symbols, obtained from the first components of the listed symbol maps with reference to a given signature, together with a mapping of these symbols to qualified symbols obtained from the second components of the listed symbol maps. The order in which symbol maps are listed is not significant (except regarding their position in relation to any specified kinds).

4.6 Compound Identifiers

```
SORT-ID        ::= ... | COMP-SORT-ID
MIX-TOKEN      ::= ... | COMP-MIX-TOKEN
COMP-SORT-ID   ::= comp-sort-id WORDS ID+
COMP-MIX-TOKEN ::= comp-mix-token ID+
```

This extension of the syntax of identifiers for sorts, operations, and predicates is of relevance to generic specifications. An ordinary *compound identifier* COMP-SORT-ID is written '$I[I_1, \ldots, I_n]$'; a mixfix compound identifier COMP-MIX-TOKEN is written by inserting '$[I_1, \ldots, I_n]$' directly after the last (non-placeholder) mixfix token of the identifier. (Compound 'invisible' identifiers without any tokens are not allowed.) Note that declaration of both compound identifiers and mixfix identifiers as operation symbols in the same local environment may give rise to ambiguity, when they involve overlapping sets of mixfix tokens.

The components I_i may (but need not) themselves identify sorts, operations, or predicates that are specified in the declared parameters of a generic specification.

When such a compound identifier is used to name, e.g., a sort in the body of a generic specification, the translation determined by fitting arguments to parameters applies to the components I_1,. . . ,I_n as well. Thus instantiations with different arguments generally give rise to different compound identifiers for what would otherwise be the same sort, which avoids unintended identifications when the instantiations are united.

For example, a generic specification of sequences of arbitrary elements might use the simple identifier *Elem* for a sort in the parameter, and a compound identifier *Seq*[*Elem*] for the sort of sequences in the body. Fitting various argument sorts to *Elem* in different instantiations then results in distinct sorts of sequences.

Subsort embeddings between component sorts do *not* induce subsort embeddings between the compound sorts: when desired, these have to be declared explicitly. For example, when *Nat* is declared as a subsort of *Int*, we do *not* automatically get *Seq*[*Nat*] embedded as a subsort of *Seq*[*Int*] in signatures containing all these sorts.

Instantiation, however, does preserve subsorts: if in a generic specification we have *Elem* declared as a subsort of *Seq*[*Elem*], where *Elem* is a parameter sort, then in the result of instantiation of *Elem* by *Nat*, one does get *Nat* automatically declared as a subsort of *Seq*[*Nat*]. Compound identifiers must not be identified through the identification of components by the fitting morphism. For example, if the body of a generic specification contains both *List*[*Elem*1] and *List*[*Elem*2], the fitting morphism must not map both *Elem*1 and *Elem*2 to *Nat*, otherwise the instantiation is not a pushout.

Higher-order extensions of CASL are expected to provide a more semantic treatment of parametrized sorts, etc.

5

Architectural Specifications

Section 5.1 explains the main concepts of architectural specifications. The rest of the chapter indicates the abstract and concrete syntax of the constructs of architectural specifications, and describes their intended interpretation, extending what was provided for basic and structured specifications in the preceding chapters: Sect. 5.2 covers architectural specification definitions, Sect. 5.3 unit declarations and definitions, Sect. 5.4 unit specifications, and Sect. 5.5 unit expressions.

5.1 Architectural Concepts

The intention with architectural specifications is primarily to impose structure on models, expressing their *composition* from component units – and thereby also a *decomposition* of the task of developing such models from requirements specifications. This is in contrast to the structured specifications summarized in Chap. 4, where the specified models have no more structure than do those of the basic specifications summarized in Chaps. 2 and 3.

5.1.1 Unit Functions

The component units may all be regarded as *unit functions*: functions without arguments give self-contained units; functions with arguments use such units in constructing further units. Note that a resulting unit may be needed for use as an argument in more than one application.

The specification of a unit function indicates the properties to be assumed of the arguments, and the properties to be guaranteed of the result. Such a specification provides the appropriate interfaces for the development of the function. In Casl, self-contained units are simply models as defined in Chaps. 2 and 3, and their properties are expressed by ordinary (usually: named) specifications.

Thus a unit function maps models of argument specifications to models of a result specification. A specification of such a function can be simply a list of the argument specifications together with the result specification. Thinking of argument and result specifications as *types* of models, a specification of a unit function may be regarded as a *function type*.

An entire *architectural specification* is a collection of unit function specifications, together with a description of how the functions are to be composed to give a resulting unit. A model of an architectural specification is a collection of unit functions with the specified types or definitions, together with the result of composing them as described.

5.1.2 Persistency and Compatibility

The intention is that a unit function should actually make use of its arguments. In particular, it should not re-implement the argument specifications. This is ensured by requiring the unit function to be *persistent*: the reduct of the result to each argument signature yields exactly the given arguments.

As a consequence, the result *signature* has to include each argument *signature* – any desired hiding has to be left to when functions are composed. Moreover, since each *symbol* in the union of the argument signatures has to be implemented the same way in the result as in each argument where it occurs, the arguments must already have the same implementation of all common symbols. In the absence of subsorts, this is sufficient to allow one to unambiguously *amalgamate* arguments into a single model over the union of argument signatures. When subsorts are present, extra conditions to ensure that implicit subsort embeddings can be defined unambiguously in such an amalgamated model may be necessary. Let us call arguments satisfying such a requirement *compatible*.

Hence the interpretation of the specification of a unit function is as all *persistent* functions from *compatible* tuples of models of the argument specifications to models of the result specification. When composing such functions, care must be taken to ensure that arguments are indeed compatible. Notice that if two arguments have the same signature, the arguments must be identical. It is not possible to specify a function that should take two arguments that implement the same signature independently – although one can get the same effect, by renaming one or both of the argument signatures.

5.2 Architectural Specification Definitions

```
ARCH-SPEC-DEFN ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC      ::= BASIC-ARCH-SPEC | ARCH-SPEC-NAME
```

An architectural specification definition `ARCH-SPEC-DEFN` is written:

> **arch spec** *ASN* =
>   *ASP*
> **end**

where the terminating '**end**' keyword is optional.

It defines the name *ASN* to refer to the architectural specification *ASP*, extending the global environment (which must not already include a definition for *ASN*). The local environment given to *ASP* is empty.

```
ARCH-SPEC-NAME ::= SIMPLE-ID
```

An architectural specification name `ARCH-SPEC-NAME` is normally displayed in a Small-Caps font, and input in mixed upper and lower case.

A reference in an architectural specification `ARCH-SPEC` to an architectural specification named *ASN* is simply written as the name itself '*ASN*'. It refers to the current global environment, and is well-formed only when the global environment includes an architectural specification definition for *ASN*. The enclosing definition then merely introduces a synonym for a previously-defined architectural specification.

```
BASIC-ARCH-SPEC ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT
UNIT-DECL-DEFN  ::= UNIT-DECL | UNIT-DEFN
RESULT-UNIT     ::= result-unit UNIT-EXPRESSION
```

A basic architectural specification `BASIC-ARCH-SPEC` is written:

> **units** UD_1; ... UD_n; **result** *UE*;

where both the last two semicolons are optional.

It consists of a list of unit declarations and definitions UD_1, ..., UD_n, together with a unit expression *UE* describing how such units are to be composed. A model of such an architectural specification consists of a unit for each UD_i, and the composition of these units as described by *UE*.

5.3 Unit Declarations and Definitions

The visibility of unit names in architectural specifications is linear: each name has to be declared or defined before it is used in a unit expression; and no unit name may be introduced more than once in a particular architectural specification. Note that declarations and definitions of units do not affect the global environment: a unit may be referenced only within the architectural specification in which it occurs.

5.3.1 Unit Declarations

```
UNIT-DECL     ::= unit-decl UNIT-NAME UNIT-SPEC UNIT-IMPORTED
UNIT-IMPORTED ::= unit-imported UNIT-TERM*
UNIT-NAME     ::= SIMPLE-ID
```

A unit declaration UNIT-DECL is written:

UN : USP **given** UT_1,...,UT_n

When the list UNIT-TERM* of imported unit terms is empty, it is simply written:

UN : USP

It provides not only a unit specification USP but also a unit name UN, which is used for referring to the unit in subsequent unit expressions, so that the same unit may be used more than once in a composition.

The UNIT-IMPORTED lists any units UT_1, ...,UT_n that are imported for the implementation of the declared unit (which corresponds to implementing a generic unit function and applying it only once, to the imported units, the argument type of the generic function being merely the list of the signatures of the UT_i). The unit specification USP is treated as an extension of the signatures of the imported units, thus being given a non-empty local environment, in general.

5.3.2 Unit Definitions

```
UNIT-DEFN ::= unit-defn UNIT-NAME UNIT-EXPRESSION
```

A unit definition UNIT-DEFN is written:

$UN = UE$

It defines the name UN to refer to the unit resulting from the composition described by the unit expression UE.

5.4 Unit Specifications

```
UNIT-SPEC-DEFN   ::= unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC        ::= UNIT-TYPE | SPEC-NAME | ARCH-UNIT-SPEC
                   | CLOSED-UNIT-SPEC
```

A unit specification definition UNIT-SPEC-DEFN is written:

unit spec SN =
 USP
end

where the terminating '**end**' keyword is optional.

It provides a name *SN* for a unit specification *USP*. The unit specification may be a unit type. It may also be the name of another unit specification (in the context-free concrete syntax, this is indistinguishable from a reference to a named structured specification in a constant unit type, but the global environment determines how the name should be interpreted). It may be an architectural specification (either a reference to the defined name of an architectural specification, or an anonymous architectural specification). Finally, it may be an explicitly-closed unit specification.

It defines the name *SN* to refer to the unit specification *USP*, extending the global environment (which must not already include a definition for *SN*). The local environment given to *USP* is empty, i.e., the unit specification is implicitly closed.

5.4.1 Unit Types

```
UNIT-TYPE ::= unit-type SPEC* SPEC
```

A unit type is written:

$SP_1 \times \ldots \times SP_n \rightarrow SP$

When the list `SPEC*` of argument specifications is empty, the unit type is simply written '*SP*'. The sign displayed as '×' may be input as '×' in ISO Latin-1, or as '`*`' in ASCII. The sign displayed as '→' is input as '`->`'.

A unit satisfies a unit type when it is a persistent function that maps compatible tuples of models of the argument specifications SP_1, ..., SP_n to models of their extension by the result specification *SP*.

5.4.2 Architectural Unit Specifications

```
ARCH-UNIT-SPEC   ::= arch-unit-spec ARCH-SPEC
```

An architectural unit specification `ARCH-UNIT-SPEC` is written:

arch spec *ASP*

A unit satisfies '**arch spec** *ASP*' when it is the result unit of some model of *ASP*. Given a (basic or named) architectural specification *ASP*, note the difference between '*ASP*' and '**arch spec** *ASP*': the former is an architectural specification, while the latter is a coercion of the architectural specification *ASP* to a unit specification.

5.4.3 Closed Unit Specifications

```
CLOSED-UNIT-SPEC ::= closed-unit-spec UNIT-SPEC
```

A closed unit specification `CLOSED-UNIT-SPEC` is written:

closed *USP*

It determines the same type as *USP* determines in the empty local environment, thus ensuring the closedness of *USP*.

5.5 Unit Expressions

```
UNIT-EXPRESSION ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING    ::= unit-binding UNIT-NAME UNIT-SPEC
```

A unit expression with some unit bindings is written:

$\lambda UN_1 : USP_1; \ldots; \ UN_n : USP_n \ \bullet \ UT$

The sign displayed as 'λ' is input as '`lambda`'. The sign displayed as ' • ' may be input as '·' in ISO Latin-1, or as '`.`' in ASCII. When the list of unit bindings is empty, just the unit term '*UT*' is written.

It describes a composition of units declared (or defined) in the enclosing architectural specification. The result unit is a function, mapping the arguments specified by the unit bindings (if any) to the unit described by the unit term *UT*. The unit names $UN_1, \ldots, UN_n$ for the arguments must be distinct, and not include the names of units previously declared in the enclosing architectural specification.

The unit bindings for the arguments (which are like unit declarations but with no possibility of importing other units) in a unit expression are for (non-parametrized) units that are required to build the result, but are not directly provided yet. This allows for compositions which express partial architectural specifications that depend on additional units, and might be used to instantiate the same composition for various realizations of the required units.

5.5.1 Unit Terms

```
UNIT-TERM ::= UNIT-REDUCTION | UNIT-TRANSLATION | AMALGAMATION
            | LOCAL-UNIT | UNIT-APPL
```

Unit terms provide counterparts to most of the constructs of structured specifications: translations, reductions, amalgamations (corresponding to unions), local unit definitions, and applications (corresponding to instantiations).

Unit terms use the same notation as structured specifications – but with a crucially different semantics, however. This is easiest to notice when considering the difference between union and amalgamation, as well as between translation and unit translation. For units, enough sharing is required so that the constructs, as applied to the given units, will always make sense and produce results. This is in contrast with the constructs for structured specifications, where well-formed unions or (non-injective) translations of consistent specifications might result in inconsistencies.

The sharing between symbols is understood here semantically: two symbols share if they coincide semantically. However, there is also a static semantics (with the corresponding static analysis supported by Casl tools) that exploits situations where symbols required to share in fact *originate* from the same symbol in some unit declaration or definition. Such direct information should be sufficient to discharge the verification conditions implicit in the above semantic requirement in most typical cases. This is simplest when no subsorting constructs are involved. The presence of subsorts, and the properties that subsort embeddings and overloaded operations and predicates must satisfy, make the static analysis more complex [62] (but still tractable in practical examples).

Taking the unit type of each unit name from its declaration, the unit term must be well-typed. All the constructs involved must get argument units over the appropriate signatures.

Unit Translations

```
UNIT-TRANSLATION ::= unit-translation UNIT-TERM RENAMING
```

A unit translation is written:

UT R

where the renaming *R* is written '**with** *SM*', and determines a mapping of symbols, cf. Sect. 4.2.1.

It allows some of the unit symbols to be renamed. Any symbols that happen to be glued together by the renaming must share.

Unit Reductions

```
UNIT-REDUCTION ::= unit-reduction UNIT-TERM RESTRICTION
```

A unit reduction is written:

UT R

where the restriction *R* is written '**hide** *SL*' or '**reveal** *SM*', and determines a set of symbols, and in the latter case also a mapping of them, cf. Sect. 4.2.2.

It allows parts of the unit to be hidden and other parts to be simultaneously renamed.

Amalgamations

```
AMALGAMATION ::= amalgamation UNIT-TERM+
```

An amalgamation is written:

UT_1 **and** ... **and** UT_n

It produces a unit that consists of the components of all the amalgamated units put together. Compatibility of the unit terms must be ensured.

Local Units

```
LOCAL-UNIT ::= local-unit UNIT-DEFN+ UNIT-TERM
```

A local unit is written:

local $UD_1; \ldots; UD_n;$ **within** UT

where the final '`;`' may be omitted.

This allows for naming units that are locally defined for use in a unit term, these units being intermediate results that are not to be visible in the models of the enclosing architectural specification.

Unit Applications

```
UNIT-APPL ::= unit-appl UNIT-NAME FIT-ARG-UNIT*
```

A unit application `UNIT-APPL` is written:

$UN[FAU_1] \ldots [FAU_n]$

It refers to a generic unit named UN that has already been declared or defined in the enclosing architectural specification, providing a fitting argument FAU_i for each declared parameter (in the same order).

```
FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*
```

A fitting argument FAU_i is written:

UT'_i **fit** SM_i

When the symbol mapping SM_i is empty, just the unit term UT'_i is written.

The fitting argument fits the argument unit given by the unit term UT'_i to the corresponding formal argument for the generic unit, via a signature morphism which is determined by the symbol mapping SM_i in the same way as for generic specifications. The result of such 'fitting' (which is the reduct of the argument unit by the signature morphism) must be a model of the corresponding parameter specification in the declaration of the unit UN, otherwise the unit application is undefined.

When there is more than one parameter, the separate fitting argument morphisms have to be compatible; moreover, the argument units determined by UT'_i must be compatible as well.

Then, the unit function denoted by UN is applied to the fitted argument units. The result is translated by the fitting signature morphisms extended to the signature of the result specification in the declaration of UN (just as for instantiations of generic specifications) and finally amalgamated with the argument units, yielding the overall result of the unit application.

6

Specification Libraries

Section 6.1 introduces the concepts underlying the specification libraries provided by CASL. The rest of the chapter indicates the abstract and concrete syntax of the library constructs, and describes their intended interpretation, extending what was provided for basic, structured, and architectural specifications in the preceding chapters. Section 6.2 presents the constructs of *local* libraries. Such libraries are not dependent on other libraries. Section 6.3 considers constructs for referencing *distributed* libraries. Finally, Sect. 6.4 explains the form and intended interpretation of library names.

6.1 Library Concepts

Specifications may be *named* by *definitions* and collected in *libraries*. In the context of a library, the (re)use of a specification may be replaced by a *reference* to it through its name. The current association between names and the specifications that they reference is called the *global environment*; it may vary throughout a library: with *linear visibility*, as in CASL, the global environment for a named specification is determined exclusively by the definitions that precede it. When overriding is forbidden, as in CASL, each valid reference to a particular name refers to the same defined entity.

The local environment given to each named specification in a library should be independent of the other specifications in the library (in CASL, it is empty). Thus any dependence between the specifications is always apparent from the explicit references to the names of specifications.

A library may be located at a particular *site* on the Internet. The library is referenced from other sites by a name which determines the location and perhaps identifies a particular version of the library. To allow libraries to be relocated without this invalidating existing references to them, library names may be interpreted relative to a *global directory* that maps names to URLs. Libraries may also be referenced directly by their (relative or absolute) URLs, independently of their registration in the global directory.

A library may incorporate the *downloading* of (the semantics of) named specifications from (perhaps particular versions of) other libraries, whenever the library is used. To ensure continuous access to specifications despite temporary failures at a particular library site, registered libraries may be mirrored at archive sites.

The semantics of a specification library is the name of the library together with a map taking each specification name defined in it to the semantics of that specification. The initial global environment for the library is empty.

6.2 Local Libraries

```
LIB-DEFN ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM ::= SPEC-DEFN | VIEW-DEFN | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
```

A library definition LIB-DEFN is written:

library LN $LI_1 \ldots LI_n$

Each library item LI_i starts with a distinctive keyword, and may be terminated by an optional '**end**'.

The library definition provides a collection of specification (and perhaps also view) definitions. It is well-formed only when the defined names are distinct, and not referenced until (strictly) after their definitions. The global environment for each definition is that determined by the preceding definitions. Thus a library in Casl provides linear visibility, and mutual or cyclic chains of references are not allowed.

The local environment for each definition is empty: the symbols declared by the preceding specifications in the library are only made available by explicit reference to the name of the specification concerned.

Each specification definition in a library must be self-contained (after resolving references to names defined in the current global environment), determining a complete signature – fragments of specifications cannot be named.

A library definition determines a library name, together with a map from names to the semantics of the named specifications.

6.3 Distributed Libraries

```
LIB-ITEM         ::= ... | DOWNLOAD-ITEMS
DOWNLOAD-ITEMS   ::= download-items LIB-NAME ITEM-NAME-OR-MAP+
ITEM-NAME-OR-MAP ::= ITEM-NAME | ITEM-NAME-MAP
ITEM-NAME-MAP    ::= item-name-map ITEM-NAME ITEM-NAME
ITEM-NAME        ::= SIMPLE-ID
```

The syntax of local libraries is here extended with a further sort of library item, for use with distributed libraries.

A downloading DOWNLOAD-ITEMS is written:

from LN **get** $IN_1 \mapsto IN'_1, \ldots, IN_n \mapsto IN'_n$ **end**

where the terminating '**end**' keyword is optional. An identity map '$IN_i \mapsto IN_i$' may be simply written 'IN_i'.

The downloading specifies which definitions to download from the remote library named LN, changing the remote names IN_i to the local names IN'_i. The semantics corresponds to having already replaced all references in the downloaded definitions by the corresponding (closed) specifications; cyclic chains of references via remote libraries are not allowed. The download items show exactly which specification names are added to the current global environment of the library in which they occur, allowing references to named specifications to be checked locally (although not whether the kind of specification to be downloaded from the remote library is consistent with its local use).

6.4 Library Names

```
LIB-NAME       ::= LIB-ID | LIB-VERSION
LIB-VERSION    ::= lib-version LIB-ID VERSION-NUMBER
VERSION-NUMBER ::= version-number NUMBER+
```

A library name LIB-NAME without a VERSION-NUMBER is written simply as a library identifier LI. A library name LIB-NAME with version numbers $N_1, \ldots, N_n$ is written:

LI **version** $N_1. \ldots .N_n$

The lists of version numbers are ordered lexicographically on the basis of the usual ordering between natural numbers.

The library name of a library definition determines how the library is to be referenced from other libraries; its interpretation as a URL determines the primary location of the library (any copies of a library are to retain the original name).

When the name of a defined library is simply a library identifier LIB-ID, it must be changed to an explicit library version LIB-VERSION before defining further versions of that library. A library identifier without an explicit version in a downloading construct always refers to the current version of the identified library: the one with the largest list of version numbers (which is not necessarily the last-created version).

```
LIB-ID        ::= DIRECT-LINK | INDIRECT-LINK
DIRECT-LINK   ::= direct-link URL
INDIRECT-LINK ::= indirect-link PATH
```

A direct link to a library is simply written as the URL of the library. The location of a library is always a directory, giving access not only to the individual specifications defined by the current version of the library but also to previously-defined versions, various indexes, and perhaps other documentation.

An indirect link is written:

$$FI_1/\ldots/FI_n$$

where each file identifier FI_i is a valid file name, as for use in a path in a URL. An indirect link is interpreted as a URL by the current global library directory.

7

Sublanguages and Extensions

From CASL, simpler languages (e.g., for interfacing with existing tools) are obtained by restriction, and CASL is also extended in more advanced languages (e.g., higher-order). CASL strikes a balance between simplicity and expressiveness.

7.1 Sublanguages

This section defines various frequently used sublogics as sublanguages of CASL. Different existing algebraic specification language implementations have a natural extension in CASL, so that specifications in such languages can be translated into CASL and can be combined with other CASL specifications. Tool support for CASL specifications can be obtained for specifications within given sublanguages by translating CASL specifications for those sublanguages into the corresponding languages of given tools.

Note that the sublanguages defined here only address basic specifications. CASL structured and architectural specifications as well as libraries remain the same for all the sublanguages.

7.1.1 A Language for Naming Sublanguages

A concise notation for a variety of sublanguages of CASL can be obtained by assigning tokens to the various features of CASL. This leads to a two-component name for the various sublanguages that can be obtained by combining CASL's features. The first component is a vector of tokens. The presence (or absence) of a token denotes the presence (or absence) of a corresponding feature, cf. Sect. 7.1.2. The second component determines the level of expressiveness of axioms due to Sect. 7.1.3.

We assign the following tokens to features:

- *Sub* stands for subsorting.
- *P* stands for partiality.
- *C* stands for sort generation constraints.
- An equality symbol (=) stands for equality.

Any subset of this set of four tokens, when combined with the notation for denoting a particular level of expressiveness, denotes the sublanguage obtained by equipping the level of expressiveness with the features expressed by the tokens (technically, this is achieved by intersecting all those sublanguages which omit a feature that is not in the set).

In order to be consistent with standard terminology, the predicate feature is combined with the levels of axiomatic expressiveness (Sect. 7.1.3), as follows.

With predicates, we have:

- *FOL* stands for the unrestricted form of axioms (first-order logic).
- *GHorn* stands for the restriction to generalized positive conditional logic.
- *Horn* stands for the restriction to positive conditional logic.
- *Atom* stands for the restriction to atomic logic.

Without predicates, we have:

- *FOAlg* stands for the unrestricted form of axioms (first-order logic).
- *GCond* stands for the restriction to generalized positive conditional logic.
- *Cond* stands for the restriction to positive conditional logic.
- *Eq* stands for the restriction to atomic logic.

Finally, we adopt the convention that the equality sign = is always put at the end, as a superscript.

Some Interesting Sublanguages of Casl

Here are some examples of what the above naming scheme means in practice:

SubPCFOL$^=$**:**
(read: subsorted partial constraint first-order logic with equality). This is the logic of Casl itself.

SubPFOL$^=$**:**
(read: subsorted partial first-order logic with equality). Casl without sort generation constraints. This is described in [13].

FOL$^=$**:**
Standard many-sorted first-order logic with equality.

PFOL$^=$**:**
Partial many-sorted first-order logic with equality.

FOAlg$^=$**:**
First-order algebra (i.e., no predicates).

***SubPHorn*$^=$:**

This is the positive conditional fragment of Casl. It has two important properties:

- Initial models and free extensions exist (see [41]).
- Using a suitable encoding of subsorting and partiality, one can use conditional term rewriting or paramodulation [51] for theorem proving.

***SubPCHorn*$^=$:**

The positive conditional fragment plus sort generation constraints. Compared with *SubPHorn*$^=$, one has to add induction techniques to the theorem proving tools.

***PCond*$^=$:**

These are Burmeister's partial quasi-varieties [10] modulo the fact that Burmeister does not have total function symbols. But total function symbols can be easily simulated by partial ones, using totality axioms, as in the partly total algebras of [11]. A suitable restriction leads to Reichel's HEP-theories [55]. Meseguer's Rewriting Logic [35] can be embedded into *PCond*$^=$.

***Horn*$^=$:**

This is Eqlog [22, 51]. By further restricting this we get Membership Equational Logic [36], Equational Type Logic [33] and Unified Algebras [48]. Of course, Membership Equational Logic, Equational Type Logic and Unified Algebras are not just restrictions of *Horn*$^=$, but all have been invented in order to represent more complex logics within a subset of *Horn*$^=$.

***Horn*:**

Logic Programming (Pure Prolog) [32].

***SubCond*$^=$:**

Subsorted conditional logic. This is similar but not equal to OBJ3 [25], see [41], the main difference being the treatment of subsorts.

***Cond*$^=$:**

This is many-sorted conditional equational logic [66] .

***SubPAtom*:**

The atomic subset of Casl. Unconditional term rewriting becomes applicable.

***SubPCAtom*:**

The atomic subset plus sort generation constraints.

***Eq*$^=$:**

This is classical equational logic [24].

***CEq*$^=$:**

Equational logic plus sort generation constraints.

In the literature, some of the above institutions are typically defined in a way allowing empty carrier sets, while Casl excludes empty carriers. This problem is discussed in [41].

7.1.2 A List of Orthogonal Features

In this section, we describe a number of CASL's features *negatively* by specifying, for each feature, the sublanguage of CASL that leaves out exactly that feature. This is possible since CASL is already the combination of all its features. A combination of only some of CASL's features can then be obtained by intersecting all those sublanguages that exclude exactly one of the undesired features. Intersection of languages as institutions is formally defined in [41].

Partiality

As indicated above, we now specify the sublanguage of CASL *without* partiality. We cross out those parts of the CASL grammar given in Sect.. II:2.1 that have to be removed in order to remove the possibility to declare and use partial functions:

```
OP-TYPE           ::= TOTAL-OP-TYPE ~~| PARTIAL-OP-TYPE~~

OP-HEAD           ::= TOTAL-OP-HEAD ~~| PARTIAL-OP-HEAD~~

ALTERNATIVE       ::= TOTAL-CONSTRUCT ~~| PARTIAL-CONSTRUCT~~

COMPONENTS        ::= TOTAL-SELECT ~~| PARTIAL-SELECT~~ | SORT

TERM              ::= SIMPLE-ID | QUAL-VAR | APPLICATION
                    | SORTED-TERM | CONDITIONAL ~~| CAST~~
```

Note that we can keep DEFINEDNESS, EXISTL-EQUATION and STRONG-EQUATION, since in the total case, the former is semantically equivalent to true and the latter two are equivalent.

Predicates

This is easy as well: from the CASL grammar we just have to remove the possibility to declare and use predicates:

```
SIG-ITEMS         ::= SORT-ITEMS | OP-ITEMS ~~| PRED-ITEMS~~
                    | DATATYPE-ITEMS

ATOM              ::= TRUTH ~~| PREDICATION~~ | DEFINEDNESS
                    | EXISTL-EQUATION | STRONG-EQUATION
```

Subsorting

Just remove everything from Sect. II:2.1.2 from the grammar.

Sort Generation Constraints

To remove sort generation constraints, just change the grammar as follows:

```
BASIC-ITEMS      ::= SIG-ITEMS ~~| FREE-DATATYPE | SORT-GEN~~
                   | VAR-ITEMS | LOCAL-VAR-AXIOMS | AXIOM-ITEMS
```

Equality

To remove equations, just change the grammar as follows:

```
ATOM             ::= TRUTH | PREDICATION | DEFINEDNESS
                   ~~|  EXISTL-EQUATION | STRONG-EQUATION~~
```

7.1.3 A List of Levels of Expressiveness

In this and the following section, the sublanguages are identified in a purely syntactical way, namely by restricting the grammar for the Casl abstract syntax (cf. Sect. II:2.1). Thus, given a particular specification, a tool can easily determine the minimal sublanguage of Casl to which the specification belongs.

We start with the four different level of axiomatic expressiveness.

First-Order Logic

This is given by the unrestricted Casl grammar.

Positive Conditional Logic

Positive conditional logic more precisely means: universally-quantified positive conditional logic. Usually this means that formulas are restricted to universally-quantified implications that consist of a premise that is a conjunction of atoms, and a conclusion that is an atom:

$$\forall x_1{:}s_1 \ldots \forall x_n{:}s_n \bullet \varphi_1 \wedge \ldots \wedge \varphi_n \Rightarrow \varphi$$

Positive conditional means that the atoms must not implicitly contain negative parts. Now strong equations are implicit implications (*if* both sides are defined, *then* they are equal, or both sides are undefined) and thus may not occur in the premises of positive conditional axioms. The reason for this is that we want to have initial models for positive conditional axioms, and these do not exist if strong equations are allowed in the premises (see [12, 3]).

The new grammar for formulas, which describes a subset of Casl (even though it uses some new nonterminals, written *THUS*), is as follows:

FORMULA ::= QUANTIFICATION ~~| CONJUNCTION | DISJUNCTION~~
| IMPLICATION ~~| EQUIVALENCE | NEGATION | ATOM~~
QUANTIFICATION ::= quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER ::= universal ~~| existential | unique-existential~~
P-CONJUNCTION ::= conjunction *P-ATOM+*
IMPLICATION ::= implication *P-CONJUNCTION ATOM*

P-ATOM ::= TRUTH | PREDICATION | DEFINEDNESS
| EXISTL-EQUATION ~~| STRONG-EQUATION~~
ATOM ::= TRUTH | PREDICATION | DEFINEDNESS
| EXISTL-EQUATION | STRONG-EQUATION
TRUTH ::= true ~~| false~~

SUBSORT-DEFN ::= subsort-defn SORT VAR SORT *P-ATOM*

P-CONJUNCTION allows a conjunction in the premise of an implication. Note that *P-ATOM* is needed to forbid strong equations in the premise.

Since subsort definitions are reduced to equivalences (and can be further reduced to two implications), in order to obtain a positive conditional formula, we must ensure that the defining formula is a *P-ATOM*.

Generalized Positive Conditional Logic

In the following, we generalize the above form of positive conditional formulas by allowing also:

- conjunctions of atoms in the conclusion (they can be removed by writing, for each conjunct, an implication with the same premise and the conjunct as conclusion),
- nested conjunctions in the premise and conclusion (they can be flattened),
- equivalences in addition to implications (an equivalence is equivalent to two implications), and
- nesting of conjunction and universal quantification (by the rules for prenex normal form [64], we can always shift the quantifiers inside, getting a conjunction of universally quantified implications).

Each formula of this more general kind is equivalent to a set of formulas of the standard conditional kind. Thus there is an easy transformation from generalized positive conditional logic to plain positive conditional logic.

The new grammar for formulas is as follows

FORMULA ::= QUANTIFICATION | *C-CONJUNCTION*
| *F-CONJUNCTION* ~~| DISJUNCTION~~
| IMPLICATION | EQUIVALENCE ~~| NEGATION~~ | ATOM
QUANTIFICATION ::= quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER ::= universal ~~| existential | unique-existential~~
F-CONJUNCTION ::= conjunction FORMULA+
P-CONJUNCTION ::= conjunction *PREMISE+*

C-CONJUNCTION ::= conjunction *CONCLUSION+*
PREMISE ::= *P-CONJUNCTION* | *P-ATOM*
CONCLUSION ::= *C-CONJUNCTION* | *ATOM*
IMPLICATION ::= implication *PREMISE CONCLUSION*
EQUIVALENCE ::= equivalence *PREMISE PREMISE*

P-ATOM ::= TRUTH | PREDICATION | DEFINEDNESS
| EXISTL-EQUATION ~~| STRONG-EQUATION~~
ATOM ::= TRUTH | PREDICATION | DEFINEDNESS
| EXISTL-EQUATION | STRONG-EQUATION
TRUTH ::= true ~~| false~~

SUBSORT-DEFN ::= subsort-defn SORT VAR SORT *PREMISE*

P-CONJUNCTION and *C-CONJUNCTION* allow nested conjunctions in the premises and conclusion of an implication. *F-CONJUNCTION* allows nesting of quantification and conjunction.

Atomic Logic

This is the restriction of conditional logic to unconditional (i.e., universally quantified atomic) formulas. Strong equations are only allowed if at least one of the sides of the equation consists entirely of total function symbols and variables; this is indicated by the nonterminal written '(STRONG-EQUATION)' below. Other strong equations are removed due to their conditional nature: in [37] it is proved that strong equations can simulate positive conditional formulas. Since definitions of partial functions involve strong equations with possibly partial function symbols occurring on both sides of the equations, these are removed as well. Likewise, associativity and commutativity attributes are removed. Finally, due to the conditional nature of subsort definitions we have to forbid them entirely.

SORT-ITEM ::= SORT-DECL | SUBSORT-DECL | ISO-DECL ~~| SUBSORT-DEFN~~

OP-HEAD ::= TOTAL-OP-HEAD ~~| PARTIAL-OP-HEAD~~

BINARY-OP-ATTR ::= ~~assoc-op-attr~~ | ~~|comm-op-attr~~ | idem-op-attr
FORMULA ::= QUANTIFICATION | *F-CONJUNCTION* ~~| DISJUNCTION~~
~~| IMPLICATION | EQUIVALENCE | NEGATION~~ | *P-ATOM*
QUANTIFICATION ::= quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER ::= universal ~~| existential | unique-existential~~
F-CONJUNCTION ::= conjunction FORMULA+

P-ATOM ::= TRUTH | PREDICATION | DEFINEDNESS
| EXISTL-EQUATION | (STRONG-EQUATION)
TRUTH ::= true ~~| false~~

7.2 Extensions

Various extensions of Casl have been proposed. They have not been developed by CoFI as a whole, but by subgroups of CoFI, and have not yet been approved by CoFI and IFIP WG1.3. For a language to be approved as an extension of Casl, its syntax and intended semantics have to be documented in relation to the Casl Summary (i.e., the foregoing chapters of this part of the reference manual), and it has to include most constructs of Casl– respecting their usual syntax and semantics.

Most of the extensions are defined at the level of Casl basic specifications. The exceptions are CoCasl, HetCasl and the refinement language: these languages also define new structuring constructs.

7.2.1 Higher-Order and Coalgebraic Extensions

HasCasl

HasCasl [60, 61] is an extension of Casl that establishes a connection with functional programming languages such as Haskell. To this end, Casl has been extended by features that support the type system of these languages, in particular higher-order types, type constructors, and parametric polymorphism. The HasCasl semantics is tuned to allow program development by specification refinement, while at the same time staying close to the set-theoretic semantics of first-order Casl. The number of primitive concepts in the logic has been kept as small as possible; various extensions to the logic can be formulated within the language itself. Together with the HasCasl tool support, an environment is created for the specification and formal implementation of software, which allows the coherent development of formal specifications and executable functional programs in a common framework.

CoCasl

CoCasl [46] is a simple coalgebraic extension of Casl. CoCasl admits the nested combination of algebraic datatypes and coalgebraic process types. CoCasl dualizes Casl's generated and free types to cogenerated and cofree types, and provides a coalgebraic modal logic for these. At the level of structured specifications, Casl's free construct is dualized to a cofree construct.

7.2.2 Reactive Extensions

Casl-Ltl

Casl-Ltl is an extension of Casl that allows for specification of dynamic systems by modelling them by means of labelled transition systems and by expressing their properties with temporal formulas. It is based on LTL, a logic-algebraic formalism for the specification of concurrent systems. A detailed summary of the syntax and intended semantics of Casl-Ltl can be found in [54].

Sb-Casl

Sb-Casl [4] is an extension of Casl that deals with the specification of state-based systems using the state-as-algebra approach. In this approach, the state of a software system is modeled as an algebra (in Sb-Casl, as an algebra of the Casl institution), and operations changing the state as (partial) functions on classes of algebras.

Sb-Casl incorporates ideas from Gurevich's Abstract State Machines (ASM), d-oids by Astesiano and Zucca, and others. In particular, this extension combines an operational style of specification (in the sense of ASMs) with a declarative style (in the sense of Z).

Csp-Casl

Csp-Casl [56] is a combination of Casl with the process algebra CSP, following the paradigm "integrating a formalism for concurrent aspects with algebraic specification of static datatypes" [2]. Its novel aspects include the use of denotational semantics for the process part, loose semantics for the datatypes, and their combination in terms of a two-step semantics leading to decomposition theorems concerning an appropriate notion of refinement.

7.2.3 Extensions at the Structured Level

HetCasl

HetCasl stand for heterogeneous Casl, and allows for mixing of specifications written in different logics (using translations between the logics) [40, 42]. It extends Casl only at the level of structuring constructs, by adding constructs for choosing the logic and translating specifications among logics. HetCasl is needed when combining specifications written in Casl with specifications written in its sublanguages and extensions.

A Simple Refinement Language

A simple refinement language built on top of Casl has been proposed in [43, 47]. It allows to refine unit specifications and architectural specifications, until monomorphic unit specifications are reached. Under suitable restrictions, the latter can then be translated into a programming language.

Part II

CASL Syntax

The CoFI Language Design Group

Editors: Bernd Krieg-Brückner and Peter D. Mosses

P.D. Mosses (Ed.): CASL Reference Manual, LNCS 2960, pp. 71–112, 2004.

1

Introduction

This part of the Casl Reference Manual is concerned with syntax. It makes the usual distinction between concrete syntax and abstract syntax: the former deals with the representation of specifications as sequences of characters, and with how these sequences can be grouped to form specifications, whereas the latter reflects only the compositional structure of specifications after they have been properly grouped.

The abstract syntax of Casl plays a particularly central rôle: not only is it the basis for the Casl semantics, which is explained informally in Part I and defined formally in Part III, but also the abstract syntax of Casl specifications can be stored in libraries, so that tools can process the specifications without having to (re)parse them.

In acknowledgment of the importance of abstract syntax, consideration of concrete syntax for Casl was deferred until after the design of the abstract syntax – and of most of the details of its semantics – had been settled. The presentation of the Casl syntax here reflects the priority given to the abstract syntax:

- Chapter 2 specifies the abstract syntax of Casl;
- Chapter 3 gives a context-free grammar for Casl specifications, indicating also how ambiguities are resolved;
- Chapter 4 specifies the grouping of sequences of characters into sequences of lexical symbols, and determines their display format; and finally,
- Chapter 5 explains the form and use of comments and annotations (which are both included in abstract syntax, but have no effect on the semantics of specifications).

The relationship between the concrete syntax and the abstract syntax is rather straightforward – except that mapping the use of mixfix notation in a concrete `ATOM` to an abstract `ATOM` depends on the declared operation and predicate symbols (although not on their profiles). Here, the relationship is merely suggested by the use of the same nonterminal symbols in the concrete and abstract grammars.

Acknowledgement. The design of the abstract syntax of CASL has been heavily influenced by the work of the CoFI Semantics group on the formal semantics of CASL (see Part III).

The initial design of the concrete syntax (both input syntax and display format) of CASL was produced by Michel Bidoit, Christine Choppy, Bernd Krieg-Brückner, and Frédéric Voisin (with Peter Mosses as moderator of the lively discussions). Feedback from the development of various prototype parsers for CASL by Hubert Baumeister, Mark van den Brand, Kolyang, Christian Maeder, Till Mossakowski, Markus Roggenbach, Axel Schairer, Christophe Tronche, Frédéric Voisin, and Bjarke Wedemeijer contributed significantly to the final design of the concrete syntax.

2

Abstract Syntax

The abstract syntax is central to the definition of a formal language. It stands between the concrete representations of documents, such as marks on paper or images on screens, and the abstract entities, semantic relations, and semantic functions used for defining their meaning.

The abstract syntax has the following objectives:

- to identify and separately name the abstract syntactic entities;
- to simplify and unify underlying concepts, putting like things with like, and reducing unnecessary duplication.

There are many possible ways of constructing an abstract syntax, and the choice of form is a matter of judgment, taking into account the somewhat conflicting aims of simplicity and economy of semantic definition.

The abstract syntax is presented as a set of production rules in which each kind of entity is defined in terms of its sub-kinds:

```
SOME-KIND       ::=  SUB-KIND-1  |  ...  |  SUB-KIND-n
```

or in terms of its constructor and components:

```
SOME-CONSTRUCT  ::=  some-construct COMPONENT-1 ... COMPONENT-n
```

The productions form a context-free grammar; algebraically, the nonterminal symbols of the grammar correspond to sorts (of trees), and the terminal symbols correspond to constructor operations. The notation COMPONENT* indicates repetition of COMPONENT any number of times; COMPONENT+ indicates repetition at least once. These repetitions can be replaced by auxiliary sorts and constructs, after which it would be straightforward to transform the grammar into a Casl library of specifications using datatype declarations.

The context conditions for well-formedness of specifications are context-sensitive, and considered as part of the Casl semantics, see Part III.

Many constructs can have comments and annotations attached to them (see Sect. 5.2), but these are not shown in the grammar.

2.1 Normal Grammar

The grammar given in this section has the property that there is a nonterminal for each abstract construct (although an exception is made for constant constructs with no components). Section 2.2 provides an abbreviated grammar defining the same abstract syntax.

The following nonterminal symbols correspond to the lexical syntax, and are left unspecified in the abstract syntax: `WORDS`, `DOT-WORDS`, `SIGNS`, `DIGIT`, `DIGITS`, `NUMBER`, `QUOTED-CHAR`, `PLACE`, `URL`, and `PATH`.

2.1.1 Basic Specifications

```
BASIC-SPEC        ::= basic-spec BASIC-ITEMS*

BASIC-ITEMS       ::= SIG-ITEMS | FREE-DATATYPE | SORT-GEN
                    | VAR-ITEMS | LOCAL-VAR-AXIOMS | AXIOM-ITEMS

SIG-ITEMS         ::= SORT-ITEMS | OP-ITEMS | PRED-ITEMS
                    | DATATYPE-ITEMS

SORT-ITEMS        ::= sort-items SORT-ITEM+
SORT-ITEM         ::= SORT-DECL

SORT-DECL         ::= sort-decl SORT+

OP-ITEMS          ::= op-items OP-ITEM+
OP-ITEM           ::= OP-DECL | OP-DEFN

OP-DECL           ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
OP-TYPE           ::= TOTAL-OP-TYPE | PARTIAL-OP-TYPE
TOTAL-OP-TYPE     ::= total-op-type SORT-LIST SORT
PARTIAL-OP-TYPE   ::= partial-op-type SORT-LIST SORT
SORT-LIST         ::= sort-list SORT*
OP-ATTR           ::= BINARY-OP-ATTR | UNIT-OP-ATTR
BINARY-OP-ATTR    ::= assoc-op-attr | comm-op-attr | idem-op-attr
UNIT-OP-ATTR      ::= unit-op-attr TERM

OP-DEFN           ::= op-defn OP-NAME OP-HEAD TERM
OP-HEAD           ::= TOTAL-OP-HEAD | PARTIAL-OP-HEAD
TOTAL-OP-HEAD     ::= total-op-head ARG-DECL* SORT
PARTIAL-OP-HEAD   ::= partial-op-head ARG-DECL* SORT
ARG-DECL          ::= arg-decl VAR+ SORT

PRED-ITEMS        ::= pred-items PRED-ITEM+
PRED-ITEM         ::= PRED-DECL | PRED-DEFN

PRED-DECL         ::= pred-decl PRED-NAME+ PRED-TYPE
PRED-TYPE         ::= pred-type SORT-LIST
```

```
PRED-DEFN         ::= pred-defn PRED-NAME PRED-HEAD FORMULA
PRED-HEAD         ::= pred-head ARG-DECL*

DATATYPE-ITEMS    ::= datatype-items DATATYPE-DECL+
DATATYPE-DECL     ::= datatype-decl SORT ALTERNATIVE+
ALTERNATIVE       ::= TOTAL-CONSTRUCT | PARTIAL-CONSTRUCT
TOTAL-CONSTRUCT   ::= total-construct   OP-NAME COMPONENTS*
PARTIAL-CONSTRUCT::= partial-construct OP-NAME COMPONENTS+
COMPONENTS        ::= TOTAL-SELECT | PARTIAL-SELECT | SORT
TOTAL-SELECT      ::= total-select   OP-NAME+ SORT
PARTIAL-SELECT    ::= partial-select OP-NAME+ SORT

FREE-DATATYPE     ::= free-datatype DATATYPE-ITEMS

SORT-GEN          ::= sort-gen SIG-ITEMS+

VAR-ITEMS         ::= var-items VAR-DECL+
VAR-DECL          ::= var-decl VAR+ SORT

LOCAL-VAR-AXIOMS ::= local-var-axioms VAR-DECL+ AXIOM+

AXIOM-ITEMS       ::= axiom-items AXIOM+

AXIOM             ::= FORMULA
FORMULA           ::= QUANTIFICATION | CONJUNCTION | DISJUNCTION
                    | IMPLICATION | EQUIVALENCE | NEGATION | ATOM
QUANTIFICATION    ::= quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER        ::= universal | existential | unique-existential
CONJUNCTION       ::= conjunction FORMULA+
DISJUNCTION       ::= disjunction FORMULA+
IMPLICATION       ::= implication FORMULA FORMULA
EQUIVALENCE       ::= equivalence FORMULA FORMULA
NEGATION          ::= negation FORMULA

ATOM              ::= TRUTH | PREDICATION | DEFINEDNESS
                    | EXISTL-EQUATION | STRONG-EQUATION
TRUTH             ::= true-atom | false-atom
PREDICATION       ::= predication PRED-SYMB TERMS
PRED-SYMB         ::= PRED-NAME | QUAL-PRED-NAME
QUAL-PRED-NAME    ::= qual-pred-name PRED-NAME PRED-TYPE
DEFINEDNESS       ::= definedness TERM
EXISTL-EQUATION   ::= existl-equation TERM TERM
STRONG-EQUATION   ::= strong-equation TERM TERM

TERMS             ::= terms TERM*
TERM              ::= SIMPLE-ID | QUAL-VAR | APPLICATION
                    | SORTED-TERM | CONDITIONAL
QUAL-VAR          ::= qual-var VAR SORT
APPLICATION       ::= application OP-SYMB TERMS
```

```
OP-SYMB          ::= OP-NAME | QUAL-OP-NAME
QUAL-OP-NAME     ::= qual-op-name OP-NAME OP-TYPE
SORTED-TERM      ::= sorted-term TERM SORT
CONDITIONAL      ::= conditional TERM FORMULA TERM

SORT             ::= SORT-ID
OP-NAME          ::= ID
PRED-NAME        ::= ID
VAR              ::= SIMPLE-ID

SORT-ID          ::= WORDS
SIMPLE-ID        ::= WORDS
ID               ::= id MIX-TOKEN+
MIX-TOKEN        ::= TOKEN | PLACE | BRACED-ID | BRACKET-ID | EMPTY-BRS
TOKEN            ::= WORDS | DOT-WORDS | SIGNS | DIGIT | QUOTED-CHAR
BRACED-ID        ::= braced-id ID
BRACKET-ID       ::= bracket-id ID
EMPTY-BRS        ::= empty-braces | empty-brackets
```

2.1.2 Subsorting Specifications

```
SORT-ITEM        ::= ... | SUBSORT-DECL | ISO-DECL | SUBSORT-DEFN

SUBSORT-DECL     ::= subsort-decl SORT+ SORT
ISO-DECL         ::= iso-decl SORT+
SUBSORT-DEFN     ::= subsort-defn SORT VAR SORT FORMULA

ALTERNATIVE      ::= ... | SUBSORTS
SUBSORTS         ::= subsorts SORT+

ATOM             ::= ... | MEMBERSHIP
MEMBERSHIP       ::= membership TERM SORT

TERM             ::= ... | CAST
CAST             ::= cast TERM SORT
```

2.1.3 Structured Specifications

```
SPEC             ::= BASIC-SPEC | TRANSLATION | REDUCTION
                   | UNION | EXTENSION | FREE-SPEC | LOCAL-SPEC
                   | CLOSED-SPEC | SPEC-INST

TRANSLATION      ::= translation SPEC RENAMING
RENAMING         ::= renaming  SYMB-MAP-ITEMS+

REDUCTION        ::= reduction SPEC RESTRICTION
RESTRICTION      ::= HIDDEN | REVEALED
HIDDEN           ::= hidden   SYMB-ITEMS+
REVEALED         ::= revealed SYMB-MAP-ITEMS+

UNION            ::= union SPEC+
EXTENSION        ::= extension SPEC+
FREE-SPEC        ::= free-spec SPEC
LOCAL-SPEC       ::= local-spec SPEC SPEC
CLOSED-SPEC      ::= closed-spec SPEC

SPEC-DEFN        ::= spec-defn SPEC-NAME GENERICITY SPEC
GENERICITY       ::= genericity PARAMS IMPORTED
PARAMS           ::= params SPEC*
IMPORTED         ::= imported SPEC*

SPEC-INST        ::= spec-inst SPEC-NAME FIT-ARG*

FIT-ARG          ::= FIT-SPEC | FIT-VIEW
FIT-SPEC         ::= fit-spec SPEC SYMB-MAP-ITEMS*
FIT-VIEW         ::= fit-view VIEW-NAME FIT-ARG*

VIEW-DEFN        ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE
                               SYMB-MAP-ITEMS*
VIEW-TYPE        ::= view-type SPEC SPEC

SYMB-ITEMS       ::= symb-items     SYMB-KIND SYMB+
SYMB-MAP-ITEMS   ::= symb-map-items SYMB-KIND SYMB-OR-MAP+
SYMB-KIND        ::= implicit | sorts-kind | ops-kind | preds-kind

SYMB             ::= ID | QUAL-ID
QUAL-ID          ::= qual-id ID TYPE
TYPE             ::= OP-TYPE | PRED-TYPE
SYMB-MAP         ::= symb-map SYMB SYMB
SYMB-OR-MAP      ::= SYMB | SYMB-MAP

SPEC-NAME        ::= SIMPLE-ID
VIEW-NAME        ::= SIMPLE-ID

SORT-ID          ::= ... | COMP-SORT-ID
COMP-SORT-ID     ::= comp-sort-id WORDS ID+
MIX-TOKEN        ::= ... | COMP-MIX-TOKEN
COMP-MIX-TOKEN   ::= comp-mix-token ID+
```

2.1.4 Architectural Specifications

```
ARCH-SPEC-DEFN   ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC        ::= BASIC-ARCH-SPEC | ARCH-SPEC-NAME
BASIC-ARCH-SPEC  ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT

UNIT-DECL-DEFN   ::= UNIT-DECL | UNIT-DEFN
UNIT-DECL        ::= unit-decl UNIT-NAME UNIT-SPEC UNIT-IMPORTED
```

```
UNIT-IMPORTED    ::= unit-imported UNIT-TERM*
UNIT-DEFN        ::= unit-defn UNIT-NAME UNIT-EXPRESSION

UNIT-SPEC-DEFN   ::= unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC        ::= UNIT-TYPE | SPEC-NAME | ARCH-UNIT-SPEC
                   | CLOSED-UNIT-SPEC
ARCH-UNIT-SPEC   ::= arch-unit-spec ARCH-SPEC
CLOSED-UNIT-SPEC ::= closed-unit-spec UNIT-SPEC
UNIT-TYPE        ::= unit-type SPEC* SPEC

RESULT-UNIT      ::= result-unit UNIT-EXPRESSION
UNIT-EXPRESSION  ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING     ::= unit-binding UNIT-NAME UNIT-SPEC
UNIT-TERM        ::= UNIT-REDUCTION | UNIT-TRANSLATION | AMALGAMATION
                   | LOCAL-UNIT | UNIT-APPL
UNIT-TRANSLATION ::= unit-translation UNIT-TERM RENAMING
UNIT-REDUCTION   ::= unit-reduction UNIT-TERM RESTRICTION
AMALGAMATION     ::= amalgamation UNIT-TERM+
LOCAL-UNIT       ::= local-unit UNIT-DEFN+ UNIT-TERM
UNIT-APPL        ::= unit-appl UNIT-NAME FIT-ARG-UNIT*
FIT-ARG-UNIT     ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*

ARCH-SPEC-NAME   ::= SIMPLE-ID
UNIT-NAME        ::= SIMPLE-ID
```

2.1.5 Specification Libraries

```
LIB-DEFN         ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM         ::= SPEC-DEFN | VIEW-DEFN
                   | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
                   | DOWNLOAD-ITEMS

DOWNLOAD-ITEMS   ::= download-items LIB-NAME ITEM-NAME-OR-MAP+
ITEM-NAME-OR-MAP ::= ITEM-NAME | ITEM-NAME-MAP
ITEM-NAME-MAP    ::= item-name-map ITEM-NAME ITEM-NAME
ITEM-NAME        ::= SIMPLE-ID

LIB-NAME         ::= LIB-ID | LIB-VERSION
LIB-VERSION      ::= lib-version LIB-ID VERSION-NUMBER
VERSION-NUMBER   ::= version-number NUMBER+
LIB-ID           ::= DIRECT-LINK | INDIRECT-LINK
DIRECT-LINK      ::= direct-link URL
INDIRECT-LINK    ::= indirect-link PATH
```

2.2 Abbreviated Grammar

This section provides an abbreviated grammar that defines the same (tree) language as the one in Sect. 2.1. It was obtained by eliminating each nonterminal that corresponds to a single construct, when this nonterminal occurs only once as an alternative.

As in Sect. 2.1, the following nonterminal symbols correspond to lexical syntax, and are left unspecified in the abstract syntax: WORDS, DOT-WORDS, SIGNS, DIGIT, DIGITS, NUMBER, QUOTED-CHAR, PLACE, URL, and PATH.

2.2.1 Basic Specifications

```
BASIC-SPEC       ::= basic-spec BASIC-ITEMS*

BASIC-ITEMS      ::= SIG-ITEMS
                   | free-datatype DATATYPE-DECL+
                   | sort-gen SIG-ITEMS+
                   | var-items VAR-DECL+
                   | local-var-axioms VAR-DECL+ FORMULA+
                   | axiom-items FORMULA+

SIG-ITEMS        ::= sort-items SORT-ITEM+
                   | op-items OP-ITEM+
                   | pred-items PRED-ITEM+
                   | datatype-items DATATYPE-DECL+

SORT-ITEM        ::= sort-decl SORT+

OP-ITEM          ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
                   | op-defn OP-NAME  OP-HEAD TERM
OP-TYPE          ::= total-op-type   SORT-LIST SORT
                   | partial-op-type SORT-LIST SORT

SORT-LIST        ::= sort-list SORT*

OP-ATTR          ::= assoc-op-attr | comm-op-attr | idem-op-attr
                   | unit-op-attr TERM

OP-HEAD          ::= total-op-head   ARG-DECL* SORT
                   | partial-op-head ARG-DECL* SORT

ARG-DECL         ::= arg-decl VAR+ SORT

PRED-ITEM        ::= pred-decl PRED-NAME+ PRED-TYPE
                   | pred-defn PRED-NAME  PRED-HEAD FORMULA
PRED-TYPE        ::= pred-type SORT-LIST
PRED-HEAD        ::= pred-head ARG-DECL*
```

```
DATATYPE-DECL    ::= datatype-decl SORT ALTERNATIVE+

ALTERNATIVE      ::= total-construct   OP-NAME COMPONENTS*
                   | partial-construct OP-NAME COMPONENTS+

COMPONENTS       ::= total-select   OP-NAME+ SORT
                   | partial-select OP-NAME+ SORT
                   | SORT

VAR-DECL         ::= var-decl VAR+ SORT

FORMULA          ::= quantification QUANTIFIER VAR-DECL+ FORMULA
                   | conjunction FORMULA+
                   | disjunction FORMULA+
                   | implication FORMULA FORMULA
                   | equivalence FORMULA FORMULA
                   | negation FORMULA
                   | true-atom | false-atom
                   | predication PRED-SYMB TERMS
                   | definedness TERM
                   | existl-equation TERM TERM
                   | strong-equation TERM TERM

QUANTIFIER       ::= universal | existential | unique-existential

PRED-SYMB        ::= PRED-NAME | qual-pred-name PRED-NAME PRED-TYPE

TERMS            ::= terms TERM*
TERM             ::= SIMPLE-ID
                   | qual-var VAR SORT
                   | application OP-SYMB TERMS
                   | sorted-term TERM SORT
                   | conditional TERM FORMULA TERM

OP-SYMB          ::= OP-NAME | qual-op-name OP-NAME OP-TYPE

SORT             ::= SORT-ID
OP-NAME          ::= ID
PRED-NAME        ::= ID
VAR              ::= SIMPLE-ID

SORT-ID          ::= WORDS
ID               ::= id MIX-TOKEN+
SIMPLE-ID        ::= WORDS
MIX-TOKEN        ::= TOKEN | PLACE
                   | bracket-id ID | empty-brackets
                   | braced-id ID | empty-braces
TOKEN            ::= WORDS | DOT-WORDS | SIGNS | DIGIT | QUOTED-CHAR
```

2.2.2 Subsorting Specifications

```
SORT-ITEM        ::= ...
                   | subsort-decl SORT+ SORT
                   | subsort-defn SORT VAR SORT FORMULA
                   | iso-decl SORT+

ALTERNATIVE      ::= ...
                   | subsorts SORT+

FORMULA          ::= ...
                   | membership TERM SORT

TERM             ::= ...
                   | cast TERM SORT
```

2.2.3 Structured Specifications

```
SPEC             ::= BASIC-SPEC
                   | translation SPEC RENAMING
                   | reduction SPEC RESTRICTION
                   | union SPEC+
                   | extension SPEC+
                   | free-spec SPEC
                   | local-spec SPEC SPEC
                   | closed-spec SPEC
                   | spec-inst SPEC-NAME FIT-ARG*

RENAMING         ::= renaming SYMB-MAP-ITEMS+

RESTRICTION      ::= hide   SYMB-ITEMS+
                   | reveal SYMB-MAP-ITEMS+

SPEC-DEFN        ::= spec-defn SPEC-NAME GENERICITY SPEC
GENERICITY       ::= genericity PARAMS IMPORTED
PARAMS           ::= params SPEC*
IMPORTED         ::= imported SPEC*

FIT-ARG          ::= fit-spec SPEC SYMB-MAP-ITEMS*
                   | fit-view VIEW-NAME FIT-ARG*

VIEW-DEFN        ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE
                               SYMB-MAP-ITEMS*
VIEW-TYPE        ::= view-type SPEC SPEC

SYMB-ITEMS       ::= symb-items     SYMB-KIND SYMB+
SYMB-MAP-ITEMS   ::= symb-map-items SYMB-KIND SYMB-OR-MAP+
SYMB-KIND        ::= implicit | sorts-kind | ops-kind | preds-kind
```

```
SYMB             ::= ID | qual-id ID TYPE
TYPE             ::= OP-TYPE | PRED-TYPE
SYMB-MAP         ::= symb-map SYMB SYMB
SYMB-OR-MAP      ::= SYMB | SYMB-MAP

SPEC-NAME        ::= SIMPLE-ID
VIEW-NAME        ::= SIMPLE-ID

SORT-ID          ::= ... | COMP-SORT-ID
MIX-TOKEN        ::= ... | COMP-MIX-TOKEN
COMP-SORT-ID     ::= comp-sort-id WORDS ID+
COMP-MIX-TOKEN   ::= comp-mix-token ID+
```

2.2.4 Architectural Specifications

```
ARCH-SPEC-DEFN   ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC        ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT
                   | ARCH-SPEC-NAME
UNIT-DECL-DEFN   ::= UNIT-DECL | UNIT-DEFN

UNIT-DECL        ::= unit-decl UNIT-NAME UNIT-SPEC UNIT-IMPORTED
UNIT-IMPORTED    ::= unit-imported UNIT-TERM*
UNIT-DEFN        ::= unit-defn UNIT-NAME UNIT-EXPRESSION

UNIT-SPEC-DEFN   ::= unit-spec-defn SPEC-NAME UNIT-SPEC
UNIT-SPEC        ::= UNIT-TYPE | SPEC-NAME | arch-unit-spec ARCH-SPEC
                   | closed-unit-spec UNIT-SPEC
UNIT-TYPE        ::= unit-type SPEC* SPEC

RESULT-UNIT      ::= result-unit UNIT-EXPRESSION
UNIT-EXPRESSION  ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING     ::= unit-binding UNIT-NAME UNIT-SPEC
UNIT-TERM        ::= unit-translation UNIT-TERM RENAMING
                   | unit-reduction UNIT-TERM RESTRICTION
                   | amalgamation UNIT-TERM+
                   | local-unit UNIT-DEFN+ UNIT-TERM
                   | unit-appl UNIT-NAME FIT-ARG-UNIT*
FIT-ARG-UNIT     ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*

ARCH-SPEC-NAME   ::= SIMPLE-ID
UNIT-NAME        ::= SIMPLE-ID
```

2.2.5 Specification Libraries

```
LIB-DEFN         ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM         ::= SPEC-DEFN | VIEW-DEFN
                   | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
                   | download-items LIB-NAME ITEM-NAME-OR-MAP+
ITEM-NAME-OR-MAP ::= ITEM-NAME | item-name-map ITEM-NAME ITEM-NAME
ITEM-NAME        ::= SIMPLE-ID

LIB-NAME         ::= LIB-ID | LIB-VERSION
LIB-VERSION      ::= lib-version LIB-ID VERSION-NUMBER
VERSION-NUMBER   ::= version-number NUMBER+
LIB-ID           ::= direct-link URL | indirect-link PATH
```

3

Concrete Syntax

Section 3.1 gives a context-free grammar for the concrete syntax of CASL. The grammar is ambiguous; Sect. 3.2 explains various precedence rules for disambiguation, and the intended grouping of mixfix formulas and terms.

The following meta-notation for context-free grammars is used not only for specifying the grouping syntax of CASL in this chapter, but also for specifying lexical symbols in Chaps. 4, and comments and annotations in Chap. 5.

Nonterminal symbols are written as uppercase words, possibly hyphenated, e.g., `SORT`, `BASIC-SPEC`.

Terminal symbols are written as either:

- lowercase words, e.g. `free`, `assoc`; or
- sequences of characters enclosed in double-quotes, e.g. `"."`, `"::="`; or
- sequences of characters enclosed in single quotes, e.g. `'"'`, `'\"'`.

When sequences of characters cannot be confused with the meta-notation introduced below, the enclosing quotes are usually omitted.

Sequences of symbols are written with spaces between the symbols. The empty sequence is denoted by the reserved nonterminal symbol `EMPTY`.

Optional symbols are underlined, e.g. $\underline{\texttt{end}}$, $\underline{\texttt{;}}$. This is used also for the optional plural 's' at the end of some lowercase words used as terminal symbols, e.g. $\texttt{sort}\underline{\texttt{s}}$.

Repetitions are indicated by ellipsis '`...`', e.g. `MIXFIX...MIXFIX` denotes one or more occurrences of `MIXFIX`, and `[SPEC]...[SPEC]` denotes one or more occurrences of `[SPEC]`. Repetitions often involve separators, e.g. `SORT,...,SORT` denotes one or more occurrences of `SORT` separated by '`,`'.

Alternative sequences are separated by vertical bars, e.g. `idem | unit TERM` where the alternatives are `idem` and `unit TERM`.

Production rules are written with the nonterminal symbol followed by '`::=`', followed by one or more alternatives. When a production extends a previously-given production for the same nonterminal symbol, this is indicated by writing '`...`' as its first alternative.

Start symbols are not specified.

3.1 Context-Free Grammar

The lexical symbols of Casl are given in Chap. 4. They consist of:

- key words and symbols;
- tokens: `WORDS`, `DOT-WORDS`, `DIGIT`, `SIGNS`, and `QUOTED-CHAR`;
- literals: `STRING`, `DIGITS`, `NUMBER`, `FRACTION`, and `FLOATING`;
- `URL` and `PATH`; and
- `COMMENT` and `ANNOTATION`.

The context-free grammar of Casl below treats these lexical symbols as terminal symbols. The language generated by this grammar is both LALR(1) and LL(1), and parsers can be generated from appropriate deterministic grammars using tools such as ML-yacc and Haskell combinator parsers.

Lexical analysis for Casl is generally independent of the context-free parsing (apart from the recognition of `NUMBER`, `URL` and `PATH`, which may appear in libraries but not within individual specifications).

Context-free parsing of Casl specifications according to the grammar in this section yields a parse tree where terms and formulas occurring in axioms and definitions have been grouped with respect to explicit parentheses and brackets, but where the intended applicative structure has not yet been recognized. A further phase of *mixfix grouping analysis* is needed, dependent on the identifiers declared in the specification and on parsing annotations, before the parse tree can be mapped to a complete abstract syntax tree.

3.1.1 Basic Specifications

```
BASIC-SPEC    ::= BASIC-ITEMS...BASIC-ITEMS  |  { }

BASIC-ITEMS   ::= SIG-ITEMS
                | free      types DATATYPE-DECL ;...; DATATYPE-DECL ;
                | generated types DATATYPE-DECL ;...; DATATYPE-DECL ;
                | generated { SIG-ITEMS...SIG-ITEMS } ;
                | vars VAR-DECL ;...; VAR-DECL ;
                | forall VAR-DECL ;...; VAR-DECL
                           "." FORMULA "."..."." FORMULA ;
                | "." FORMULA "."..."." FORMULA ;

SIG-ITEMS     ::= sorts SORT-ITEM ;...; SORT-ITEM ;
                | ops OP-ITEM ;...; OP-ITEM ;
                | preds PRED-ITEM ;...; PRED-ITEM  ;
                | types DATATYPE-DECL ;...; DATATYPE-DECL ;

SORT-ITEM     ::= SORT ,..., SORT

OP-ITEM       ::= OP-NAME ,..., OP-NAME : OP-TYPE
                | OP-NAME ,..., OP-NAME :
                  OP-TYPE , OP-ATTR ,..., OP-ATTR
                | OP-NAME OP-HEAD = TERM
```

```
OP-TYPE         ::= SORT *...* SORT -> SORT    |  SORT
                  | SORT *...* SORT -> ? SORT  |  ? SORT

OP-ATTR         ::= assoc  |  comm  |  idem  |  unit TERM

OP-HEAD         ::= ( ARG-DECL ;...; ARG-DECL ) : SORT    |  : SORT
                  | ( ARG-DECL ;...; ARG-DECL ) : ? SORT  |  : ? SORT

ARG-DECL        ::= VAR ,..., VAR : SORT

PRED-ITEM       ::= PRED-NAME ,..., PRED-NAME : PRED-TYPE
                  | PRED-NAME PRED-HEAD <=> FORMULA
                  | PRED-NAME <=> FORMULA

PRED-TYPE       ::= SORT *...* SORT  |  ( )

PRED-HEAD       ::= ( ARG-DECL ;...; ARG-DECL )

DATATYPE-DECL   ::= SORT "::=" ALTERNATIVE "|"..."|" ALTERNATIVE

ALTERNATIVE     ::= OP-NAME  ( COMPONENT ;...; COMPONENT )
                  | OP-NAME  ( COMPONENT ;...; COMPONENT ) ?
                  | OP-NAME

COMPONENT       ::= OP-NAME ,..., OP-NAME : SORT
                  | OP-NAME ,..., OP-NAME : ? SORT
                  | SORT

VAR-DECL        ::= VAR ,..., VAR : SORT

FORMULA         ::= QUANTIFIER  VAR-DECL ;...; VAR-DECL "." FORMULA
                  | FORMULA /\ FORMULA /\.../\ FORMULA
                  | FORMULA \/ FORMULA \/...\/ FORMULA
                  | FORMULA => FORMULA
                  | FORMULA if FORMULA
                  | FORMULA <=> FORMULA
                  | not FORMULA
                  | true | false
                  | def TERM
                  | TERM =e= TERM
                  | TERM = TERM
                  | ( FORMULA )
                  | MIXFIX...MIXFIX

QUANTIFIER      ::= forall  |  exists  |  exists!

TERMS           ::= TERM ,..., TERM

TERM            ::= MIXFIX...MIXFIX
```

```
MIXFIX          ::= TOKEN  |  LITERAL  |  PLACE
                  | QUAL-PRED-NAME  |  QUAL-VAR-NAME  |  QUAL-OP-NAME
                  | TERM : SORT
                  | TERM when FORMULA else TERM
                  | ( TERMS )
                  | [ TERMS ]  |  [ ]
                  | { TERMS }  |  { }

QUAL-VAR-NAME   ::= ( var VAR : SORT )

QUAL-PRED-NAME  ::= ( pred PRED-NAME : PRED-TYPE )

QUAL-OP-NAME    ::= ( op OP-NAME : OP-TYPE )

SORT            ::= SORT-ID

OP-NAME         ::= ID

PRED-NAME       ::= ID

VAR             ::= SIMPLE-ID

TOKEN           ::= WORDS  |  DOT-WORDS  |  DIGIT  |  SIGNS
                  | QUOTED-CHAR

LITERAL         ::= STRING  |  DIGITS  |  FRACTION  |  FLOATING

PLACE           ::= __


SORT-ID         ::= WORDS

SIMPLE-ID       ::= WORDS

ID              ::= MIX-TOKEN ... MIX-TOKEN

MIX-TOKEN       ::= TOKEN  |  PLACE
                  | [ ID ]  |  [ ]
                  | { ID }  |  { }
```

3.1.2 Subsorting Specifications

```
SORT-ITEM       ::= ...
                  | SORT ,..., SORT < SORT
                  | SORT = { VAR : SORT "." FORMULA }
                  | SORT =...= SORT

ALTERNATIVE     ::= ...
                  | sorts SORT ,..., SORT
```

```
FORMULA         ::= ...
                  | TERM in SORT
MIXFIX          ::= ...
                  | TERM as SORT
```

3.1.3 Structured Specifications

```
SPEC            ::= BASIC-SPEC
                  | SPEC RENAMING
                  | SPEC RESTRICTION
                  | SPEC and SPEC and...and SPEC
                  | SPEC then SPEC then...then SPEC
                  | free GROUP-SPEC
                  | local SPEC within SPEC
                  | closed GROUP-SPEC
                  | GROUP-SPEC

GROUP-SPEC      ::= { SPEC }
                  | SPEC-NAME
                  | SPEC-NAME [ FIT-ARG ]...[ FIT-ARG ]

RENAMING        ::= with SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS

RESTRICTION     ::= hide SYMB-ITEMS ,..., SYMB-ITEMS
                  | reveal SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS

SPEC-DEFN       ::= spec SPEC-NAME = SPEC end
                  | spec SPEC-NAME SOME-GENERICS = SPEC end

SOME-GENERICS   ::= SOME-PARAMS  |  SOME-PARAMS SOME-IMPORTED

SOME-PARAMS     ::= [ SPEC ]...[ SPEC ]

SOME-IMPORTED   ::= given GROUP-SPEC ,..., GROUP-SPEC

FIT-ARG         ::= SPEC fit SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS
                  | SPEC
                  | view VIEW-NAME
                  | view VIEW-NAME [ FIT-ARG ]...[ FIT-ARG ]

VIEW-DEFN       ::= view VIEW-NAME : VIEW-TYPE end
                  | view VIEW-NAME : VIEW-TYPE =
                         SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS end
                  | view VIEW-NAME SOME-GENERICS : VIEW-TYPE end
                  | view VIEW-NAME SOME-GENERICS : VIEW-TYPE =
                         SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS end

VIEW-TYPE       ::= GROUP-SPEC to GROUP-SPEC
```

```
SYMB-ITEMS      ::= SYMB
                  | SOME-SYMB-KIND SYMB ,..., SYMB

SYMB-MAP-ITEMS  ::= SYMB-OR-MAP
                  | SOME-SYMB-KIND SYMB-OR-MAP ,..., SYMB-OR-MAP

SOME-SYMB-KIND  ::= sorts  |  ops  |  preds

SYMB            ::= ID  |  ID : TYPE

TYPE            ::= OP-TYPE  |  PRED-TYPE

SYMB-MAP        ::= SYMB "|->" SYMB

SYMB-OR-MAP     ::= SYMB  |  SYMB-MAP

SPEC-NAME       ::= SIMPLE-ID
VIEW-NAME       ::= SIMPLE-ID

SORT-ID         ::= ...  |  WORDS [ ID ,..., ID ]

MIX-TOKEN       ::= ...  |  [ ID ,..., ID ]
```

3.1.4 Architectural Specifications

```
ARCH-SPEC-DEFN  ::= arch spec ARCH-SPEC-NAME = ARCH-SPEC end

ARCH-SPEC       ::= BASIC-ARCH-SPEC  |  GROUP-ARCH-SPEC

GROUP-ARCH-SPEC ::= { ARCH-SPEC }  |  ARCH-SPEC-NAME

BASIC-ARCH-SPEC ::= units UNIT-DECL-DEFN ;...; UNIT-DECL-DEFN ;
                              result UNIT-EXPRESSION ;

UNIT-DECL-DEFN  ::= UNIT-DECL  |  UNIT-DEFN

UNIT-DECL       ::= UNIT-NAME : UNIT-SPEC
                          given GROUP-UNIT-TERM ,..., GROUP-UNIT-TERM
                  | UNIT-NAME : UNIT-SPEC

UNIT-DEFN       ::= UNIT-NAME = UNIT-EXPRESSION

UNIT-SPEC-DEFN  ::= unit spec SPEC-NAME = UNIT-SPEC end

UNIT-SPEC       ::= GROUP-SPEC
                  | GROUP-SPEC *...* GROUP-SPEC -> GROUP-SPEC
                  | arch spec GROUP-ARCH-SPEC
                  | closed UNIT-SPEC
```

```
UNIT-EXPRESSION ::= lambda UNIT-BINDING ;...;
                         UNIT-BINDING "." UNIT-TERM
                  | UNIT-TERM

UNIT-BINDING    ::= UNIT-NAME : UNIT-SPEC

UNIT-TERM       ::= UNIT-TERM RENAMING
                  | UNIT-TERM RESTRICTION
                  | UNIT-TERM and...and UNIT-TERM
                  | local UNIT-DEFN ;...; UNIT-DEFN ; within UNIT-TERM
                  | GROUP-UNIT-TERM

GROUP-UNIT-TERM ::= { UNIT-TERM }
                  | UNIT-NAME
                  | UNIT-NAME [ FIT-ARG-UNIT ]...[ FIT-ARG-UNIT ]

FIT-ARG-UNIT    ::= UNIT-TERM
                  | UNIT-TERM fit SYMB-MAP-ITEMS ,..., SYMB-MAP-ITEMS

ARCH-SPEC-NAME  ::= SIMPLE-ID
UNIT-NAME       ::= SIMPLE-ID
```

3.1.5 Specification Libraries

```
LIB-DEFN        ::= library LIB-NAME LIB-ITEM...LIB-ITEM

LIB-ITEM        ::= SPEC-DEFN  |  VIEW-DEFN
                  | ARCH-SPEC-DEFN  |  UNIT-SPEC-DEFN
                  | from LIB-NAME
                       get ITEM-NAME-OR-MAP ,..., ITEM-NAME-OR-MAP end

ITEM-NAME-OR-MAP::= ITEM-NAME  |  ITEM-NAME "|->" ITEM-NAME

ITEM-NAME       ::= SIMPLE-ID

LIB-NAME        ::= LIB-ID  |  LIB-ID VERSION-NUMBER

VERSION-NUMBER  ::= version NUMBER "."..."." NUMBER
```

3.2 Disambiguation

The context-free grammar given in Sect. 3.1 for input syntax is quite ambiguous. This section explains various precedence rules for disambiguation, and the intended grouping of mixfix formulas and terms (which is to be recognized in a separate phase, dependent on the declared symbols and parsing annotations).

3.2.1 Precedence

At the level of structured specifications, ambiguities of grouping are resolved as follows, in decreasing order of precedence:

- ‘`free`’ and ‘`closed`’.
- ‘`with`’, ‘`reveal`’, and ‘`hide`’.
- ‘`within`’.
- ‘`and`’.
- ‘`then`’.

At the level of architectural specifications, ambiguities of grouping in unit terms are resolved in the same way as for structured specifications. Moreover, a `SPEC-NAME` occurring as a `UNIT-SPEC` gives rise to just the `SPEC-NAME` itself in the abstract syntax tree, rather than a `UNIT-TYPE` with an empty list `SPEC*` of argument specifications.

In `BASIC-ITEMS`, a list of ‘`. FORMULA ... . FORMULA`’ extends as far to the right as possible. Within a `FORMULA`, the use of prefix and infix notation for the logical connectives gives rise to some potential ambiguities. These are resolved as follows, in decreasing order of precedence:

- ‘`not FORMULA`’.
- ‘`FORMULA /\.../\ FORMULA`’ and ‘`FORMULA \/...\/ FORMULA`’. These constructs may not be combined without explicit grouping.
- The connectives ‘`FORMULA => FORMULA`’, ‘`FORMULA if FORMULA`’, ‘`FORMULA <=> FORMULA`’. When repeated, ‘`=>`’ groups to the right, whereas ‘`if`’ groups to the left; ‘`<=>`’ may not be repeated without explicit grouping. These constructs may not be combined without explicit grouping.
- ‘`QUANTIFIER VAR-DECL;... . FORMULA`’. The last `FORMULA` extends as far to the right as possible, e.g., ‘`forall x:S . F => G`’ is disambiguated as ‘`forall x:S . (F => G)`’, not as ‘`(forall x:S . F) => G`’.
  Moreover, a quantification may be used on the right of a logical connective without grouping parentheses. For instance,
  ‘`F <=> exists x:s . G <=> H`’ is parsed as
  ‘`F <=> (exists x:s . G <=> H)`’.

The declaration[1] of infix, prefix, postfix, and general mixfix operation symbols may introduce further potential ambiguities, which are partially resolved as follows, in decreasing order of precedence (remaining ambiguities have to be eliminated by explicit use of grouping parentheses in terms, or by use of parsing annotations):

- Ordinary function application ‘`OP-SYMB(TERMS)`’.
- Applications of postfix symbols. This extends to all mixfix symbols of the form ‘t_0`__...__`t_n’ with t_0 empty, and to sorted terms and casts.

[1] Declarations occurring anywhere in the enclosing list of basic items are taken into account when disambiguating the grouping of symbols in a term.

- Applications of prefix symbols. This extends to all mixfix symbols of the form ‘t_0__. . .__t_n’ with t_n empty.
- Applications of infix symbols. This extends to all mixfix symbols of the form ‘t_0__. . .__t_n’ with both t_0 and t_n empty. Mixtures of different infix symbols and iterations of the same infix symbol have to be explicitly grouped – although the attribute of associativity implies a parsing annotation that allows iterated applications of that symbol to be written without grouping.
- The conditional ‘`TERM when FORMULA else TERM`’. Iterations such as:
  T_1 `when` F_1 `else` T_2 `when` F_2 `else` T_3
  are implicitly grouped to the right:
  T_1 `when` F_1 `else (`T_2 `when` F_2 `else` T_3`)`

Various other techniques for allowing the omission of grouping parentheses and/or list-separators in input (and display) are familiar from previous specification and programming languages, e.g., user-specified precedence (relative or absolute). Moreover, not all parsers are expected to implement full mixfix notation. CASL therefore allows *parsing annotations* on (libraries of) specifications, to indicate the possible omission of grouping parentheses, and the degree of use of mixfix notation. (Such annotations are expected to apply uniformly to CASL sublanguages, and to most extensions.) Parsing annotations may even override the rules given above for the relative precedence of postfix, prefix, and infix symbols. See Sect. 5.2.3 for details of the available parsing annotations.

3.2.2 Mixfix Grouping Analysis

Mixfix grouping analysis of a specification should be *equivalent* to context-free parsing according to a derived grammar – obtained from the grammar in Sect. 3.1 by replacing the phrases involving `MIXFIX` with phrases determined (partly) by the declared symbols, as follows:

```
FORMULA ::= ... | QUAL-PRED-NAME
              | QUAL-PRED-NAME ( TERMS )

TERMS   ::= TERM ,..., TERM

TERM    ::= LITERAL | QUAL-VAR-NAME | QUAL-OP-NAME
              | QUAL-OP-NAME ( TERMS )
              | TERM : SORT
              | TERM as SORT
              | TERM when FORMULA else TERM
              | ( TERM )
```

plus, for each declared variable or constant name *id*,

```
TERM    ::= ... | id
```

plus, for each declared operation symbol *id* of positive arity,

TERM ::= ... | id (TERMS)

plus, for each declared mixfix operation symbol ‘t_0__. . . .__t_n’ (with t_0 and t_n possibly empty),

TERM ::= ... | t_0 TERM ... TERM t_n

plus, for each annotation ‘%list b_1__b_2, c, f’,

TERM ::= ... | b_1 b_2

(provided that b_1 b_2 is different from c) and

TERM ::= ... | b_1 TERMS b_2

plus, for each declared predicate constant name *id*,

FORMULA ::= ... | id

plus, for each declared predicate symbol *id* of positive arity,

FORMULA ::= id (TERMS)

plus, for each declared mixfix predicate symbol ‘t_0__. . . .__t_n’ (with t_0 and t_n possibly empty),

FORMULA ::= t_0 TERM ... TERM t_n

It would be possible to obtain a fixed grammar for a sublanguage of Casl lacking mixfix notation in a similar way, using the appropriate kinds of ID in place of the declared *id*s above. (It may be convenient to obtain all these various grammars as extensions of a root grammar that is completely uncommitted about the notation used for applications, etc.)

The context-free parsing during mixfix grouping analysis involves disambiguation as determined by the general precedence rules for applications (see Sect. 3.2.1) and by any parsing annotations (see Sect. 5.2.3).

3.2.3 Mixfix Identifiers

An ID is well-formed only when no two adjacent MIX-TOKENs are TOKENs. Thus adjacent WORDS or SIGNS in an ID have to be separated by brackets or PLACEs.

Moreover, when an ID contains a TOKEN immediately followed by ‘[ID]’ or ‘[ID,...,ID]’, any further MIX-TOKENs in the same sequence of MIX-TOKENs must all be PLACEs. This ensures that a list of identifiers used to indicate a compound identifier can only be attached to the last token in an ID.

4

Lexical Symbols

This chapter describes the lexical symbols of Casl. Section 4.1 lists the key words and signs, Sect. 4.2 specifies the tokens used to form identifiers, and Sect. 4.3 describes the form of literal symbols for quoted characters, strings, and numbers. Finally, Sect. 4.4 gives a grammar for the simple URLs and paths used to identify libraries. The meta-notation used for grammars in this chapter is the same as in Chap. 3. The description of comments and annotations is deferred to Chap. 5.

Spaces and other layout characters terminate lexical symbols, and are otherwise ignored, except in quoted characters and strings. The next lexical symbol recognized at each stage is as long as possible. The lexical syntax of Casl forms a regular language, and a lexical analyzer for Casl can be generated using ML-lex from a grammar given on the accompanying CD-ROM. Note that when a library name is expected, a different lexical analysis is required, see Sect. 4.4. Some Casl parsers are scannerless [8], which facilitates context-dependent analysis of lexical symbols.

The character set for Casl specifications is ISO Latin-1. However, specifications can always be input in the ASCII subset.

For enhanced readability of specifications, each lexical symbol has a display format for use with graphic screens and printers, involving various type styles (upright, italic, boldface, and small capitals) as well as common mathematical signs. (No restrictions are imposed concerning which font families are to be used for displaying Casl specifications.) The display format for particular identifiers can be determined by means of display annotations, as explained in Sect. 5.2.2. The input syntax of lexical symbols is easy to relate to their display format, and also sufficiently readable for use in (plain-text) e-mail messages. A LaTeX package implementing the display format is available [50].

4.1 Key Words and Signs

The lexical symbols of Casl include various key words and signs that occur as terminal symbols in the context-free grammar in Chap. 3. Key words and signs that represent mathematical signs are displayed as such, when possible, and those signs that are available in the ISO Latin-1 character set may also be used for input.

4.1.1 Key Words

Key words are always written lowercase. The following key words are reserved, and are not available for use as complete identifiers (nor as complete mixfix tokens in mixfix identifiers) although they can be used as parts of tokens:

```
and arch as axiom axioms closed def else end exists
false fit forall free from generated get given hide
if in lambda library local not op ops pred preds
result reveal sort sorts spec then to true type types
unit units var vars version view when with within
```

The following key words are *not* reserved:

```
assoc comm idem
```

4.1.2 Key Signs

The following key signs are reserved, and are not available for use as complete identifiers (nor as mixfix tokens in mixfix identifiers):

```
:  :?  ::=  =  =>  <=>  ¬  .  ·  |  |->  \/  /\
```

The ISO Latin-1 characters '¬' and '·' are equivalent as key signs to the ASCII characters '`not`' and '`.`', respectively. The following key signs are *not* reserved:

```
<  *  ×  ->  ?  !
```

The ISO Latin-1 key character '×' is equivalent as a key sign to the ASCII character '`*`'.

4.1.3 Display Format

The following key words represent mathematical signs, and are displayed accordingly when possible, as indicated below:

`forall`	`exists`	`not`	`in`	`lambda`
$\forall$	$\exists$	$\neg$	$\in$	λ

The following key words are displayed in the same (italic) font as identifiers when they occur in formulas, in attributes, or in alternatives of datatype declarations:

```
true false not if when else assoc comm idem unit
op ops pred preds sort sorts var vars
```

Otherwise, key words are always displayed in a boldface font.

The following key signs represent mathematical signs, and are displayed accordingly when possible, as indicated below:

`->`	`=>`	`<=>`	`=e=`	`.`	`·`	`|->`	`/\`	`\/`
$\rightarrow$	$\Rightarrow$	$\Leftrightarrow$	$\overset{e}{=}$	$\bullet$	$\bullet$	$\mapsto$	$\wedge$	$\vee$

4.2 Tokens

This section defines the tokens used to form identifiers: words, signs, single digits, and quoted characters. Words are essentially alphanumeric sequences, allowing also some further characters. Signs are sequences of mathematical and punctuation characters.

4.2.1 Words

The lexical grammar for the tokens WORDS, DOT-WORDS, and DIGIT is as follows:

```
WORDS     ::= WORD _ ... _ WORD

DOT-WORDS ::= "." WORDS

WORD      ::= WORD-CHAR ... WORD-CHAR

WORD-CHAR ::= LETTER | "’" | DIGIT

LETTER    ::= A | B | C | D | E | F | G | H | I | J | K | L | M
            | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
            | a | b | c | d | e | f | g | h | i | j | k | l | m
            | n | o | p | q | r | s | t | u | v | w | x | y | z
            | À | Á | Â | Ã | Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì
            | Í | Î | Ï | Ð | Ñ | Ò | Ó | Ô | Õ | Ö | Ø | Ù | Ú
            | Û | Ü | Ý | Þ | ß | à | á | â | ã | ä | å | æ | ç
            | è | é | ê | ë | ì | í | î | ï | ð | ñ | ò | ó | ô
            | õ | ö | ø | ù | ú | û | ü | ý | þ | ÿ

DIGIT     ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

A WORDS must start with a LETTER, and must not be one of the reserved key words used in the context-free syntax in Sect. 4.1.1. Note that LETTER includes all the ISO Latin-1 national and accented letters.

4.2.2 Signs

The lexical grammar for the token SIGNS is as follows:

```
SIGNS ::= SIGN ... SIGN

SIGN  ::= + | - | * | / | \ | & | = | < | >
        | ! | ? | : | . | $ | @ | # | ^ | ~
        | ¡ | ¿ | × | ÷ | £ | © | ± | ¶ | §
        | ¹ | ² | ³ | · | ¢ | ° | ¬ | µ | "|"
```

A SIGNS must not be one of the reserved signs:

```
:  :?  ::=  =  =>  <=>  ¬  .  ·  |  |->  \/  /\
```

These sequences of characters may however be used together with other characters in a `SIGNS`. For example, ‘`==`’, ‘`:=`’, and ‘`||`’ are each recognized as a complete `SIGNS`. Note that identifiers that start or finish with a `SIGNS` need to be separated by (e.g.) a space from adjacent reserved signs: a sequence of characters such as ‘ `#:` ’ is always recognized as a single symbol, whereas ‘ `# :` ’ is recognized as two symbols.

A single character ‘`<`’, ‘`*`’, ‘×’, ‘`?`’, or ‘`!`’ is also recognized as a complete `SIGNS`, despite its use as a key sign as described in Sect. 4.1.2.

Note that `SIGN` does not include the following ASCII signs:

```
( ) [ ] { } ; , ‘ " %
```

nor the ISO Latin-1 signs for general currency, yen, broken vertical bar, registered trade mark, masculine and feminine ordinals, left and right angle quotes, fractions, soft hyphen, acute accent, cedilla, macron, and umlaut.

4.2.3 Quoted Characters

The lexical grammar for the token `QUOTED-CHAR` is as follows (where ‘`|...|`’ indicates evident alternatives that are omitted here for brevity):

```
QUOTED-CHAR ::= "’" CHAR "’"

CHAR        ::= LETTER | DIGIT | SIGN
              | ; | , | ‘ | % | _ | " "
              | ( | ) | [ | ] | { | }
              | \n | \t | \r | \v | \b | \f
              | \a | \? | ’\"’ | "\’" | \\
              | \000  | ... | \255
              | \x00  | ... | \xFF
              | \o000 | ... | \o377
```

4.3 Literal Strings and Numbers

Casl provides literal symbols for quoted strings and numbers. Their interpretation can be determined using annotations, as explained in Sect. 5.2.4. (Casl has no built-in datatypes, so literal symbols cannot have a default interpretation.)

In contrast to the tokens described in Sect. 4.2, literal symbols abbreviate terms, and cannot be used as identifiers. The lexical grammar of the symbols `STRING`, `DIGITS`, `NUMBER`, `FRACTION`, and `FLOATING` is a follows:

```
STRING   ::= ’"’ ’"’ | ’"’ CHAR ... CHAR ’"’

DIGITS   ::= DIGIT DIGIT ... DIGIT

NUMBER   ::= DIGIT  |  DIGITS
```

```
FRACTION ::= NUMBER . NUMBER

FLOATING ::= NUMBER "E" OPT-SIGN NUMBER
           | FRACTION "E" OPT-SIGN NUMBER

OPT-SIGN ::= + | - | EMPTY
```

NUMBER is recognized as a lexical symbol only where a VERSION-NUMBER is expected in specification libraries; elsewhere, a single digit is recognized as a DIGIT, and a sequence of two or more digits as a DIGITS (since only the former can be used as an identifier).

4.4 URLs and Paths

URL and PATH are recognized as lexical symbols only directly following the key words '`library`' and '`from`' in specification libraries. The following grammar provides a minimal syntax for URL: further forms may be recognized and supported.

```
PATH-CHAR ::= A |...| Z | a |...| z | 0 |...| 9
            | $ | - | _ | @ | . | & | + | ! | *
            | '"' | "'" | ( | ) | , | : | ~
            | % HEX-CHAR HEX-CHAR
HEX-CHAR  ::= A |...| F | a |...| f | 0 |...| 9

PATH-WORD ::= PATH-CHAR ... PATH-CHAR
PATH      ::= PATH-WORD /.../ PATH-WORD
URL       ::= http:// PATH
            | ftp:// PATH
            | file:/// PATH
```

5

Comments and Annotations

This chapter starts with a description of the common features of comments and annotations. Section 5.1 explains how comments are written. Section 5.2 covers various kinds of annotations, including those used to provide literal syntax for numbers, strings, and lists. The meta-notation used for grammars in this chapter is the same as in Chap. 3.

Comments and annotations can be used to provide auxiliary information that gets attached to the nodes of abstract syntax trees of Casl specifications during parsing. They do not affect the semantics of specifications, but may have significance for tool support. Comments may also be used to ignore parts of specifications (so-called 'commenting-out').

The general form of comments and annotations is similar:

- they start with a percent character '`%`';
- their extent can be indicated by grouping brackets, terminated by '`%`';
- they have abbreviated forms for use at the end of the line (except for label annotations); and
- they cannot be nested (except for commenting-outs, which are treated as blank space).

Ordinary comments and annotations – collectively referred to here simply as annotations – can get attached to the following sorts of nodes in abstract syntax trees:

- a `SORT-ITEM`, `OP-ITEM`, `PRED-ITEM`, or `ALTERNATIVE`;
- a complete `FORMULA` (i.e., not part of a larger `FORMULA` or `TERM`) in an `AXIOM`, `PRED-DEFN`, or `SUBSORT-DEFN`;
- a complete `TERM` (i.e., not part of a larger `TERM`) in an `OP-DEFN`;
- a `DATATYPE-DECL` or `SIG-ITEMS` in a `SORT-GEN` or `BASIC-ITEMS`;
- a `SPEC`, `FIT-ARG`, `ARCH-SPEC`, `UNIT-DECL-DEFN`, `UNIT-TERM`, `LIB-ITEM`, or `LIB-DEFN`.

Such nodes can be formed by parsing text as the corresponding nonterminal symbols of the concrete syntax grammar given in Sect. 3.1 (they can also be constructed by other tools). During parsing, the recognition of each of the above nodes may collect both *preceding* and *trailing* annotations. At first, a (possibly-empty) sequence of preceding annotations is collected. During the recognition of a node, any annotations not collected by inner nodes are kept as trailing annotations of the outer node. Finally, after the recognition of the outer node is complete, any further trailing annotations are collected – as long as they are on consecutive lines, i.e., without a blank line. (Commented-out text is treated as blank space, so it may give the effect of a blank line.) After a blank line, preceding annotations for the next node that collects annotations may appear. (Final annotations of a library `LIB-DEFN` may extend to the end of the file.)

Thus annotations may occur anywhere between lexical tokens, but are only attached to the above sorts of nodes. The interpretation of some annotations is further explained in Sect. 5.2.

The nonterminal symbols `TEXT` and `TEXT-LINES` are used only for specifying the form of comments and annotations, and are not themselves regarded as lexical symbols:

```
TEXT          ::= NOT-NEWLINE ... NOT-NEWLINE | EMPTY
TEXT-LINE     ::= TEXT NEWLINE
TEXT-LINES    ::= TEXT | TEXT-LINE TEXT-LINES
```

`NEWLINE` denotes the character that indicates the start of a new line; `NOT-NEWLINE` denotes all the other printable ISO Latin-1 characters, together with the space and tab characters.

5.1 Comments

```
COMMENT       ::= COMMENT-LINE | COMMENT-GROUP | COMMENT-OUT
COMMENT-LINE  ::= %% TEXT-LINE
COMMENT-GROUP ::= %{ TEXT-LINES }%
COMMENT-OUT   ::= %[ TEXT-LINES ]%
```

A single-line comment of the form '`%%`*text newline*' is equivalent to the grouped comment '`%{`*text*`}%`'; the latter form also allows multi-line comments. Arbitrary *text* is 'commented out' by writing '`%[`*text*`]%`'; commenting-out may be nested.

Comments are generally displayed with the body in the same font as ordinary informal text that might appear before and after a Casl specification. However, this may be overruled by explicit formatting instructions in the text of the comment. The preferred formatting of a part of a comment by different formatters is indicated using the following syntax (which is similar to that of display annotations, see Sect. 5.2.2):

`%display` *text* `%HTML ... %LATEX ... %RTF ...`

at the end of a line, or, possibly over several lines:

```
%display( text %HTML ... %LATEX ... %RTF ... )%
```

Both the above indicate that the *text* is to be displayed according to the formatting instructions given for HTML, LaTeX, and RTF (which may be listed in any order, or omitted). Formatters for which there are no instructions should display the *text* exactly as input, preserving the line breaks of multi-line comments.

If available, a smaller base font than normal may be used when displaying comments. The delimiters of comments are always to be displayed in boldface.

CASL specification text within comments should be delimited by a bracketed group[1] of the form '`%CASL(...)%`', to allow its appropriate display. The kind of CASL construct may be indicated by using a nonterminal symbol from the CASL abstract syntax (such as '`ID`' or '`TERM`', see Chap. 2) instead of '`CASL`'.

5.2 Annotations

```
ANNOTATION        ::= ANNOTATION-LINE | ANNOTATION-GROUP | LABEL
ANNOTATION-LINE   ::= %WORDS TEXT-LINE
ANNOTATION-GROUP  ::= %WORDS( TEXT-LINES )%
LABEL             ::= %( TEXT-LINE )%
```

In `ANNOTATION-LINE` and `ANNOTATION-GROUP`, spaces are not allowed before the `WORDS`, and an opening bracket '`(`' directly following the `WORDS` distinguishes an `ANNOTATION-GROUP` from an `ANNOTATION-LINE`. (The lexical token `WORDS` is defined in Sect. 4.2.1.) A single-line annotation of the form '`%`*words text newline*' is equivalent to '`%`*words*`(`*text*`)%`'.

Annotations at the beginning of a library (between the `LIB-NAME` and the first `LIB-ITEM`) apply globally to all its `LIB-ITEM`s, and to all libraries that download any of those items. *Conflicting* annotations that arise due to downloading from different remote libraries are simply ignored, whereas local annotations override conflicting annotations from remote libraries. Conflicting annotations within the same library are ignored as well.

Each kind of annotation imposes restrictions on the syntax of its text. (It is envisaged that further kinds of annotations will be added later, but only with the same general form as indicated above.)

Unless otherwise indicated below, annotations '`%`*words*...' are to be displayed in a smaller font than usual, when possible, and with the delimiters in boldface. CASL symbols in the body of the annotation are to be shown in their display format. Tools may suppress the display of particular kinds of annotations.

[1] The delimiters for CASL specification text in comments are similar to those used for multi-line annotations, see Sect. 5.2.

5.2.1 Label Annotations

A label annotation is written '`%(`*text-lines*`)%`', where *text-lines* is the label itself. For instance, in '`%(reverse-NeList)%`' the label is '`reverse-NeList`'.

A label annotation is normally attached to a complete `FORMULA`, although other constructs within a specification may be labelled as well. A label on an axiom is to be displayed flush with the right margin of the enclosing list of axioms, with the *text-lines* in the same font as used for text in comments.

Labels are used by tools to reference parts of specifications, and should therefore be unique at least within the same `LIB-ITEM`.

5.2.2 Display Annotations

A single-line display annotation `ANNOTATION-LINE` is written:

```
%display id %HTML ... %LATEX ... %RTF ...
```

It indicates that the identifier with input syntax *id* is to be displayed according to the formatting instructions given for HTML,

LaTeX, and RTF (which may be listed in any order, or omitted). When there are no instructions given for the language of the formatter being used, the identifier is displayed as its input syntax.

The following example indicates that the identifier input as '`div`' should be displayed as '÷' by formatters that understand HTML or LaTeX commands:

```
%display div %HTML &divide; %LATEX \div
```

Display annotations generalize to formatting mixfix notation by interpreting the place-holder '`__`' as such in the formatting instructions, e.g.:

```
%display( sum__to__
%HTML  SUM<sub>__<sup>__
%LATEX \sum_{__}^{__}
)%
```

The HTML level is assumed to be 4.0; the version of LaTeX is assumed to be LaTeX2e, using the Casl package [50], in math mode.

Display annotations may occur only at the beginning of libraries (between the `LIB-NAME` and the first `LIB-ITEM`), and apply globally. Display annotations for the same identifier are regarded as conflicting unless their formatting instructions are identical or complementary, up to reordering. For displaying the annotation itself, only the input syntax and the display relevant to the formatter being used are to be shown.

5.2.3 Parsing Annotations

These annotations are to allow users to specify the precedence and associativity of operation symbols. Their primary purpose is to allow the omission of grouping parentheses in the input; but formatters may also exploit them to avoid superfluous parentheses in the display. Parsing annotations may occur only at the beginning of libraries (between the `LIB-NAME` and the first `LIB-ITEM`), and apply globally.

Precedence

A single-line precedence annotation `ANNOTATION-LINE` is written:

`%prec {`id_1`, ... , `id_n`} < {`id_{n+1}`, ..., `id_{n+k}`}`

Each id_i is a mixfix identifier of the form ‘`__...__...__`’. The relation `{`id_1`}<` `{`id_2`}` specifies that the symbol id_1 has lower priority (i.e., binds weaker) than the symbol id_2. It is also possible to specify that mixfix identifiers (of any form) are *not* allowed to be combined without explicit grouping parentheses. This is done using ‘`<>`’ instead of ‘`<`’. In both cases, a precedence annotation involving groups of identifiers abbreviates the collection of corresponding precedence annotations between each pair of identifiers from the two groups.

Two different precedence annotations for the same pair of identifiers are regarded as conflicting.

The precedence annotations determine a pre-order, which is obtained in the following way:

1. Expand all precedence relations into binary relations:
   - from annotations of the form ‘`%prec {`id_1`} < {`id_2`}`’ we get $\{(id_1, id_2)\}$, and
   - from annotations of the form ‘`%prec {`id_1`} <> {`id_2`}`’ we get $\{(id_1, id_2), (id_2, id_1)\}$.
2. Take the union of all the expanded precedence relations thus obtained with the predefined precedences listed in Sect. 3.2.
3. Take the reflexive transitive closure of this union.

If two symbols occurring in a term or atomic formula are equivalent (i.e. related in both directions) or incomparable (i.e. related in no direction) in the precedence relation, their grouping has to be explicitly specified by using parentheses.

Associativity

A single-line left-associativity annotation `ANNOTATION-LINE` is written:

`%left_assoc `id_1`, ..., `id_n

The id_i must be infix operation symbols. Similarly for right-associativity annotations.

An associativity annotation involving a group of identifiers abbreviates the collection of corresponding associativity annotations for each identifier in the group. Left and right associativity annotations for the same identifier are regarded as conflicting.

For example, declaring '__+__' to be left associative means that t_1+t_2+t_3 is parsed as (t_1+t_2)+t_3 while declaring it to be right associative leads to t_1+(t_2+t_3). If there is no associativity annotation for an infix symbol, it is not allowed to repeat that symbol without explicit grouping using parentheses.

An associativity *attribute* 'assoc' for an operation (see Sect. 3.2) has the effect of an implicit associativity annotation. In contrast to explicit associativity annotations, such an implicit associativity annotation is *local*, and only influences parsing of the surrounding specification (more precisely, only the items after the 'assoc' attribute) plus any specification importing the specification containing the attribute. If the operation with the assoc attribute is renamed, the local parsing annotation applies to the new name instead of the old one.

5.2.4 Literal Annotations

In this section, several annotations for operations are introduced that can be used to interpret the literal syntax for numbers and strings in Casl (see Sect. 4.3), and provide a literal syntax for lists. Literal annotations may occur only at the beginning of libraries (between the LIB-NAME and the first LIB-ITEM), and apply globally.

Literal Syntax for Numbers

The annotation for declaring an operation to be used for concatenation of digits within a number is written '%number f'.

The annotation has the effect that a DIGITS of the form $d_1 \ldots d_n$ (where $n > 1$ and each d_i is a DIGIT) is translated to the (abstract syntax of) the term $f(f(\ldots f(t_1, t_2) \ldots, t_{n-1}), t_n)$, where t_i is the abstract syntax tree for d_i. For example, to interpret DIGITS as decimal notation in connection with a particular datatype of integers, f would be a binary operation which, when applied to x and y, returns $10x + y$, and the decimal digits would be defined as the expected numbers from 0 to 9.

Vice versa, an abstract syntax tree corresponding to a term of the above form which is maximal (i.e., it is not a sub-term of a larger term of the same form) is expected to be printed as $d_1 \ldots d_n$.

Different '%number' annotations are regarded as conflicting. If there is no '%number' annotation, then a DIGITS is not recognized as a well-formed LITERAL.

The annotation for declaring the operations used for evaluating the decimal point and the exponentiation 'E' within FRACTION or a FLOATING is written '%floating f, g'.

The annotation has the effect that a FRACTION of the form $n_1.n_2$ (where each n_i is a NUMBER) is translated to the (abstract syntax of) the term $f(t_1, t_2)$, where t_i is the abstract syntax tree for n_i, $i = 1, 2$.

Similarly, a FLOATING of the form 'n_1En_2' (where n_1 is a NUMBER or a FRACTION and n_2 is of form OPT-SIGN NUMBER) is translated to the (abstract syntax of) the term $g(t_1, t_2)$, where t_i is the abstract syntax tree for n_i, $i = 1, 2$.

Vice versa, an abstract syntax tree corresponding to a term of one of the above forms which is maximal (i.e., it is not a sub-term of a larger term of the same form) is expected to be printed as $n_1.n_2$ or $n_1 E n_2$, respectively.

Different '%floating' annotations are regarded as conflicting. If there is no '%floating' annotation, then neither a FRACTION nor a FLOATING is recognized as a well-formed LITERAL.

Literal Syntax for Strings

The annotation for declaring operations for the empty string and for concatenation of a character with a string is written '%string c, f'.

The annotation has the effect that a STRING of the form '"c_1 ... c_n"' (where $n \geq 0$ and each c_i is a CHAR) is translated to the (abstract syntax of) the term $f(t_1, f(t_2, \ldots f(t_n, c) \ldots))$, where t_i is the abstract syntax tree for the QUOTED-CHAR ''c_i'', or simply to c when $n = 0$.

Vice versa, an abstract syntax tree corresponding to a term of the above form which is maximal (i.e., it is not a sub-term of a larger term of the same form) is expected to be printed as '"c_1 ... c_n"'.

Different '%string' annotations are regarded as conflicting. If there is no '%string' annotation, then a STRING is not recognized as a well-formed LITERAL.

Literal Syntax for Lists

The annotation for declaring a macro for applying a binary function on a list of arguments is written '%list b_1__b_2, c, f'. The symbol 'b_1__b_2' is a mixfix identifier with a single place-holder, where b_1 at least contains an open bracket ('[' or '{') that must be matched by b_2. This annotation can in particular be used to introduce a syntax for lists, e.g., '%list [__], nil, cons' allows the use of the notation '[x_1, ..., x_n]' for lists constructed using cons, starting from the empty list nil.

A list of the form 'b_1 t_1, ..., t_n b_2' (where $n \geq 0$ and each t_i is a TERM) is translated to the (abstract syntax of) the term $f(u_1, f(u_2, \ldots f(u_n, c) \ldots))$, where u_i is the abstract syntax tree for t_i, or simply to c when $n = 0$.

Vice versa, an abstract syntax tree corresponding to a term of the above form which is maximal (i.e., it is not a sub-term of a larger term of the same form) is expected to be printed as 'b_1 t_1, ..., t_n b_2'.

Different '%list' annotations are regarded as conflicting when their mixfix identifiers 'b_1__b_2' are identical.

5.2.5 Semantic Annotations

These annotations are used to express known (or presumed) features of the semantics of the specification, e.g., that an extension is 'conservative', or that certain formulas are consequences of the specification. Theorem-proving tools may interpret these annotations as *proof obligations*. Note, however, that the annotations do *not* affect the semantics of a specification, regardless of whether the specification has the indicated features or not.

Implied Axioms

The annotation for an implied axiom is written '`%implied`' (at the end of a line).

In the context of basic specifications the annotation `%implied` is used to characterize implicit or explicit axioms as logical consequences of the enclosing basic specification. `%implied` may annotate a `SORT-ITEM`, `OP-ITEM`, `PRED-ITEM`, `AXIOM`, `ALTERNATIVE`, `DATATYPE-DECL`, or a `BASIC-ITEMS` consisting of a `FREE-DATATYPE` or `SORT-GEN`.

Within a *basic specification SP* the annotations `%implied` hold if the annotation `%implies` holds for the following *structured specification*:

SP_1 **then** `%implies` SP_2

Here, SP_1 consists of two parts: the first declares the whole signature of SP, the second is SP without the items marked with `%implied`. SP_2 is as SP without the non-`%implied` items, except that global variable declarations and parsing annotations (possibly arising from operation attributes) are kept.

Extension Annotations

All the remaining semantic annotations precede a specification SP' that follows either:

- a '`then`' keyword within an extension – in which case, let SP be the part of the extension just up to, but excluding this occurrence of '`then`'; or
- the equals sign within a `SPEC-DEFN` – in which case, let SP be the union of the imports, extended by the union of the parameters.

Different semantic annotations at the same position are regarded as complementary.

Conservative Extension

The annotation for conservative extension is written '`%cons`'. It expresses that SP' is a conservative extension of SP, i.e. each SP-model can be expanded to an (SP **then** SP')-model.

Note that a model M' is an expansion of a model M iff M is a reduct of M'.

Monomorphic Extension

The annotation for monomorphic extension is written '`%mono`'. It expresses that SP' is a monomorphic extension of SP, i.e. each model of SP can be expanded to a model of (SP **then** SP') that is unique up to isomorphism.

Note that '`%mono`' is strictly stronger than the '`%cons`' annotation.

Definitional Extension

The annotation for definitional extension is written '`%def`'. It expresses that SP' is a definitional extension of SP, i.e. each model of SP can be uniquely expanded to a model of (SP **then** SP') (this implies a bijective correspondence between the two model classes).

Note that '`%def`' is strictly stronger than the '`%mono`' annotation.

Implied Extension

The annotation for implied extension is written '`%implies`'. The annotation '`%implies`' is well-formed iff the signature of (SP **then** SP') is the signature of SP. A well-formed '`%implies`' annotation holds iff the model class of (SP **then** SP') is the model class of SP.

Note that '`%implies`' is strictly stronger than the '`%def`' annotation.

5.2.6 Miscellaneous Annotations

The annotations described in this section apply either to whole libraries or to the single library items that follow them.

Authors

An authors annotation is written:

`%authors` $name_1$ `<`$email_1$`>, ...,` $name_n$ `<`$email_n$`>`

at the end of a line, or, possibly over several lines:

`%authors(` $name_1$ `<`$email_1$`>, ...,` $name_n$ `<`$email_n$`> )%`

It indicates the authors of the annotated construct. When a library item has no authors annotation, its authors are assumed to be the same as those of the enclosing library. The order of listing the authors is not constrained, and the listing of e-mail addresses is optional.

Date

A date annotation is written:

`%date` $date_1$`, ...,` $date_n$

at the end of a line, or, possibly over several lines:

`%date(` $date_1$`, ...,` $date_n$ `)%`

It indicates the latest modification date of the annotated construct. Any additional dates indicate some previous modification dates (possibly including the creation date). The order of listing the dates may be either increasing or decreasing. The format of the dates should be uniform and unambiguous.

Part III

CASL Semantics

Hubert Baumeister
Maura Cerioli
Anne Haxthausen
Till Mossakowski
Peter D. Mosses
Donald Sannella
Andrzej Tarlecki

Editors: Donald Sannella and Andrzej Tarlecki

P.D. Mosses (Ed.): CASL Reference Manual, LNCS 2960, pp. 113–271, 2004.

1 Introduction

This part of the CASL Reference Manual defines the formal semantics of the language CASL, as informally presented in the CASL Summary (Part I). Apart from this Introduction, which is partly devoted to defining some basic notation and explaining the style of the semantics, the structure of this document is deliberately almost identical to the structure of the CASL Summary to aid cross-reference. As in the CASL Summary, Chap. 2 deals with *many-sorted basic specifications*, and Chap. 3 extends this by adding features for *subsorted basic specifications*. Chapter 4 provides *structured specifications*, together with *specification definitions*, *instantiations*, and *views*. Chapter 5 summarizes *architectural and unit specifications*, which, in contrast to structured specifications, prescribe the separate development of composable, reusable implementation units. Finally, Chap. 6 considers *specification libraries*. There are two exceptions to the structural match between this document and the CASL Summary. One is in Chap. 4, where the subsections of Sect. 4.1 define many concepts and notations underlying the semantics of structured specifications that were not mentioned in the CASL Summary. The other is in Chap. 5, where Sect. 5.6 presents a more precise analysis of the constructs considered in Sects. 5.2–5.5.

The first section of each chapter defines the *semantic concepts* underlying the kind of specification concerned, with the remaining sections presenting the abstract syntax of the associated CASL *language constructs* and defining their semantics. The abstract syntax is identical to that given in the CASL Summary; it is repeated here for ease of reference.

Brief informal summaries of the main concepts and constructs precede each block of formal definitions. This material, which is in boxes (like this paragraph) is provided as a supplement to the formal material; since it deliberately glosses over the details, it should *not* be regarded as definitive. There is other informal explanatory text in between the definitions, but nothing that is likely to be mistaken for a definition.

1.1 Notation

This section summarizes some of the basic notation used in the definitions below.

Sets

$Set(A)$ is the set of all subsets of A, and $FinSet(A)$ is the set of finite subsets of A. If A is a set then $|A|$ is the cardinality of A. *unit* denotes the singleton set $\{*\}$.

Tuples

$A_1 \times \cdots \times A_n$ is the set of n-tuples with jth component from A_j. Tuples are written like this: $(a_1, \ldots, a_n)$. Sometimes the parentheses are omitted, especially when tuples are used as subscripts or superscripts.

Sequences

$FinSeq(A)$ is the set of finite sequences of elements from A. Sequences are written like this: $\langle a_1, \ldots, a_n \rangle$, where $n \geq 0$. (This notation is different from that used in the CASL Summary and the abstract syntax, where $FinSeq(A)$ is written A^* and $\langle a_1, \ldots, a_n \rangle$ is written $a_1 \ldots a_n$.) If $w = \langle a_1, \ldots, a_n \rangle$ then $|w| = n$.

Functions

$A \rightharpoonup B$ is the set of partial functions from A to B. $Dom(f) \subseteq A$ is the domain of $f : A \rightharpoonup B$. $A \to B$ is the set of total functions from A to B. Any total function $f : A \to B$ can also be regarded as a partial function $f : A' \rightharpoonup B$ for any $A' \supseteq A$, and any partial function $f : A \rightharpoonup B$ is a total function $f : Dom(f) \to B$. Functions are written like this: $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$, where $n \geq 0$, or $\{x \mapsto x + 3 \mid x \in Nat\}$. We use the notation $f(x)$ for application of a function f to an argument x. Sometimes the parentheses are omitted, for instance when x is a tuple or a sequence. The graph of a function $f : A \rightharpoonup B$ is the set of pairs $graph(f) = \{(x, f(x)) \mid x \in Dom(f)\}$ and the kernel of f is $ker(f) = \{(x, y) \mid x, y \in Dom(f) \text{ and } f(x) = f(y)\}$. When f is an indexed family (a function from an index set to a domain of elements) we write f_x instead of $f(x)$. $A \overset{\text{fin}}{\to} B$ is the set of finite maps (i.e. partial functions with finite domain) from A to B.

Union and $\emptyset$

We use union ($\cup$) to combine semantic objects of various kinds, with the evident interpretation (e.g. component-wise union for tuples and point-wise union for functions, that is $(f \cup g)(x) = f(x) \cup g(x)$ if f and g are set-valued functions such that $Dom(f) = Dom(g)$). More generally, for any set-valued functions f and g we take

$$(f \cup g)(x) = \begin{cases} f(x) \cup g(x) & \text{if } x \in Dom(f) \cap Dom(g) \\ f(x) & \text{if } x \in Dom(f) \setminus Dom(g) \\ g(x) & \text{if } x \in Dom(g) \setminus Dom(f) \\ \text{undefined} & \text{otherwise} \end{cases}$$

which gives $Dom(f \cup g) = Dom(f) \cup Dom(g)$. Similarly, $\emptyset$ is used for the empty object of various kinds (e.g. empty signature, empty function). If $n = 0$ then $A_1 \cup \cdots \cup A_n$ denotes $\emptyset$.

Disjoint union

$A \uplus B$ is the disjoint union of A and B. Injection from A and B to $A \uplus B$ is implicit.

Given a union of syntactic categories, as in

```
OP-TYPE          ::=  TOTAL-OP-TYPE | PARTIAL-OP-TYPE
```

we distinguish between $a \in$ `TOTAL-OP-TYPE` (resp. $a \in$ `PARTIAL-OP-TYPE`) and $a \in$ `OP-TYPE` by writing the latter as 'a qua `OP-TYPE`'. The syntactic categories in question are always disjoint, with different constructors being used to form their phrases (in this case, the constructors are `total-op-type` and `partial-op-type`).

Function completion

We sometimes need to 'complete' a function f with $Dom(f) \subseteq S$ to give a function with domain S by mapping values in $S \setminus Dom(f)$ to an appropriate neutral value. In particular, if f is a set-valued function, we define $complete(f, S) = f \cup \{x \mapsto \emptyset \mid x \in S \setminus Dom(f)\}$.

Categories

Some elementary category theory is used in places. A suitable introduction is [52]. The category of sets is denoted **Set** and the (quasi)category of categories is denoted **CAT** [1]. We use the notation $f \circ g$ for (applicative order) composition of morphisms in a category. In **Set** this gives $(f \circ g)(x) = f(g(x))$. The class of objects of a category $\mathcal{C}$ is written $|\mathcal{C}|$ and the identity morphism on A is written id_A.

[1] There are foundational problems connected with the use of **CAT** – see [27] for how to solve them.

Semantic domains

We define various semantic domains below. By convention, semantic domains containing 'syntactic' objects (e.g. *Signature*) are in italics and semantic domains containing 'semantic' objects (e.g. **Model**) are in boldface. Here is an example of a domain of 'syntactic' objects:

$$(w, s) \text{ or } ws \in \mathit{FunProfile} = \mathit{FinSeq}(\mathit{Sort}) \times \mathit{Sort}$$

This defines the set *FunProfile* as the set of pairs having finite sequences of elements from *Sort* as first component and elements of *Sort* as second component. The metavariable *ws* ranges over elements of *FunProfile*. When we need to refer to the components of the pair we use the notation (w, s) instead, so w ranges over elements of $\mathit{FinSeq}(\mathit{Sort})$ and s ranges over elements of *Sort*.

Validity

Typically, semantic domains are constructed from 'more basic' domains together with some well-formedness requirements. Then a *valid* object is a value in the given set that satisfies the given requirements. Here is an example:

$$X \in \mathit{Variables} = \mathit{Sort} \overset{\text{fin}}{\rightarrow} \mathit{FinSet}(\mathit{Var})$$

Requirements on an S-sorted set of variables X:

- $\mathit{Dom}(X) = S$
- for all $s, s' \in S$ such that $s \neq s'$, $X_s \cap X_{s'} = \emptyset$.

This says that a 'set of variables' is a finite map taking elements of *Sort* to finite subsets of *Var*, while a 'valid S-sorted set of variables' is a finite map of this kind that satisfies the two requirements given. Often, as in this case, validity of an object is relative to some other (valid) object, here a set S of sorts. We will tacitly require all objects that arise in defining the semantics of CASL phrases to be valid.

Abstract syntax

For an introduction to the form of grammar used to define the abstract syntax of language constructs, see Chap. II:2, which also contains the abstract syntax of the entire CASL specification language.

1.2 Static Semantics and Model Semantics

The semantics of language constructs is given in two parts. The *static semantics* checks well-formedness of phrases of the abstract syntax and produces a 'syntactic' object as result, failing to produce any result for ill-formed phrases.

For example, for a many-sorted basic specification (see Chap. 2) the static semantics yields an enrichment containing the sorts, function symbols, predicate symbols and axioms that belong to the specification. A judgement of the static semantics has the following form: *context* $\vdash$ *phrase* $\rhd$ *result*. The *model semantics* provides the corresponding model-theoretic part of the semantics, and is intended to be applied only to phrases that are well-formed according to the static semantics. For a basic specification, the model semantics yields a class of models. A judgement of the model semantics has the following form: *context* $\vdash$ *phrase* $\Rightarrow$ *result*.

A statically well-formed phrase may still be ill-formed according to the model semantics, and then no result is produced. This can never happen in the semantics of basic constructs but it can happen in the semantics of structured specifications and architectural specifications.

1.3 Semantic Rules

The judgements of the static semantics and model semantics are defined inductively by means of rules in the style of Natural Semantics [30]. For each phrase class we give a group of rules defining the semantics of the constructs in that class. The group is preceded by a specification of the 'type' of the judgement(s) being defined. This is followed by pre-conditions on the 'inputs' to the judgement(s) which, if satisfied, guarantee that the 'outputs' satisfy the given post-conditions. Each of the rules should ensure that this is the case. For example, here is the section of the semantics for the phrase class `AXIOM-ITEMS` from Sect. 2.5 below, for which there is just one rule.

$$\boxed{\Sigma, X \vdash \texttt{AXIOM-ITEMS} \rhd \Psi}$$

X is required to be a valid set of variables over the sorts of Σ. Ψ is a set of Σ-sentences.

$$\frac{\Sigma, X \vdash \texttt{AXIOM}_1 \rhd \psi_1 \qquad \cdots \qquad \Sigma, X \vdash \texttt{AXIOM}_n \rhd \psi_n}{\Sigma, X \vdash \texttt{axiom-items AXIOM}_1 \ \ldots\ \texttt{AXIOM}_n \rhd \{\psi_1, \ldots, \psi_n\}}$$

The 'type' of the judgement is $\Sigma, X \vdash \texttt{AXIOM-ITEMS} \rhd \Psi$. Intuitively, this says that in the local environment Σ with declared variables X, a phrase `AXIOM-ITEMS` yields a set Ψ of sentences. The pre-condition on the 'inputs' is the requirement that X be a valid set of variables over the sorts of Σ. (The requirement that Σ itself be valid is implicit – use of a metavariable always refers to a valid object of the relevant kind.) The post-condition on the 'output' is the assertion that Ψ will then be a set of Σ-sentences. It is easy to see that the given rule satisfies the pre/post-condition: if Σ, X satisfy the pre-condition then the post-condition associated with `AXIOM` guarantees that all of $\psi_1, \ldots, \psi_n$ will be Σ-sentences, and Ψ here is just $\{\psi_1, \ldots, \psi_n\}$.

Rules in the static semantics and model semantics have the form

$$\frac{\alpha_1 \qquad \cdots \qquad \alpha_n}{\beta}$$

where the conclusion β is a judgement and each premise α_j is either a judgement or a side-condition. When all the judgements occurring in all rules are positive (i.e. not negated) then the rules unambiguously define a family of relations via the usual notion of derivation tree, or equivalently as the smallest family of relations that is closed under the rules. Conclusions are always positive but there are situations in which negative premises are convenient. These are potentially problematic for at least two reasons: first, there may be *no* family of relations that is closed under the rules; second, there may be no *smallest* family of relations that is closed under the rules. It follows that care is required in situations where the natural choice of rules would involve negative premises. One way out is to simultaneously define a relation and its negation using rules with positive premises only, as in Sect. 2.1.4 below. Another is via the use of stratification to ensure the absence of dangerous circularities, cf. 'negation by failure' in logic programming [53], as in Sect. 2.5.3. See [19] for further discussion.

When a syntactic category C is defined as the disjoint union of other syntactic categories $C_1, \ldots, C_n$, rules that merely translate a judgement for C_1 etc. to a judgement for C are elided. Here is a schematic example of the kind of rules that are elided, for the static semantics:

$$\frac{\mathit{context} \vdash \mathit{phrase} \rhd \mathit{result}}{\mathit{context} \vdash \mathit{phrase}\ \mathrm{qua}\ C \rhd \mathit{result}}$$

Whenever such a rule is elided there will be a statement to this effect in the rule's place.

1.4 Institution Independence

CASL is the heart of a *family* of languages. *Sublanguages* of CASL are obtained by imposing syntactic or semantic restrictions, while *extensions* of CASL support various paradigms and applications.

The features of CASL for defining structured specifications, architectural specifications and specification libraries do not depend on the details of the features for basic specifications, so this part of the design is orthogonal to the rest. As a consequence, sublanguages and extensions of CASL can be defined by restricting or extending the language of basic specifications without the need to reconsider or change the rest of the language. On a semantic level, this is reflected by giving the semantics in an 'institution independent' style. The semantics of basic specifications with subsorts defines an *institution* [20] for CASL– actually, a variant of the notion of institution called an *institution with symbols* [39] – and the rest of the semantics is based on an arbitrary institution (with symbols). See Sect. 4.1 for more details.

Acknowledgement. The formal semantics of each part of the CASL Summary was written by one or more authors under the watchful gaze of a kibitzer[2]. The authors were responsible for actually doing the work, while the kibitzer was to serve as first reader, act as devil's advocate, push the authors to do the work, and perhaps jump in and help if needed. Authors and kibitzers were as follows:

Basic specifications: Don Sannella (kibitzer Hubert Baumeister)
Subsorted specifications: Maura Cerioli and Anne Haxthausen (kibitzer Till Mossakowski)
Structured specifications: Hubert Baumeister and Till Mossakowski (kibitzer Andrzej Tarlecki)
Architectural specifications: Andrzej Tarlecki (kibitzer Don Sannella)
Specification libraries: Peter Mosses (kibitzer Till Mossakowski)

This document was assembled by Don Sannella. The CoFI Semantics Group is coordinated by Andrzej Tarlecki.

Alexandre Zamulin read drafts of all parts of this document and sent many useful comments and suggestions, for which the authors are extremely grateful. Special thanks to Piotr Hoffman for pointing out inadequacies in an earlier version of the semantics of architectural specifications. Other useful suggestions were contributed by Christian Maeder, Markus Roggenbach, and Lutz Schröder.

This research was partly supported by CoFI-WG (ESPRIT Working Group 29432). Hubert Baumeister's work was partly supported by AGILE (FP5 project IST-2001-32747). Till Mossakowski's work was partly supported by the *Deutsche Forschungsgemeinschaft* under grant KR 1191/5-1. Don Sannella's work was partly supported by MRG (FP5 project IST-2001-33149). Andrzej Tarlecki's work was partly supported by KBN grant no. 7 T11C 002 21 and by AGILE.

[2] kibitzer, n. Meddlesome person, one who gives advice gratuitously; one who watches a game of cards from behind the players.

2

Basic Specification Semantics

A *basic specification* describes an extension Δ to the local environment Σ, together with a set of sentences over $\Sigma \cup \Delta$. This describes in turn the class of all models over $\Sigma \cup \Delta$ that satisfy those sentences.

To make this precise, Sect. 2.1 defines the underlying concepts, and the remaining sections cover the language constructs provided by Casl for use in such specifications, giving their abstract syntax and defining their interpretation. Consideration of the extra features concerned with subsorts is deferred to Chap. 3.

2.1 Basic Concepts

The concepts underlying basic specifications in Casl are those involved in defining an *institution* [20] for Casl. The following elements are required:

- a category **Sig** of *signatures* Σ, with *signature morphisms* $\sigma : \Sigma \to \Sigma'$;
- a (contravariant) functor $\mathbf{Mod} : \mathbf{Sig}^{op} \to \mathbf{CAT}$ giving for each signature Σ a category $\mathbf{Mod}(\Sigma)$ of *models* over Σ, with *homomorphisms* between them, and for each signature morphism $\sigma : \Sigma \to \Sigma'$ a *reduct* functor $\mathbf{Mod}(\sigma) : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ (usually written $.|_\sigma$) translating models and homomorphisms over Σ' to models and homomorphisms over Σ;
- a functor $\mathbf{Sen} : \mathbf{Sig} \to \mathbf{Set}$ giving for each signature Σ a set $\mathbf{Sen}(\Sigma)$ of *sentences* (or *axioms*) over Σ, and for each signature morphism $\sigma : \Sigma \to \Sigma'$ a *translation* function $\mathbf{Sen}(\sigma)$, usually written $\sigma(.)$, taking Σ-sentences to Σ'-sentences; and
- a relation $\models$ of *satisfaction* between models and sentences over the same signature.

Satisfaction is required to be compatible with reducts of models and translation of sentences: for all $\psi \in \mathbf{Sen}(\Sigma)$ and $M' \in \mathbf{Mod}(\Sigma')$,

$$M'|_\sigma \models \psi \iff M' \models \sigma(\psi).$$

(Additional structure is required for Chaps. 4 and 5, including a functor $|.| : \mathbf{Sig} \to \mathbf{Set}$ with certain properties which determines the set of *signature symbols* of any signature.)

The rest of this section defines the signatures, models, sentences, and satisfaction relation that underlie many-sorted basic specifications.

2.1.1 Signatures

A *many-sorted signature Σ* consists of: a set of *sorts*; separate families of sets of *total* and *partial function symbols*, indexed by *function profile* (a sequence of *argument sorts* and a *result sort* – *constants* are treated as functions with no arguments); and a family of sets of *predicate symbols*, indexed by *predicate profile* (a sequence of argument sorts). Constants and functions are also referred to as *operations*.

The internal structure of identifiers used to identify sorts, functions and predicates is insignificant for the semantics of basic specifications, see Sect. 2.6. Following the order of presentation in the Casl Summary, we therefore leave this unspecified for now, promising that there will be no circularity when the definitions of the sets *Sort*, *FunName* and *PredName* are eventually provided:

$$\begin{aligned} s &\in \mathit{Sort} \\ f &\in \mathit{FunName} \\ p &\in \mathit{PredName} \end{aligned}$$

(In Sect. 2.3 the internal structure of sorts will be defined as `SORT-ID` and the internal structure of function and predicate symbols will be defined as `ID`.)

$$\begin{aligned} S &\in \mathit{SortSet} = \mathit{FinSet}(\mathit{Sort}) \\ (w, s) \text{ or } ws &\in \mathit{FunProfile} = \mathit{FinSeq}(\mathit{Sort}) \times \mathit{Sort} \\ \mathit{TF}, \mathit{PF} &\in \mathit{FunSet} = \mathit{FunProfile} \rightharpoonup \mathit{FinSet}(\mathit{FunName}) \\ w &\in \mathit{PredProfile} = \mathit{FinSeq}(\mathit{Sort}) \\ P &\in \mathit{PredSet} = \mathit{PredProfile} \rightharpoonup \mathit{FinSet}(\mathit{PredName}) \end{aligned}$$

For a set of total function symbols TF over S it is required that $\mathit{Dom}(\mathit{TF}) = \mathit{FinSeq}(S) \times S$ and that $\mathit{TF}_{ws} \neq \emptyset$ for only finitely many function profiles $ws \in \mathit{FinSeq}(S) \times S$, and similarly for a set of partial function symbols PF. For a set of predicate symbols P over S it is required that $\mathit{Dom}(P) = \mathit{FinSeq}(S)$ and that $P_w \neq \emptyset$ for only finitely many predicate profiles $w \in \mathit{FinSeq}(S)$.

$$\begin{aligned} &(S, \mathit{TF}, \mathit{PF}, P) \\ &\qquad \text{or } \Sigma \in \mathit{Signature} = \\ &\qquad\qquad \mathit{SortSet} \times \mathit{FunSet} \times \mathit{FunSet} \times \mathit{PredSet} \end{aligned}$$

Requirements on a signature $(S, \mathit{TF}, \mathit{PF}, P)$:

- TF and PF are sets of total resp. partial function symbols over S
- P is a set of predicate symbols over S
- for all $ws \in \mathit{FinSeq}(S) \times S,\ \mathit{TF}_{ws} \cap \mathit{PF}_{ws} = \emptyset$

(An alternative to the use of the separate signature components TF and PF would be a single component F with a totality marker, so e.g. $F : \mathit{FunProfile} \times \{\mathsf{total}, \mathsf{partial}\} \rightharpoonup \mathit{FinSet}(\mathit{FunName})$ cf. [38, 67].)

Later we will need *signature extensions* as well. These are signature fragments that are interpreted relative to some other signature. First we define *signature fragments*.

$$\begin{aligned}(S, \mathit{TF}, \mathit{PF}, P) \in \mathit{SigFragment} = \\ \mathit{SortSet} \times \mathit{FunSet} \times \mathit{FunSet} \times \mathit{PredSet}\end{aligned}$$

These are simply signatures minus the validity requirements.

Union of signature fragments is defined as follows:

$$\begin{aligned}(S, \mathit{TF}, \mathit{PF}, P) \cup (S', \mathit{TF}', \mathit{PF}', P') = \\ \mathit{reconcile}(S \cup S',\ & \mathit{complete}(\mathit{TF} \cup \mathit{TF}', \mathit{FinSeq}(S'') \times S''), \\ & \mathit{complete}(\mathit{PF} \cup \mathit{PF}', \mathit{FinSeq}(S'') \times S''), \\ & \mathit{complete}(P \cup P', \mathit{FinSeq}(S'')))\end{aligned}$$

where

$$\begin{aligned}S'' = S \cup S' & \cup \mathit{sorts}(\mathit{Dom}(\mathit{TF})) \cup \mathit{sorts}(\mathit{Dom}(\mathit{PF})) \cup \mathit{sorts}(\mathit{Dom}(P)) \\ & \cup \mathit{sorts}(\mathit{Dom}(\mathit{TF}')) \cup \mathit{sorts}(\mathit{Dom}(\mathit{PF}')) \cup \mathit{sorts}(\mathit{Dom}(P'))\end{aligned}$$

and

$$\mathit{reconcile}(S, \mathit{TF}, \mathit{PF}, P) = (S, \mathit{TF}, \{ws \mapsto \mathit{PF}_{ws} \backslash \mathit{TF}_{ws} \mid ws \in \mathit{Dom}(\mathit{PF})\}, P).$$

Here, $\mathit{sorts}(T)$ is the set of sorts appearing in function/predicate profiles in T. The idea of this definition is to give the same result as if signature fragments were defined as sets of individual sort/function/predicate declarations. Note that any signature is also a signature fragment so this definition also defines the union of two signatures as well as the union of a signature and a signature fragment. According to this definition, the union of two signatures will always be a signature with $S'' = S \cup S'$. When a function name is declared as both partial and total, the *reconcile* function causes it to be regarded as total in the union, as required in Sects. I:2.3.2 and I:4.2.3.

$$(S, \mathit{TF}, \mathit{PF}, P) \text{ or } \Delta \in \mathit{Extension} = \mathit{SigFragment}$$

A signature extension relative to a signature Σ is a signature fragment Δ such that $\Sigma \cup \Delta$ (the "target" of the signature extension) is a signature. This guarantees that all the sorts used for function and predicate profiles in Δ are declared in either Δ or Σ.

Proposition 2.1. *If Δ and Δ' are signature extensions relative to Σ then $\Delta \cup \Delta'$ is a signature extension relative to Σ.*

Proof. Straightforward. □

A signature Σ is a *subsignature* of a signature Σ' if there is some extension Δ relative to Σ such that $\Sigma' = \Sigma \cup \Delta$. Note that this allows a function name to be a partial function symbol in Σ but a total function symbol in Σ'.

Symbols used to identify sorts, operations, and predicates may be *overloaded.* For example, it is possible that $f \in TF_{ws}$ and $f \in TF_{ws'}$ for $ws \neq ws'$, as well as $f \in S$. To ensure that there is no ambiguity in sentences at this level, function symbols f and predicate symbols p are always *qualified* by profiles when used, written f_{ws} and p_w respectively. (The language considered later in this chapter allows the omission of such qualifications when they are unambiguously determined by the context.)

$$\begin{aligned} f_{ws} &\in \mathit{QualFunName} = \mathit{FunName} \times \mathit{FunProfile} \\ p_w &\in \mathit{QualPredName} = \mathit{PredName} \times \mathit{PredProfile} \end{aligned}$$

Requirements on a qualified function name f_{ws} over $\Sigma = (S, TF, PF, P)$:

- $ws \in \mathit{FinSeq}(S) \times S$
- $f \in TF_{ws} \cup PF_{ws}$

Requirements on a qualified predicate name p_w over $\Sigma = (S, TF, PF, P)$:

- $w \in \mathit{FinSeq}(S)$
- $p \in P_w$

Following [39], Chaps. 4 and 5 below require that we define a set *SigSym* of *signature symbols* and a function $|.|$ taking any signature to the set of signature symbols it contains (in fact we need a functor $|.| : \mathbf{Sig} \to \mathbf{Set}$ having certain properties, see Prop. 2.4 below). Signature symbols are essentially just qualified function/predicate names together with sort names.

$$\begin{array}{rl} & \mathit{SigSym} = \\ s \in & \qquad \mathit{Sort} \uplus \\ f_{ws} \in & \qquad \mathit{QualFunName} \uplus \\ p_w \in & \qquad \mathit{QualPredName} \end{array}$$

If $\Sigma = (S, TF, PF, P)$, we define $|\Sigma| \subseteq \mathit{SigSym}$ as follows:

$$\begin{aligned} |\Sigma| = S &\cup \{f_{ws} \mid ws \in \mathit{FinSeq}(S) \times S, f \in TF_{ws} \cup PF_{ws}\} \\ &\cup \{p_w \mid w \in \mathit{FinSeq}(S), p \in P_w\} \end{aligned}$$

A *many-sorted signature morphism* $\sigma : \Sigma \to \Sigma'$ maps symbols in Σ to symbols in Σ'. A partial function symbol may be mapped to a total function symbol, but not vice versa.

$$\begin{aligned}
\sigma^{\mathsf{S}} &\in \mathit{SMap} = \mathit{Sort} \overset{\text{fin}}{\to} \mathit{Sort}\\
\sigma^{\mathsf{TF}} &\in \mathit{TFMap} = \mathit{FunProfile} \rightharpoonup (\mathit{FunName} \overset{\text{fin}}{\to} \mathit{FunName})\\
\sigma^{\mathsf{PF}} &\in \mathit{PFMap} = \mathit{FunProfile} \rightharpoonup (\mathit{FunName} \overset{\text{fin}}{\to} \mathit{FunName})\\
\sigma^{\mathsf{P}} &\in \mathit{PMap} = \mathit{PredProfile} \rightharpoonup (\mathit{PredName} \overset{\text{fin}}{\to} \mathit{PredName})
\end{aligned}$$

$$\begin{aligned}
&(\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}}) : \Sigma \to \Sigma'\\
&\text{or } \sigma : \Sigma \to \Sigma' \in \mathit{SignatureMorphism} =\\
&\qquad\qquad \mathit{Signature}\\
&\qquad\qquad\quad \times \mathit{SMap} \times \mathit{TFMap} \times \mathit{PFMap} \times \mathit{PMap}\\
&\qquad\qquad\quad \times \mathit{Signature}
\end{aligned}$$

Requirements on a signature morphism $(\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}}) : (S, \mathit{TF}, \mathit{PF}, P) \to (S', \mathit{TF}', \mathit{PF}', P')$:

- $\sigma^{\mathsf{S}} : S \to S'$
- $\mathit{Dom}(\sigma^{\mathsf{TF}}) = \mathit{Dom}(\sigma^{\mathsf{PF}}) = \mathit{FinSeq}(S) \times S$
- for all $ws \in \mathit{FinSeq}(S) \times S$:
  - $\sigma^{\mathsf{TF}}_{ws} : \mathit{TF}_{ws} \to \mathit{TF}'_{\sigma^{\mathsf{S}}(ws)}$
  - $\sigma^{\mathsf{PF}}_{ws} : \mathit{PF}_{ws} \to \mathit{TF}'_{\sigma^{\mathsf{S}}(ws)} \cup \mathit{PF}'_{\sigma^{\mathsf{S}}(ws)}$
- $\mathit{Dom}(\sigma^{\mathsf{P}}) = \mathit{FinSeq}(S)$
- for all $w \in \mathit{FinSeq}(S)$, $\sigma^{\mathsf{P}}_{w} : P_w \to P'_{\sigma^{\mathsf{S}}(w)}$

where, for $w = \langle s_1, \ldots, s_n \rangle$, $\sigma^{\mathsf{S}}(w) = \langle \sigma^{\mathsf{S}}(s_1), \ldots, \sigma^{\mathsf{S}}(s_n) \rangle$ and $\sigma^{\mathsf{S}}(w, s) = (\sigma^{\mathsf{S}}(w), \sigma^{\mathsf{S}}(s))$. If Σ is a subsignature of Σ', we write $\Sigma \hookrightarrow \Sigma'$ for the evident signature morphism. Such a signature morphism is called a *signature inclusion*. Note that a signature extension Δ relative to Σ can be viewed more abstractly as the signature inclusion $\Sigma \hookrightarrow \Sigma \cup \Delta$. However, information about any re-declaration in Δ of symbols in Σ is lost by this abstraction. Therefore Δ is kept explicitly together with the signature inclusion in Chap. 4.

If $\sigma : \Sigma \to \Sigma'$ and $\rho : \Sigma' \to \Sigma''$ are signature morphisms, where $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$, $\rho = (\rho^{\mathsf{S}}, \rho^{\mathsf{TF}}, \rho^{\mathsf{PF}}, \rho^{\mathsf{P}})$ and $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$, then the composition $\rho \circ \sigma : \Sigma \to \Sigma''$ is the signature morphism $(\delta^{\mathsf{S}}, \delta^{\mathsf{TF}}, \delta^{\mathsf{PF}}, \delta^{\mathsf{P}})$ where

$$\begin{aligned}
\delta^{\mathsf{S}} &= \rho^{\mathsf{S}} \circ \sigma^{\mathsf{S}},\\
\delta^{\mathsf{TF}} &= \{ws \mapsto \rho^{\mathsf{TF}}_{\sigma^{\mathsf{S}}(ws)} \circ \sigma^{\mathsf{TF}}_{ws} \mid ws \in \mathit{FinSeq}(S) \times S\},\\
\delta^{\mathsf{PF}} &= \{ws \mapsto (\rho^{\mathsf{TF}}_{\sigma^{\mathsf{S}}(ws)} \cup \rho^{\mathsf{PF}}_{\sigma^{\mathsf{S}}(ws)}) \circ \sigma^{\mathsf{PF}}_{ws} \mid ws \in \mathit{FinSeq}(S) \times S\},\\
\delta^{\mathsf{P}} &= \{w \mapsto \rho^{\mathsf{P}}_{\sigma^{\mathsf{S}}(w)} \circ \sigma^{\mathsf{P}}_{w} \mid w \in \mathit{FinSeq}(S)\}
\end{aligned}$$

Identity morphisms id_{Σ} are obvious.

Proposition 2.2. *The composition of signature morphisms does indeed yield a signature morphism.*

Proof. In the definition of δ^{PF}, $\rho^{\mathsf{TF}}_{\sigma^{\mathsf{S}}(ws)} \cup \rho^{\mathsf{PF}}_{\sigma^{\mathsf{S}}(ws)}$ is a function because $\mathit{TF}'_{\sigma^{\mathsf{S}}(ws)} \cap \mathit{PF}'_{\sigma^{\mathsf{S}}(ws)} = \emptyset$. The rest of the proof is straightforward. □

Proposition 2.3. *Signatures and signature morphisms form a finitely cocomplete category,* **Sig**.

Proof. It is easy to see that **Sig** is a category. Regarding finite cocompleteness, see [38] for a more general result. □

If $\sigma : \Sigma \to \Sigma'$ is a signature morphism, where $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$, $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$ and $\Sigma' = (S', \mathit{TF}', \mathit{PF}', P')$, then the function $|\sigma| : |\Sigma| \to |\Sigma'|$ is defined as follows:

$$\begin{aligned}
|\sigma|(s) &= \sigma^{\mathsf{S}}(s) \qquad \text{for all } s \in S \\
|\sigma|(f_{ws}) &= \begin{cases} \sigma^{\mathsf{TF}}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)} \text{ for } f \in \mathit{TF}_{ws} \\ \sigma^{\mathsf{PF}}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)} \text{ for } f \in \mathit{PF}_{ws} \end{cases} \\
&\qquad\qquad \text{for all } ws \in \mathit{FinSeq}(S) \times S \\
|\sigma|(p_w) &= \sigma^{\mathsf{P}}_{w}(p)_{\sigma^{\mathsf{S}}(w)} \\
&\qquad\qquad \text{for all } w \in \mathit{FinSeq}(S) \text{ and } p \in P_w
\end{aligned}$$

Proposition 2.4. $|.| :$ **Sig** $\to$ **Set** *is a faithful functor.*

Proof. It is easy to see that $|.|$ is a functor. Faithfulness is also obvious: $|\sigma|$ (together with the partiality data in Σ and Σ') carries no less information than $\sigma : \Sigma \to \Sigma'$. □

Proposition 2.5. *A signature morphism $\sigma : \Sigma \to \Sigma'$ is a signature inclusion iff $|\sigma|$ is an inclusion of $|\Sigma|$ into $|\Sigma'|$.*

Proof. Straightforward. It is essential that in a signature inclusion $\sigma : \Sigma \to \Sigma'$, a function name may be a partial function symbol in Σ but a total function symbol in Σ'. □

2.1.2 Models

For a many-sorted signature Σ, a *many-sorted model* $M \in \mathbf{Mod}(\Sigma)$ assigns a non-empty *carrier set* to each sort in Σ, a *partial* resp. *total function* to each partial resp. total function symbol, and a *predicate* to each predicate symbol. Requiring carriers to be non-empty simplifies deduction [21] and will allow axioms in specifications to be implicitly universally quantified, see Sect. 2.4.

$$\begin{aligned}
S^M(s) \text{ or } s^M &\in \mathbf{Carrier} = \text{the class of all sets} \\
S^M &\in \mathbf{Carriers} = \mathit{Sort} \overset{\text{fin}}{\to} \mathbf{Carrier} \\
F^M_{ws}(f) \text{ or } f^M &\in \mathbf{PartialFun} = \text{the class of all partial functions} \\
F^M &\in \mathbf{PartialFuns} = \mathit{FunProfile} \rightharpoonup (\mathit{FunName} \overset{\text{fin}}{\to} \mathbf{PartialFun}) \\
P^M_w(p) \text{ or } p^M &\in \mathbf{Pred} = \text{the class of all predicates} \\
P^M &\in \mathbf{Preds} = \mathit{PredProfile} \rightharpoonup (\mathit{PredName} \overset{\text{fin}}{\to} \mathbf{Pred}) \\
(S^M, F^M, P^M)& \\
\text{or } M &\in \mathbf{Model} = \mathbf{Carriers} \times \mathbf{PartialFuns} \times \mathbf{Preds} \\
\mathcal{M} &\in \mathbf{ModelClass} = \mathit{Set}(\mathbf{Model})
\end{aligned}$$

Requirements on a Σ-model $M = (S^M, F^M, P^M)$ for $\Sigma = (S, TF, PF, P)$:

- $Dom(S^M) = S$
- for all $s \in S$, $S^M(s) \neq \emptyset$
- $Dom(F^M) = FinSeq(S) \times S$
- for all $w \in FinSeq(S)$ and $s \in S$:
  - $Dom(F^M_{w,s}) = PF_{w,s} \cup TF_{w,s}$
  - for all $f \in TF_{w,s}$, $F^M_{w,s}(f) : w^M \to s^M$ (a *total* function)
  - for all $f \in PF_{w,s}$, $F^M_{w,s}(f) : w^M \rightharpoonup s^M$
- $Dom(P^M) = FinSeq(S)$
- for all $w \in FinSeq(S)$:
  - $Dom(P^M_w) = P_w$
  - for all $p \in P_w$, $P^M_w(p) \subseteq w^M$

where $\langle s_1, \ldots, s_n \rangle^M = s_1^M \times \cdots \times s_n^M$.

Every model in a Σ-model class $\mathcal{M}$ is required to be a valid Σ-model.

Given two Σ-models $M, M' \in \mathbf{Mod}(\Sigma)$, a *many-sorted homomorphism* $h : M \to M'$ maps the values in the carriers of M to values in the corresponding carriers of M' in such a way that the values of functions and their definedness is preserved, as well as the truth of predicates.

$$h : M \to M' \in \mathbf{Homomorphism} = \mathbf{Model} \times (Sort \overset{\text{fin}}{\to} \mathbf{PartialFun}) \times \mathbf{Model}$$

Requirements on a Σ-homomorphism $h : M \to M'$ for $\Sigma = (S, TF, PF, P)$:

- M and M' are valid Σ-models
- $Dom(h) = S$
- for all $s \in S$, $h_s : s^M \to s^{M'}$ (a total function)
- for all $w = \langle s_1, \ldots, s_n \rangle \in FinSeq(S)$, $s \in S$, $f \in TF_{w,s} \cup PF_{w,s}$ and $a_1 \in s_1^M, \ldots, a_n \in s_n^M$, whenever $f^M(a_1, \ldots, a_n)$ is defined then so is $f^{M'}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$, and in that case $h_s(f^M(a_1, \ldots, a_n)) = f^{M'}(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$
- for all $w = \langle s_1, \ldots, s_n \rangle \in FinSeq(S)$, $p \in P_w$ and $a_1 \in s_1^M, \ldots, a_n \in s_n^M$, if $(a_1, \ldots, a_n) \in p^M$ then $(h_{s_1}(a_1), \ldots, h_{s_n}(a_n)) \in p^{M'}$.

Composition of homomorphisms is as usual: if $h : M \to M'$ and $h' : M' \to M''$ are Σ-homomorphisms for $\Sigma = (S, TF, PF, P)$, then the Σ-homomorphism $h' \circ h : M \to M''$ is given by $(h' \circ h)_s = h'_s \circ h_s$ for all $s \in S$. Identity homomorphisms are S-sorted identity functions.

Proposition 2.6. *The composition $h' \circ h : M \to M''$ is indeed a Σ-homomorphism.*

Proof. Routine. □

Proposition 2.7. *Σ-models together with Σ-homomorphisms form a category,* $\mathbf{Mod}(\Sigma)$*.*

Proof. Easy. □

A signature morphism $\Sigma : \Sigma \to \Sigma'$ determines the *many-sorted reduct* of each Σ'-model resp. Σ'-homomorphism to a Σ-model resp. Σ-homomorphism, defined by interpreting symbols of Σ in the reduct in the same way that their images under σ are interpreted.

Let $M' = (S^{M'}, F^{M'}, P^{M'})$ be a Σ'-model and let $\sigma : \Sigma \to \Sigma'$ be a signature morphism where $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$ and $\Sigma = (S, TF, PF, P)$. The *reduct* of M' with respect to σ is the Σ-model $M'|_\sigma = (S^M, F^M, P^M)$ defined as follows:

$$S^M = S^{M'} \circ \sigma^{\mathsf{S}}$$

$$F^M_{ws}(f) = \begin{cases} F^{M'}_{\sigma^{\mathsf{S}}(ws)}(\sigma^{\mathsf{TF}}_{ws}(f)) \text{ if } f \in TF_{ws} \\ F^{M'}_{\sigma^{\mathsf{S}}(ws)}(\sigma^{\mathsf{PF}}_{ws}(f)) \text{ if } f \in PF_{ws} \end{cases}$$

$$P^M_w(p) = P^{M'}_{\sigma^{\mathsf{S}}(w)}(\sigma^{\mathsf{P}}_w(p))$$

Proposition 2.8. *If $\sigma : \Sigma \to \Sigma'$ is a signature morphism and M' is a Σ'-model then $M'|_\sigma$ is indeed a Σ-model.*

Proof. Routine. □

Suppose that Σ is a subsignature of Σ', so there is a signature inclusion $\Sigma \hookrightarrow \Sigma'$. Then we sometimes write $M'|_\Sigma$ as an abbreviation for $M'|_{\Sigma \hookrightarrow \Sigma'}$, and we say that a Σ'-model M' *extends* a Σ-model M if $M'|_\Sigma = M$. These notations are extended to classes of models, so $\mathcal{M}'|_\sigma = \{M'|_\sigma \mid M' \in \mathcal{M}'\}$ if $\sigma : \Sigma \to \Sigma'$ and $\mathcal{M}'$ is a class of Σ'-models, and $\mathcal{M}'|_\Sigma = \{M'|_\Sigma \mid M' \in \mathcal{M}'\}$ if σ is a signature inclusion.

Let $h' : M1' \to M2'$ be a Σ'-homomorphism and let $\sigma : \Sigma \to \Sigma'$ be a signature morphism where $\Sigma = (S, TF, PF, P)$ and $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$. The *reduct* of h' with respect to σ is the Σ-homomorphism $h'|_\sigma : M1'|_\sigma \to M2'|_\sigma$ defined by $(h'|_\sigma)_s = h'_{\sigma^{\mathsf{S}}(s)}$ for all $s \in S$. If Σ is a subsignature of Σ' then we sometimes write $h'|_\Sigma$ as an abbreviation for $h'|_{\Sigma \hookrightarrow \Sigma'}$.

Proposition 2.9. *If $\sigma : \Sigma \to \Sigma'$ is a signature morphism and $h' : M1' \to M2'$ is a Σ'-homomorphism then $h'|_\sigma : M1'|_\sigma \to M2'|_\sigma$ is indeed a Σ-homomorphism.*

Proof. Easy. □

Proposition 2.10. *Reduct of models and homomorphisms extends* **Mod** *to a finitely continuous functor* **Mod** : $\mathbf{Sig}^{op} \to \mathbf{CAT}$ *(i.e.* **Mod** *takes finite colimits in* **Sig** *to limits in* **CAT***).*

Proof. It is easy to see that **Mod** is a functor. For continuity, see [38] for a sketch of the proof of a more general result; cf. [15]. □

Let $h : M \to M'$ be a Σ-homomorphism. If there is a Σ-homomorphism $h^{-1} : M' \to M$ such that $h \circ h^{-1}$ is the identity on M' and $h^{-1} \circ h$ is the identity on M then h is a Σ- *isomorphism* and we write $M \cong M'$.

2.1.3 Sentences

The *many-sorted terms* on a signature Σ and a set X of variables consist of variables from X together with applications of qualified function symbols to argument terms of appropriate sorts. We refer to such terms as *fully-qualified terms*, to avoid confusion with the terms of the language considered later in this chapter, which allow explicit qualifications to be omitted when they are determined by the context.

Following the order of presentation in the CASL Summary, we leave the syntax of variables (Var) unspecified for now. It will be defined in Sect. 2.4 below.

$$\begin{aligned} x &\in \mathit{Var} \\ X &\in \mathit{Variables} = \mathit{Sort} \overset{\text{fin}}{\to} \mathit{FinSet}(\mathit{Var}) \\ x_s &\in \mathit{QualVarName} = \mathit{Var} \times \mathit{Sort} \end{aligned}$$

Requirements on an S-sorted set of variables X:

- $\mathit{Dom}(X) = S$
- for all $s, s' \in S$ such that $s \neq s'$, $X_s \cap X_{s'} = \emptyset$.

In a qualified variable name x_s, it is required that $s \in S$.

We write $X + \{x_s\}$ for the $(S \cup \{s\})$-sorted set of variables such that

$$(X + \{x_s\})_s = \begin{cases} X_s \cup \{x\} & \text{if } s \in \mathit{Dom}(X) \\ \{x\} & \text{otherwise} \end{cases}$$

and $(X + \{x_s\})_{s'} = X_{s'} \setminus \{x\}$ for $s' \in S$ such that $s' \neq s$. We write $X + X'$ for the extension of this to arbitrary S'-sorted sets of variables X'.

Proposition 2.11. *If X is valid for S and X' is valid for S' then $X + X'$ is valid for $S \cup S'$.*

Proof. Easy. □

If $x^i \neq x^j$ for all $1 \leq i \neq j \leq n$ then we use $\{x^1_{s_1}, \ldots, x^n_{s_n}\}$ to abbreviate $\{x^1_{s_1}\} + \cdots + \{x^n_{s_n}\}$. (The pre-condition means that the order is immaterial, as the set notation suggests.)

The definitions of fully-qualified terms and formulas are mutually recursive.

$$\begin{array}{rl}
t \in \mathit{FQTerm} = & \\
x_s \in & \mathit{QualVarName} \uplus \\
f_{ws}\langle t_1, \ldots, t_n\rangle \in & \mathit{QualFunName} \times \mathit{FinSeq}(\mathit{FQTerm}) \uplus \\
\varphi \to t \mid t' \in & \mathit{Formula} \times \mathit{FQTerm} \times \mathit{FQTerm}
\end{array}$$

For any $t \in \mathit{FQTerm}$, define $\mathit{sort}(t) \in \mathit{Sort}$ as follows:

$$\begin{array}{rl}
\mathit{sort}(x_s) = & s \\
\mathit{sort}(f_{w,s}\langle t_1, \ldots, t_n\rangle) = & s \\
\mathit{sort}(\varphi \to t' \mid t'') = & \begin{cases} \mathit{sort}(t') & \text{if } \mathit{sort}(t') = \mathit{sort}(t'') \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}$$

Requirements on a fully-qualified Σ-term t over an S-sorted set of variables X, for $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$:

- if t is x_s, then x_s is valid for S and $x \in X_s$
- if t is $f_{w,s}\langle t_1, \ldots, t_n\rangle$, then:
  - $f_{w,s}$ is a valid qualified function name over Σ
  - $t_1, \ldots, t_n$ are valid fully-qualified Σ-terms over X
  - $|w| = n$
  - $w = \langle \mathit{sort}(t_1), \ldots, \mathit{sort}(t_n)\rangle$
- if t is $\varphi \to t' \mid t''$, then:
  - φ is a valid Σ-formula over X
  - t' and t'' are valid fully-qualified Σ-terms over X
  - $\mathit{sort}(t') = \mathit{sort}(t'')$

The fully-qualified term $\varphi \to t \mid t'$ is only needed to deal with the conditional term construct, see Sect. 2.5.4 below. An alternative is to deal with these by transformation as described in Sect. I:2.5.4 of the CASL Summary. Then fully-qualified terms of the form $\varphi \to t \mid t'$ are not required. Since these terms are non-standard, this is what is done in the proof calculus for basic specifications in Sect. IV:2.

The *many-sorted sentences* in **Sen**(Σ) are sort-generation constraints – described below – and the usual closed many-sorted first-order logic formulas, built from atomic formulas (application of qualified predicate symbols to argument terms of appropriate sorts, assertions about the definedness of fully-qualified terms, and existential and strong equations between fully-qualified terms of the same sort) using quantification and logical connectives. Predicate application, existential equations, implication and universal quantification are taken as primitive, the other forms being regarded as derived.

$$\begin{array}{rl}
\varphi \in \mathit{Formula} = & \\
p_w\langle t_1, \ldots, t_n\rangle \in & \mathit{QualPredName} \times \mathit{FinSeq}(\mathit{FQTerm}) \uplus \\
t \overset{\mathrm{e}}{=} t' \in & \mathit{FQTerm} \times \mathit{FQTerm} \uplus \\
\mathit{false} \in & \mathit{unit} \uplus \\
\varphi \Rightarrow \varphi' \in & \mathit{Formula} \times \mathit{Formula} \uplus \\
\forall x_s . \varphi \in & \mathit{QualVarName} \times \mathit{Formula}
\end{array}$$

Requirements on a Σ-formula φ over an S-sorted set of variables X, for $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$:

- if φ is $p_w\langle t_1, \ldots, t_n\rangle$, then:
  - p_w is a valid qualified predicate name over Σ
  - $t_1, \ldots, t_n$ are valid fully-qualified Σ-terms over X
  - $|w| = n$
  - $w = \langle \mathit{sort}(t_1), \ldots, \mathit{sort}(t_n)\rangle$
- if φ is $t \overset{\mathrm{e}}{=} t'$, then:
  - t and t' are valid fully-qualified Σ-terms over X
  - $\mathit{sort}(t) = \mathit{sort}(t')$
- if φ is $\varphi' \Rightarrow \varphi''$, then φ' and φ'' are valid Σ-formulas over X
- if φ is $\forall x_s.\varphi'$, then:
  - x_s is valid for S
  - φ' is a valid Σ-formula over $X + \{x_s\}$

Abbreviations are defined as follows:

$$\begin{aligned}
\neg\varphi &\text{ abbreviates } \varphi \Rightarrow \mathit{false}\\
\varphi \vee \varphi' &\text{ abbreviates } (\neg\varphi) \Rightarrow \varphi'\\
\varphi \wedge \varphi' &\text{ abbreviates } \neg(\neg\varphi \vee \neg\varphi')\\
\varphi \Leftrightarrow \varphi' &\text{ abbreviates } (\varphi \Rightarrow \varphi') \wedge (\varphi' \Rightarrow \varphi)\\
\mathit{true} &\text{ abbreviates } \neg\mathit{false}\\
D(t) &\text{ abbreviates } t \overset{\mathrm{e}}{=} t\\
t \overset{\mathrm{s}}{=} t' &\text{ abbreviates } (D(t) \Rightarrow t \overset{\mathrm{e}}{=} t') \wedge (D(t') \Rightarrow t \overset{\mathrm{e}}{=} t')\\
\forall\{x^1_{s^1}, \ldots, x^n_{s^n}\}.\varphi &\text{ abbreviates } \forall x^1_{s^1}.\cdots\forall x^n_{s^n}.\varphi\\
\exists X.\varphi &\text{ abbreviates } \neg(\forall X.\neg\varphi)\\
\exists! X.\varphi &\text{ abbreviates } \exists X.(\varphi \wedge \forall\hat{X}.(\varphi[\hat{X}/X] \Rightarrow X \overset{\mathrm{e}}{=} \hat{X}))
\end{aligned}$$

where in the last clause the variables $\hat{X}$ are variants of X chosen to avoid all variable clashes, $\varphi[\hat{X}/X]$ is substitution, and $X \overset{\mathrm{e}}{=} \hat{X}$ abbreviates the evident conjunction of equations. The notation for strong equations is deliberately more explicit than that in Casl itself, where undecorated equality is used.

Let X be an S-sorted set of variables, for $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$. The set $\mathit{FV}(t)$ of *free variables* of a fully-qualified Σ-term t over X, and the set $\mathit{FV}(\varphi)$ of free variables of a Σ-formula φ over X, are defined simultaneously by induction as follows, giving an S-sorted set of variables:

- $\mathit{FV}(x_s) = \{x_s\}$
- $\mathit{FV}(f_{ws}\langle t_1, \ldots, t_n\rangle) = \mathit{FV}(t_1) \cup \cdots \cup \mathit{FV}(t_n)$
- $\mathit{FV}(\varphi' \to t' \mid t'') = \mathit{FV}(\varphi') \cup \mathit{FV}(t') \cup \mathit{FV}(t'')$

- $\mathit{FV}(p_w\langle t_1, \ldots, t_n\rangle) = \mathit{FV}(t_1) \cup \cdots \cup \mathit{FV}(t_n)$
- $\mathit{FV}(t_1 \overset{\mathrm{e}}{=} t_2) = \mathit{FV}(t_1) \cup \mathit{FV}(t_2)$
- $\mathit{FV}(\mathit{false}) = \emptyset$
- $\mathit{FV}(\varphi' \Rightarrow \varphi'') = \mathit{FV}(\varphi') \cup \mathit{FV}(\varphi'')$
- $\mathit{FV}(\forall x_s.\varphi') = \mathit{FV}(\varphi') \setminus \{x_s\}$

If $n = 0$ then $\varphi_1 \wedge \cdots \wedge \varphi_n$ means *true*, $\varphi_1 \vee \cdots \vee \varphi_n$ means *false*, and $\forall x^1_{s^1}. \cdots \forall x^n_{s^n}.\varphi$ means φ. (This is metanotation: ellipses are not included in the syntax of sentences.)

Let φ be a Σ-formula over X and let $(\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$ be a signature morphism $\sigma : \Sigma \rightarrow \Sigma'$ where $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$ and $\Sigma' = (S', \mathit{TF}', \mathit{PF}', P')$. Let X' be the S'-sorted set of variables such that $X'_{s'} = \bigcup_{\sigma^{\mathsf{S}}(s)=s'} X_s$ for all $s' \in S'$. The *translation* of φ along σ is the Σ'-formula $\sigma(\varphi)$ over X' obtained by replacing each qualified variable name x_s in φ by $x_{\sigma^{\mathsf{S}}(s)}$, each qualified function name f_{ws} such that $f \in \mathit{TF}_{ws}$ by $\sigma^{\mathsf{TF}}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)}$, each qualified function name f_{ws} such that $f \in \mathit{PF}_{ws}$ by $\sigma^{\mathsf{PF}}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)}$, and each qualified predicate name p_w by $\sigma^{\mathsf{P}}_{w}(p)_{\sigma^{\mathsf{S}}(w)}$.

Proposition 2.12. *If $\sigma : \Sigma \rightarrow \Sigma'$ is a signature morphism and φ is a Σ-formula over X then $\sigma(\varphi)$ is indeed a Σ'-formula over X'. If X is empty then so is X'.*

Proof. Straightforward. □

The sentences **Sen**(Σ) also include *sort-generation constraints*, used to require that models are reachable on a subset of sorts.

$$(S', F', \sigma) \in \mathit{Constraint} = \mathit{SortSet} \times \mathit{FunSet} \times \mathit{SignatureMorphism}$$

Requirements on a Σ-constraint (S', F', σ):

- $\sigma : \overline{\Sigma} \rightarrow \Sigma$ where $\overline{\Sigma} = (\overline{S}, \overline{\mathit{TF}}, \overline{\mathit{PF}}, \overline{P})$, and then:
- $S' \subseteq \overline{S}$
- $\mathit{Dom}(F') = \mathit{FinSeq}(\overline{S}) \times \overline{S}$
- for all $ws \in \mathit{FinSeq}(\overline{S}) \times \overline{S}$, $F'_{ws} \subseteq \overline{\mathit{TF}}_{ws} \cup \overline{\mathit{PF}}_{ws}$

Let $\sigma' : \Sigma \rightarrow \Sigma''$ be a signature morphism. The *translation* $\sigma'(S', F', \sigma)$ of a Σ-constraint (S', F', σ) along σ' is the Σ''-constraint $(S', F', \sigma' \circ \sigma)$.

Proposition 2.13. *Translating a Σ-constraint along $\sigma : \Sigma \rightarrow \Sigma'$ gives a Σ'-constraint.*

Proof. Obvious. □

We use the abbreviation (S', F') for the Σ-constraint (S', F', id_{Σ}). Only constraints of this kind are introduced by Casl specifications, see Sects. 2.3.4 and 2.3.5. Constraints with non-identity third components arise only when constraints introduced by Casl specifications are translated along signature morphisms.

$$\psi \in \mathit{Sentence} = \mathit{Formula} \uplus \mathit{Constraint}$$

Requirements on a Σ-sentence ψ:

- if ψ is a formula, it is required to be a valid Σ-formula over the empty set of variables
- if ψ is a constraint, it is required to be a valid Σ-constraint

Proposition 2.14. *The mapping from signatures Σ to sets of Σ-sentences, together with translation of sentences along signature morphisms, gives a functor* $\mathbf{Sen} : \mathbf{Sig} \to \mathbf{Set}$.

Proof. The requirement that variables cannot be overloaded is crucial because it allows the translated sets of variables X' above to be formed without the use of disjoint union. Given this, the proof is straightforward. □

$$(\Delta, \Psi) \in \mathit{Enrichment} = \mathit{Extension} \times \mathit{FinSet}(\mathit{Sentence})$$

Requirements on an enrichment (Δ, Ψ) relative to a signature Σ:

- Δ is a signature extension relative to Σ
- Each $\psi \in \Psi$ is a $\Sigma \cup \Delta$-sentence

2.1.4 Satisfaction

The satisfaction of a Σ-formula in a Σ-model is determined as usual by the holding of its atomic formulas w.r.t. assignments of values to all the variables that occur in them. The value of a term may be undefined, due to the presence of partial functions. Note, however, that the satisfaction of sentences is two-valued.

A predicate application holds iff the values of all its argument terms are defined and give a tuple that belongs to the predicate. A definedness assertion holds iff the value of the term is defined. An existential equation holds iff the values of both terms are defined and identical, whereas a strong equation holds also when the values of both terms are undefined.

$$\rho \in \mathbf{Assignment} = \mathit{Sort} \overset{\text{fin}}{\to} \mathbf{PartialFun}$$

Let $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$ be a signature, M a Σ-model, and X an S-sorted set of variables. Requirements on an assignment ρ of X into M, written $\rho : X \to M$:

- $\mathit{Dom}(\rho) = S$
- for all $s \in S$, $\rho_s : X_s \to s^M$

If $a \in s^M$ then we write $\rho[x_s \mapsto a]$ for the assignment of $X + \{x_s\}$ into M such that $\rho[x_s \mapsto a]_s(x) = a$, $\rho[x_s \mapsto a]_s(x') = \rho_s(x')$ for $x' \neq x$, and $\rho[x_s \mapsto a]_{s'}(x') = \rho_{s'}(x')$ for $s' \neq s$ and $x' \neq x$.

We now simultaneously define three things inductively by means of inference rules:

- the *value* $[\![t]\!]_\rho$ of a fully-qualified Σ-term t over X in a Σ-model M with respect to an assignment $\rho : X \to M$;
- *satisfaction* of a Σ-formula φ over X by a Σ-model M under an assignment $\rho : X \to M$, written $M \models_\rho \varphi$; and
- *non-satisfaction* of φ by M under ρ, written $M \not\models_\rho \varphi$.

We define both $\models$ and $\not\models$ so as to avoid negative occurrences of $\models$ in its own definition.

$$\frac{\rho_s(x_s) = a}{[\![x_s]\!]_\rho = a}$$

$$\frac{[\![t_1]\!]_\rho = a_1 \quad \cdots \quad [\![t_n]\!]_\rho = a_n \qquad f^M(a_1, \ldots, a_n) = a}{[\![f_{ws}\langle t_1, \ldots, t_n\rangle]\!]_\rho = a}$$

According to this rule, the value of $f_{ws}\langle t_1, \ldots, t_n\rangle$ is defined only if the values of $t_1, \ldots, t_n$ are defined and the resulting tuple of values is in $Dom(f^M)$.

$$\frac{M \models_\rho \varphi \qquad [\![t]\!]_\rho = a}{[\![\varphi \to t \mid t']\!]_\rho = a} \qquad\qquad \frac{M \not\models_\rho \varphi \qquad [\![t']\!]_\rho = a'}{[\![\varphi \to t \mid t']\!]_\rho = a'}$$

$$\frac{[\![t_1]\!]_\rho = a_1 \quad \cdots \quad [\![t_n]\!]_\rho = a_n \qquad (a_1, \ldots, a_n) \in p^M}{M \models_\rho p_w\langle t_1, \ldots, t_n\rangle}$$

$$\frac{[\![t_j]\!]_\rho \text{ not defined for some } 1 \leq j \leq n}{M \not\models_\rho p_w\langle t_1, \ldots, t_n\rangle}$$

$$\frac{[\![t_1]\!]_\rho = a_1 \quad \cdots \quad [\![t_n]\!]_\rho = a_n \qquad (a_1, \ldots, a_n) \notin p^M}{M \not\models_\rho p_w\langle t_1, \ldots, t_n\rangle}$$

$$\frac{[\![t]\!]_\rho = a \qquad [\![t']\!]_\rho = a}{M \models_\rho t \overset{\text{e}}{=} t'}$$

$$\frac{[\![t]\!]_\rho \text{ not defined}}{M \not\models_\rho t \overset{\text{e}}{=} t'} \qquad\qquad \frac{[\![t']\!]_\rho \text{ not defined}}{M \not\models_\rho t \overset{\text{e}}{=} t'}$$

$$\frac{[\![t]\!]_\rho = a \qquad [\![t']\!]_\rho = a' \qquad a \neq a'}{M \not\models_\rho t \overset{\text{e}}{=} t'}$$

$$\frac{}{M \not\models_\rho \mathit{false}}$$

$$\frac{M \not\models_\rho \varphi}{M \models_\rho \varphi \Rightarrow \varphi'} \qquad \frac{M \models_\rho \varphi'}{M \models_\rho \varphi \Rightarrow \varphi'}$$

$$\frac{M \models_\rho \varphi \qquad M \not\models_\rho \varphi'}{M \not\models_\rho \varphi \Rightarrow \varphi'}$$

$$\frac{M \models_{\rho[x_s \mapsto a]} \varphi \text{ for all } a \in s^M}{M \models_\rho \forall x_s . \varphi}$$

$$\frac{a \in s^M \qquad M \not\models_{\rho[x_s \mapsto a]} \varphi}{M \not\models_\rho \forall x_s . \varphi}$$

Proposition 2.15. *$M \models_\rho \varphi$ iff $\neg(M \not\models_\rho \varphi)$.*

Proof. By induction on the structure of φ. □

A sort-generation constraint (S', F') is satisfied in a Σ-model M if the carriers of the sorts in S' are *generated* by the function symbols in F' from the values in the carriers of sorts not in S'. Then $M \models (S', F', \sigma)$ iff $M|_\sigma \models (S', F')$.

Suppose M is a Σ-model and (S', F', σ) is a Σ-constraint with $\sigma : \overline{\Sigma} \to \Sigma$. Then M *satisfies* (S', F', σ), written $M \models (S', F', \sigma)$, if the carriers of $M|_\sigma$ of the sorts in S' are generated by the function symbols in F', i.e. for every sort $s \in S'$ and every value $a \in s^{M|_\sigma}$, there is a $\overline{\Sigma}$-term t containing only function symbols from F' and variables of sorts not in S', and no conditional subterms (terms of the form $\varphi \to t' \mid t''$), such that $[\![t]\!]_\rho = a$ for some assignment ρ into $M|_\sigma$.

A Σ-model M *satisfies* a Σ-sentence ψ, written $M \models \psi$, if:

- ψ is a formula φ and $M \models_\emptyset \varphi$, where $\emptyset$ is here the empty assignment from the empty set of variables
- ψ is a constraint (S', F', σ) and $M \models (S', F', \sigma)$

We write $M \not\models \psi$ for $\neg(M \models \psi)$.

Proposition 2.16. *Satisfaction is compatible with reducts of models and translation of sentences: if $\sigma : \Sigma \to \Sigma'$ is a signature morphism, ψ is a Σ-sentence and M' is a Σ'-model, then*

$$M'|_\sigma \models \psi \qquad \textit{iff} \qquad M' \models \sigma(\psi)$$

Proof. See Sect. 3.1 of [41]. □

Theorem 2.17. Sig, **Mod**, **Sen** *and $\models$ form an institution [20].* **Sig** *is finitely cocomplete and* **Mod** *supports amalgamation of models and homomorphisms.*

Proof. Directly from Props. 2.3, 2.10, 2.14 and 2.16. □

Proposition 2.18. *Satisfaction is preserved and reflected by isomorphisms: if M, M' are Σ-models such that $M \cong M'$ and ψ is a Σ-sentence, then $M \models \psi$ iff $M' \models \psi$.*

Proof. Straightforward. □

The rest of this chapter gives the abstract syntax of the constructs of many-sorted basic specifications, and defines their intended interpretation. Well-formedness of phrases of the abstract syntax is defined by the *static semantics.* The *model semantics*, which yields a class of models as result, provides the corresponding model-theoretic part of the semantics. In this chapter, only basic specifications themselves (phrases of type BASIC-SPEC) are given both static and model semantics; other phrase types are given only static semantics. In this particular case, the result of the static semantics fully determines the result of the model semantics, but that is not the case in other parts of CASL.

2.2 Basic Items

A many-sorted basic specification BASIC-SPEC is a sequence of BASIC-ITEMS constructs. It determines an enrichment containing the sorts, function symbols, predicate symbols and axioms that belong to the specification; these may make reference to symbols in the local environment. This enrichment in turn determines a class of models.

```
BASIC-SPEC ::= basic-spec BASIC-ITEMS*
```

$$\boxed{\Sigma \vdash \texttt{BASIC-SPEC} \rhd (\Delta, \Psi) \qquad\qquad \Sigma, \mathcal{M} \vdash \texttt{BASIC-SPEC} \Rightarrow \mathcal{M}'}$$

(Δ, Ψ) is an enrichment relative to Σ. $\mathcal{M}$ is required to be a model class over Σ. Each model in $\mathcal{M}'$ is a valid $\Sigma \cup \Delta$-model that extends a model in $\mathcal{M}$ and satisfies Ψ.

As will become clear in Chap. 4, one use of basic specifications in CASL is in extending existing specifications. Such a basic specification will often make reference to the sorts, function symbols and predicate symbols of the existing specification (the *local environment*), for instance to declare a new function taking an argument of an existing sort. This context is captured by the signature Σ in the above judgements, with the Σ-models in $\mathcal{M}$ giving all the possible interpretations of these symbols. In contrast, variable declarations are local to basic specifications.

$$\frac{\begin{array}{c} \Sigma, \emptyset \vdash \texttt{BASIC-ITEMS}_1 \rhd (\Delta_1, \Psi_1), X_1 \\ \cdots \\ \Sigma \cup \Delta_1 \cup \cdots \cup \Delta_{n-1}, X_1 + \cdots + X_{n-1} \vdash \texttt{BASIC-ITEMS}_n \rhd (\Delta_n, \Psi_n), X_n \end{array}}{\begin{array}{r} \Sigma \vdash \texttt{basic-spec BASIC-ITEMS}_1 \ \ldots \ \texttt{BASIC-ITEMS}_n \rhd \\ (\Delta_1 \cup \cdots \cup \Delta_n, \Psi_1 \cup \cdots \cup \Psi_n) \end{array}}$$

Making the incremental information from all the preceding `BASIC-ITEMS` available to the next one in sequence gives linear visibility. The use of + to combine variable sets means that a later declaration of a given variable will override an earlier declaration of the same variable.

$$\frac{\Sigma \vdash \texttt{basic-spec BASIC-ITEMS*} \rhd (\Delta, \Psi)}{\begin{array}{l}\Sigma, \mathcal{M} \vdash \texttt{basic-spec BASIC-ITEMS*} \Rightarrow \\ \qquad \{(\Sigma \cup \Delta)\text{-model } M' \mid M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta} \in \mathcal{M} \text{ and } M' \models \psi \text{ for all } \psi \in \Psi\}\end{array}}$$

Each `BASIC-ITEMS` construct determines part of a signature and/or some sentences (except for `VAR-ITEMS`, which merely declares some global variables). There is *linear visibility* of declared symbols and variables in a list of `BASIC-ITEMS` constructs, except within a list of datatype declarations. Verbatim repetition of the declaration of a symbol is allowed, and does not affect the semantics.

```
BASIC-ITEMS ::= SIG-ITEMS | FREE-DATATYPE | SORT-GEN
              | VAR-ITEMS | LOCAL-VAR-AXIOMS | AXIOM-ITEMS
```

$$\boxed{\Sigma, X \vdash \texttt{BASIC-ITEMS} \rhd (\Delta, \Psi), X'}$$

X is required to be a valid set of variables over the sorts of Σ. (Δ, Ψ) is an enrichment relative to Σ, and X' is a valid set of variables over the sorts of $\Sigma \cup \Delta$. (Actually, X' will be a valid set of variables over the sorts of Σ since there happens to be no construct of `BASIC-ITEMS` that both declares variables and introduces signature components.)

$$\frac{\Sigma \vdash \texttt{SIG-ITEMS} \rhd (\Delta, \Delta', \Psi)}{\Sigma, X \vdash \texttt{SIG-ITEMS} \,\text{qua}\, \texttt{BASIC-ITEMS} \rhd (\Delta \cup \Delta', \Psi), \emptyset}$$

$$\frac{\Sigma \vdash \texttt{FREE-DATATYPE} \rhd (\Delta, \Psi)}{\Sigma, X \vdash \texttt{FREE-DATATYPE} \,\text{qua}\, \texttt{BASIC-ITEMS} \rhd (\Delta, \Psi), \emptyset}$$

$$\frac{\Sigma \vdash \texttt{SORT-GEN} \rhd (\Delta, \Psi)}{\Sigma, X \vdash \texttt{SORT-GEN} \,\text{qua}\, \texttt{BASIC-ITEMS} \rhd (\Delta, \Psi), \emptyset}$$

$$\frac{S \vdash \texttt{VAR-ITEMS} \rhd X'}{(S, \mathit{TF}, \mathit{PF}, P), X \vdash \texttt{VAR-ITEMS} \,\text{qua}\, \texttt{BASIC-ITEMS} \rhd (\emptyset, \emptyset), X'}$$

$$\frac{\Sigma, X \vdash \texttt{LOCAL-VAR-AXIOMS} \rhd \Psi}{\Sigma, X \vdash \texttt{LOCAL-VAR-AXIOMS} \,\text{qua}\, \texttt{BASIC-ITEMS} \rhd (\emptyset, \Psi), \emptyset}$$

$$\frac{\Sigma, X \vdash \texttt{AXIOM-ITEMS} \rhd \Psi}{\Sigma, X \vdash \texttt{AXIOM-ITEMS} \,\text{qua}\, \texttt{BASIC-ITEMS} \rhd (\emptyset, \Psi), \emptyset}$$

2.3 Signature Declarations

A list SORT-ITEMS of sort declarations determines some sorts. A list OP-ITEMS of operation declarations/definitions determines some operation symbols, and possibly some sentences; similarly for predicate declarations/definitions PRED-ITEMS. A list DATATYPE-ITEMS of datatype declarations determines some sorts together with some constructor and (optional) selector operations, and sentences defining the selector operations.

```
SIG-ITEMS ::= SORT-ITEMS | OP-ITEMS | PRED-ITEMS | DATATYPE-ITEMS
```

$$\boxed{\Sigma \vdash \texttt{SIG-ITEMS} \rhd (\Delta, \Delta', \Psi)}$$

$(\Delta \cup \Delta', \Psi)$ is an enrichment relative to Σ.

Here, Δ' are the selectors declared by DATATYPE-DECLs in SIG-ITEMS and Δ is everything else declared in SIG-ITEMS. These need to be kept separate here because they are treated differently by the sort-generation construct, see Sect. 2.3.5.

$$\frac{\Sigma \vdash \texttt{SORT-ITEMS} \rhd (\Delta, \Psi)}{\Sigma \vdash \texttt{SORT-ITEMS}\,\text{qua}\,\texttt{SIG-ITEMS} \rhd (\Delta, \emptyset, \Psi)}$$

$$\frac{\Sigma \vdash \texttt{OP-ITEMS} \rhd (\Delta, \Psi)}{\Sigma \vdash \texttt{OP-ITEMS}\,\text{qua}\,\texttt{SIG-ITEMS} \rhd (\Delta, \emptyset, \Psi)}$$

$$\frac{\Sigma \vdash \texttt{PRED-ITEMS} \rhd (\Delta, \Psi)}{\Sigma \vdash \texttt{PRED-ITEMS}\,\text{qua}\,\texttt{SIG-ITEMS} \rhd (\Delta, \emptyset, \Psi)}$$

$$\frac{\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W}{\Sigma \vdash \texttt{DATATYPE-ITEMS}\,\text{qua}\,\texttt{SIG-ITEMS} \rhd (\Delta, \Delta', \Psi)}$$

2.3.1 Sorts

```
SORT-ITEMS ::= sort-items SORT-ITEM+
SORT-ITEM  ::= SORT-DECL
```

$$\boxed{\Sigma \vdash \texttt{SORT-ITEMS} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\Sigma \vdash \texttt{SORT-ITEM}_1 \rhd (\Delta_1, \Psi_1) \qquad \cdots \qquad \Sigma \vdash \texttt{SORT-ITEM}_n \rhd (\Delta_n, \Psi_n)}{\begin{array}{l}\Sigma \vdash \texttt{sort-items SORT-ITEM}_1 \ldots \texttt{SORT-ITEM}_n \rhd \\ \qquad\qquad (\Delta_1 \cup \cdots \cup \Delta_n, \Psi_1 \cup \cdots \cup \Psi_n)\end{array}}$$

The only reason why we have $\Sigma \vdash \texttt{SORT-ITEMS} \rhd (\Delta, \Psi)$ rather than simply $\vdash \texttt{SORT-ITEMS} \rhd S$ (and similarly for $\texttt{SORT-ITEM}$ below) is to accommodate the extension to subsorts in Chap. 3 where Δ will include a subsorting relation and Ψ will include axioms for defined subsorts.

$$\boxed{\Sigma \vdash \texttt{SORT-ITEM} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\vdash \texttt{SORT-DECL} \rhd S}{\Sigma \vdash \texttt{SORT-DECL}\ \text{qua}\ \texttt{SORT-ITEM} \rhd ((S, \emptyset, \emptyset, \emptyset), \emptyset)}$$

Sort Declarations

$$\boxed{\text{A sort declaration } \texttt{SORT-DECL} \text{ declares each of the sorts given.}}$$

```
SORT-DECL ::= sort-decl SORT+
SORT      ::= SORT-ID
```

$$\boxed{\vdash \texttt{SORT-DECL} \rhd S}$$

$$\frac{}{\vdash \texttt{sort-decl}\ s_1\ \ldots\ s_n \rhd \{s_1, \ldots, s_n\}}$$

As promised in Sect. 2.1.1, we now define the universe *Sort* of sort names.

$$\mathit{Sort} = \texttt{SORT-ID}$$

2.3.2 Operations

```
OP-ITEMS ::= op-items OP-ITEM+
OP-ITEM  ::= OP-DECL | OP-DEFN
```

$$\boxed{\Sigma \vdash \texttt{OP-ITEMS} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{OP-ITEM}_1 \rhd (\Delta_1, \Psi_1) \\ \cdots \\ \Sigma \cup \Delta_1 \cup \cdots \cup \Delta_{n-1} \vdash \texttt{OP-ITEM}_n \rhd (\Delta_n, \Psi_n)\end{array}}{\Sigma \vdash \texttt{op-items}\ \texttt{OP-ITEM}_1\ \ldots\ \texttt{OP-ITEM}_n \rhd (\Delta_1 \cup \cdots \cup \Delta_n, \Psi_1 \cup \cdots \cup \Psi_n)}$$

Making the signature extensions from all the preceding $\texttt{OP-ITEM}$s available to the next one in sequence gives linear visibility. This is required here for the sake of `UNIT-OP-ATTR` attributes and operation definitions, both of which may refer to previously-declared function symbols.

$$\boxed{\Sigma \vdash \texttt{OP-ITEM} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

Rules elided (see Sect. 1.3).

Operation Declarations

An operation declaration `OP-DECL` declares each given operation name as a total or partial operation, with profile as specified, and having the given attributes. If an operation is declared both as total and as partial with the same profile, the resulting signature only contains the total operation.

```
OP-DECL         ::= op-decl OP-NAME+ OP-TYPE OP-ATTR*
OP-NAME         ::= ID
OP-TYPE         ::= TOTAL-OP-TYPE | PARTIAL-OP-TYPE
TOTAL-OP-TYPE   ::= total-op-type SORT-LIST SORT
PARTIAL-OP-TYPE ::= partial-op-type SORT-LIST SORT
SORT-LIST       ::= sort-list SORT*
```

As promised in Sect. 2.1.1, we now define the universe *FunName* of operation names.

$$\mathit{FunName} = \texttt{ID}$$

(Recall from Sect. 2.1.1 that operations are also referred to as functions, hence *FunName*.)

$$\boxed{\Sigma \vdash \texttt{OP-DECL} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c} (S, \mathit{TF}, \mathit{PF}, P) = \Sigma \qquad \mathit{ws} = (\langle s_1, \ldots, s_m \rangle, s) \\ \{s_1, \ldots, s_m, s\} \subseteq S \qquad \Delta = (\emptyset, \{\mathit{ws} \mapsto \{f^1, \ldots, f^n\}\}, \emptyset, \emptyset) \\ \begin{array}{ccc} \Sigma \cup \Delta, f^1_{\mathit{ws}} \vdash \texttt{OP-ATTR}_1 \rhd \Psi_{11} & \cdots & \Sigma \cup \Delta, f^n_{\mathit{ws}} \vdash \texttt{OP-ATTR}_1 \rhd \Psi_{n1} \\ \cdots & & \cdots \\ \Sigma \cup \Delta, f^1_{\mathit{ws}} \vdash \texttt{OP-ATTR}_p \rhd \Psi_{1p} & \cdots & \Sigma \cup \Delta, f^n_{\mathit{ws}} \vdash \texttt{OP-ATTR}_p \rhd \Psi_{np} \end{array} \end{array}}{\begin{array}{l} \Sigma \vdash \texttt{op-decl}\ f^1 \ldots f^n \\ \qquad (\texttt{total-op-type}\,(\texttt{sort-list}\ s_1 \ldots s_m)\ s) \\ \qquad \texttt{OP-ATTR}_1 \ldots \texttt{OP-ATTR}_p \rhd \\ \qquad\qquad (\Delta, (\Psi_{11} \cup \cdots \cup \Psi_{n1}) \cup \cdots \cup (\Psi_{1p} \cup \cdots \cup \Psi_{np})) \end{array}}$$

$$\frac{\begin{array}{c} (S, \mathit{TF}, \mathit{PF}, P) = \Sigma \qquad ws = (\langle s_1, \ldots, s_m \rangle, s) \\ \{s_1, \ldots, s_m, s\} \subseteq S \qquad \Delta = (\emptyset, \emptyset, \{ws \mapsto \{f^1, \ldots, f^n\}\}, \emptyset) \\ \Sigma \cup \Delta, f^1_{ws} \vdash \texttt{OP-ATTR}_1 \rhd \Psi_{11} \qquad \cdots \qquad \Sigma \cup \Delta, f^n_{ws} \vdash \texttt{OP-ATTR}_1 \rhd \Psi_{n1} \\ \cdots \qquad\qquad\qquad\qquad\qquad\qquad \cdots \\ \Sigma \cup \Delta, f^1_{ws} \vdash \texttt{OP-ATTR}_p \rhd \Psi_{1p} \qquad \cdots \qquad \Sigma \cup \Delta, f^n_{ws} \vdash \texttt{OP-ATTR}_p \rhd \Psi_{np} \end{array}}{\begin{array}{l} \Sigma \vdash \texttt{op-decl}\ f^1 \ \ldots\ f^n \\ \qquad (\texttt{partial-op-type}\,(\texttt{sort-list}\ s_1 \ \ldots\ s_m)\ s) \\ \qquad \texttt{OP-ATTR}_1 \ \ldots\ \texttt{OP-ATTR}_p \rhd \\ \qquad\qquad (\Delta, (\Psi_{11} \cup \cdots \cup \Psi_{n1}) \cup \cdots \cup (\Psi_{1p} \cup \cdots \cup \Psi_{np})) \end{array}}$$

The use of $\cup$ to combine the extensions produced by these rules, in the rules for `OP-ITEMS` and `BASIC-SPEC`, ensures that when an operation is declared both as total and as partial with the same profile, the resulting signature only contains the total operation. This is the purpose of the *reconcile* function in the definition of union, see Sect. 2.1.1.

Operation Attributes

Operation attributes assert that the operations being declared (which must be binary) have certain common properties: *associativity*, *commutativity*, *idempotency* and/or having a *unit*. (This can also be used to add attributes to operations that have previously been declared without them.)

```
OP-ATTR         ::= BINARY-OP-ATTR | UNIT-OP-ATTR
BINARY-OP-ATTR ::= assoc-op-attr | comm-op-attr | idem-op-attr
UNIT-OP-ATTR   ::= unit-op-attr TERM
```

$$\boxed{\Sigma, f_{ws} \vdash \texttt{OP-ATTR} \rhd \Psi}$$

f_{ws} is required to be a qualified function name over Σ. Ψ is a set of Σ-sentences.

$$\frac{ws = (\langle s, s \rangle, s)}{\begin{array}{l} \Sigma, f_{ws} \vdash \texttt{assoc-op-attr} \rhd \\ \qquad \{\forall x_s . \forall y_s . \forall z_s . f_{ws} \langle x_s, f_{ws} \langle y_s, z_s \rangle \rangle \overset{\mathrm{s}}{=} f_{ws} \langle f_{ws} \langle x_s, y_s \rangle, z_s \rangle \} \end{array}}$$

$$\frac{ws = (\langle s, s \rangle, s')}{\Sigma, f_{ws} \vdash \texttt{comm-op-attr} \rhd \{\forall x_s . \forall y_s . f_{ws} \langle x_s, y_s \rangle \overset{\mathrm{s}}{=} f_{ws} \langle y_s, x_s \rangle \}}$$

$$\frac{ws = (\langle s, s \rangle, s)}{\Sigma, f_{ws} \vdash \texttt{idem-op-attr} \rhd \{\forall x_s . f_{ws} \langle x_s, x_s \rangle \overset{\mathrm{s}}{=} x_s \}}$$

$$\frac{ws = (\langle s, s \rangle, s) \qquad \Sigma, \emptyset \vdash F \rhd \varphi \qquad \Sigma, \emptyset \vdash F' \rhd \varphi'}{\Sigma, f_{ws} \vdash \texttt{unit-op-attr}\ \texttt{TERM} \rhd \{\varphi, \varphi'\}}$$

where F is

```
quantification universal
  (var-decl x s)
  (strong-equation
    (application  (qual-op-name f (total-op-type (sort-list s s) s))
                  (terms TERM (qual-var x s)))
    (qual-var x s))
```

and F' is

```
quantification universal
  (var-decl x s)
  (strong-equation
    (qual-var x s)
    (application (qual-op-name f (total-op-type (sort-list s s) s))
                 (terms TERM (qual-var x s))))
```

if $f \in TF_{ws}$, for $(S, TF, PF, P) = \Sigma$, and similarly with `partial-op-type` in place of `total-op-type` if $f \in PF_{ws}$.

This rule is more complicated than those for the other forms of operation attribute because the term supplied may be ambiguous due to the presence of overloaded operations that cannot be resolved. The static semantics of formulas in Sect. 2.5 below yields a result only when this is not the case; otherwise the attribute is ill-formed.

Operation Definitions

> A total or partial operation may be *defined* at the same time as it is declared, by giving its value (when applied to a list of argument variables) as a term. The operation name may occur in the term, and may have *any* interpretation satisfying the equation – not necessarily the least fixed point.

```
OP-DEFN         ::= op-defn OP-NAME OP-HEAD TERM
OP-HEAD         ::= TOTAL-OP-HEAD | PARTIAL-OP-HEAD
TOTAL-OP-HEAD   ::= total-op-head ARG-DECL* SORT
PARTIAL-OP-HEAD ::= partial-op-head ARG-DECL* SORT
ARG-DECL        ::= arg-decl VAR+ SORT
```

$$\boxed{\Sigma \vdash \texttt{OP-DEFN} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \qquad S \vdash \texttt{ARG-DECL*} \rhd \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle \\ w = \langle s_1, \ldots, s_n \rangle \qquad s \in S \\ \Delta = (\emptyset, \{(w, s) \mapsto \{f\}\}, \emptyset, \emptyset) \qquad X = \mathit{complete}(\{x^1_{s_1}, \ldots, x^n_{s_n}\}, S) \\ \Sigma \cup \Delta, X \vdash F \rhd t \overset{\mathrm{s}}{=} t'\end{array}}{\Sigma \vdash \texttt{op-defn}\, f\, (\texttt{total-op-head ARG-DECL*}\ s)\, \texttt{TERM} \rhd (\Delta, \{\forall X . t \overset{\mathrm{s}}{=} t'\})}$$

where F is

$$\begin{array}{l}
\texttt{strong-equation} \\
\quad (\texttt{application} \\
\qquad (\texttt{qual-op-name}\ f\ (\texttt{total-op-type}\ (\texttt{sort-list}\ s_1 \dots s_n)\ s)) \\
\qquad (\texttt{terms}\ (\texttt{qual-var}\ x^1\ \ s_1) \dots (\texttt{qual-var}\ x^n\ \ s_n))) \\
\quad \texttt{TERM}
\end{array}$$

$$\frac{\begin{array}{c}(S, \mathit{TF}, \mathit{PF}, P) = \Sigma \qquad S \vdash \texttt{ARG-DECL*} \rhd \langle x^1_{s_1}, \dots, x^n_{s_n} \rangle \\ w = \langle s_1, \dots, s_n \rangle \qquad s \in S \\ \Delta = (\emptyset, \emptyset, \{(w, s) \mapsto \{f\}\}, \emptyset) \qquad X = \mathit{complete}(\{x^1_{s_1}, \dots, x^n_{s_n}\}, S) \\ \Sigma \cup \Delta, X \vdash F \rhd t \overset{\mathrm{s}}{=} t' \end{array}}{\Sigma \vdash \texttt{op-defn}\, f\ (\texttt{partial-op-head ARG-DECL*}\ \ s)\ \texttt{TERM} \rhd (\Delta, \{\forall X . t \overset{\mathrm{s}}{=} t'\})}$$

where F is

$$\begin{array}{l}
\texttt{strong-equation} \\
\quad (\texttt{application} \\
\qquad (\texttt{qual-op-name}\ f\ (\texttt{partial-op-type}\ (\texttt{sort-list}\ s_1 \dots s_n)\ s)) \\
\qquad (\texttt{terms}\ (\texttt{qual-var}\ x^1\ \ s_1) \dots (\texttt{qual-var}\ x^n\ \ s_n))) \\
\quad \texttt{TERM}
\end{array}$$

$$\boxed{S \vdash \texttt{ARG-DECL*} \rhd \langle x^1_{s_1}, \dots, x^n_{s_n} \rangle}$$

Each $x^i_{s_i}$ is a qualified variable name over S, and $x^i \neq x^j$ for all $1 \leq i \neq j \leq n$.

$$\frac{\begin{array}{c} S \vdash \texttt{ARG-DECL}_1 \rhd \langle x^{11}, \dots, x^{1m_1} \rangle, s_1 \\ \dots \\ S \vdash \texttt{ARG-DECL}_p \rhd \langle x^{p1}, \dots, x^{pm_p} \rangle, s_p \\ \{x^{i1}, \dots, x^{im_i}\} \cap \{x^{j1}, \dots, x^{jm_j}\} = \emptyset \text{ for all } 1 \leq i \neq j \leq p \end{array}}{S \vdash \texttt{ARG-DECL}_1\ \dots\ \texttt{ARG-DECL}_p \rhd \langle x^{11}_{s_1}, \dots, x^{1m_1}_{s_1},\ \dots,\ x^{p1}_{s_p}, \dots, x^{pm_p}_{s_p} \rangle}$$

$$\boxed{S \vdash \texttt{ARG-DECL} \rhd \langle x_1, \dots, x_n \rangle, s}$$

s is a sort in S and $x_i \neq x_j$ for all $1 \leq i \neq j \leq n$.

$$\frac{s \in S \qquad x_i \neq x_j \text{ for all } 1 \leq i \neq j \leq n}{S \vdash \texttt{arg-decl}\ x_1\ \dots\ x_n\ \ s \rhd \langle x_1, \dots, x_n \rangle, s}$$

2.3.3 Predicates

```
PRED-ITEMS ::= pred-items PRED-ITEM+
PRED-ITEM  ::= PRED-DECL | PRED-DEFN
PRED-NAME  ::= ID
```

$$\boxed{\Sigma \vdash \texttt{PRED-ITEMS} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{PRED-ITEM}_1 \rhd (\Delta_1, \Psi_1) \\ \cdots \\ \Sigma \cup \Delta_1 \cup \cdots \cup \Delta_{n-1} \vdash \texttt{PRED-ITEM}_n \rhd (\Delta_n, \Psi_n)\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{pred-items PRED-ITEM}_1 \texttt{ ... PRED-ITEM}_n \rhd \\ \qquad\qquad\qquad (\Delta_1 \cup \cdots \cup \Delta_n, \Psi_1 \cup \cdots \cup \Psi_n)\end{array}}$$

Making the signature extensions from all the preceding `PRED-ITEM`s available to the next one in sequence gives linear visibility. This is required here for the sake of predicate definitions which may refer to previously-declared predicate symbols.

$$\boxed{\Sigma \vdash \texttt{PRED-ITEM} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\Sigma \vdash \texttt{PRED-DECL} \rhd \Delta}{\Sigma \vdash \texttt{PRED-DECL}\,\text{qua}\,\texttt{PRED-ITEM} \rhd (\Delta, \emptyset)}$$

Rule for `PRED-DEFN` qua `PRED-ITEM` elided.

As promised in Sect. 2.1.1, we now define the universe *PredName* of predicate names.

$$\mathit{PredName} = \texttt{ID}$$

Predicate Declarations

A predicate declaration `PRED-DECL` declares each given predicate name, with profile as specified.

```
PRED-DECL ::= pred-decl PRED-NAME+ PRED-TYPE
```

$$\boxed{\Sigma \vdash \texttt{PRED-DECL} \rhd \Delta}$$

Δ is a signature extension relative to Σ.

$$\frac{S \vdash \texttt{PRED-TYPE} \rhd w}{\begin{array}{l}(S, \mathit{TF}, \mathit{PF}, P) \vdash \texttt{pred-decl}\ p_1 \ \ldots \ p_n \ \texttt{PRED-TYPE} \rhd \\ \qquad\qquad\qquad (\emptyset, \emptyset, \emptyset, \{w \mapsto \{p_1, \ldots, p_n\}\})\end{array}}$$

Predicate Types

```
PRED-TYPE ::= pred-type SORT-LIST
```

$$\boxed{S \vdash \texttt{PRED-TYPE} \rhd w}$$

All the sorts in w are in S.

$$\frac{\{s_1, \ldots, s_n\} \subseteq S}{S \vdash \texttt{pred-type}\,(\texttt{sort-list}\, s_1\ \ldots\ s_n) \rhd \langle s_1, \ldots, s_n \rangle}$$

Predicate Definitions

A predicate may be *defined* at the same time as it is declared, by asserting its equivalence with a formula. The predicate name may occur in the formula, and may have *any* interpretation satisfying the equivalence.

```
PRED-DEFN ::= pred-defn PRED-NAME PRED-HEAD FORMULA
PRED-HEAD ::= pred-head ARG-DECL*
```

$$\boxed{\Sigma \vdash \texttt{PRED-DEFN} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{rl} (S, \mathit{TF}, \mathit{PF}, P) = \Sigma & \quad S \vdash \texttt{ARG-DECL*} \rhd \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle \\ w = \langle s_1, \ldots, s_n \rangle & \quad \Delta = (\emptyset, \emptyset, \emptyset, \{w \mapsto \{p\}\}) \\ X = \mathit{complete}(\{x^1_{s_1}, \ldots, x^n_{s_n}\}, S) & \quad \Sigma \cup \Delta, X \vdash \texttt{FORMULA} \rhd \varphi \end{array}}{\begin{array}{r} \Sigma \vdash \texttt{pred-defn}\ p\ (\texttt{pred-head ARG-DECL*})\ \texttt{FORMULA} \rhd \\ (\Delta, \{\forall X . p_w \langle x^1_{s_1}, \ldots, x^n_{s_n} \rangle \Leftrightarrow \varphi\}) \end{array}}$$

2.3.4 Datatypes

The order of the datatype declarations in a list DATATYPE-ITEMS is *not* significant: there is *non-linear visibility* of the declared sorts. A list of datatype declarations must not declare a function symbol both as a constructor and selector with the same profiles.

```
DATATYPE-ITEMS ::= datatype-items DATATYPE-DECL+
```

The semantics of datatype declarations is by far the most complicated part of the semantics of basic specifications. Before proceeding, here is an overview. Some examples of the results produced for free datatypes are given just before Sect. 2.3.5; these should be helpful in understanding that part of the semantics, and working backwards to see why these results are produced should help in clarifying the semantics of non-free datatypes.

The main judgements are $\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W$ and $\Sigma \vdash \texttt{FREE-DATATYPE} \rhd (\Delta, \Psi)$. The former, for a list of datatype declarations, is subordinate to the latter, for free datatypes. It is also subordinate to the judgement for `SIG-ITEMS`, when used to declare non-free datatypes. All of the information in its result is required to determine the semantics of free datatypes but some is not required in the case of non-free datatypes. The judgements that are subordinate to `DATATYPE-ITEMS` collect information about declared sorts, constructors and selectors and check that various restrictions are satisfied. A complicating factor in these is non-linear visibility at the `DATATYPE-ITEMS` level.

Metavariables are used consistently in the judgement for `DATATYPE-ITEMS` and all of its subordinate judgements. Here is a summary of what they stand for, where these results are formed, and where and for what they are required.

1. Δ contains the sorts and constructors declared by the list of datatype declarations. It is formed by the rules for `DATATYPE-DECL` (sorts) and `ALTERNATIVE` (constructors).
2. Δ' contains the declared selectors. It is formed by the rules for `COMPONENTS`. The selectors need to be kept separate from the other signature components for the sake of the disjointness condition in the `DATATYPE-ITEMS` rule, to generate sentences in the rule for `FREE-DATATYPE`, and, in the case of a non-free datatypes, to produce the result for `SIG-ITEMS`, where a separation is required for the sake of `SORT-GEN` where selectors receive special treatment.
3. Ψ contains sentences defining the value of each selector on the values produced by the corresponding constructor. It is formed by the rules for `COMPONENTS`.
4. W is a finite map taking each constructor name in Δ to the corresponding set of partial selectors from Δ' (or to $\emptyset$ in case there are none). It is formed in the rules for `ALTERNATIVE`s. W is needed in the rule for `FREE-DATATYPE` to generate sentences that require a partial selector to return an undefined result when applied to a value produced by a constructor for which it has not been declared.

$$\boxed{\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W}$$

$(\Delta \cup \Delta', \Psi)$ is an enrichment relative to Σ and W is a finite map taking qualified function names over $\Sigma \cup \Delta$ from Δ to sets of qualified function names over $\Sigma \cup \Delta \cup \Delta'$ from Δ'.

$$\frac{\begin{array}{c}\Sigma' \vdash \texttt{DATATYPE-DECL}_1 \rhd (\Delta_1, \Delta'_1, \Psi_1), W_1 \\ \cdots \\ \Sigma' \vdash \texttt{DATATYPE-DECL}_n \rhd (\Delta_n, \Delta'_n, \Psi_n), W_n \\ \mathit{disjoint\text{-}functions}(\Delta_1 \cup \cdots \cup \Delta_n, \Delta'_1 \cup \cdots \cup \Delta'_n) \\ \Sigma' = \Sigma \cup \Delta_1 \cup \Delta'_1 \cup \cdots \cup \Delta_n \cup \Delta'_n\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{datatype-items DATATYPE-DECL}_1 \ \ldots \ \texttt{DATATYPE-DECL}_n \rhd \\ \qquad (\Delta_1 \cup \cdots \cup \Delta_n, \Delta'_1 \cup \cdots \cup \Delta'_n, \Psi_1 \cup \cdots \cup \Psi_n), W_1 \cup \cdots \cup W_n\end{array}}$$

where *disjoint-functions*$((S, TF, PF, P), (S', TF', PF', P'))$ means

$$\begin{array}{l}\text{for all } ws \in \mathit{Dom}(TF \cup PF) \cap \mathit{Dom}(TF' \cup PF'), \\ (TF \cup PF)(ws) \cap (TF' \cup PF')(ws) = \emptyset\end{array}$$

The 'recursion' in the premises of this rule is what provides non-linear visibility, making the order of the `DATATYPE-DECL`s not significant. In the subordinate judgements, it is important to remember that the context will already include the signature extensions being produced. The disjointness premise implements the requirement that a list of datatype declarations must not declare a function symbol both as a constructor and selector with the same profile.

Note that if a sort is declared several times in a `DATATYPE-ITEMS`, with several lists of alternatives, the effect is the same as if the sort had been declared only once, but with the union of the alternative lists.

Datatype Declarations

A datatype declaration `DATATYPE-DECL` declares the given sort, and for each given alternative construct the given constructor and selector operations, and determines sentences asserting the expected relationship between selectors and constructors.

```
DATATYPE-DECL ::= datatype-decl SORT ALTERNATIVE+
```

$$\boxed{\Sigma \vdash \texttt{DATATYPE-DECL} \rhd (\Delta, \Delta', \Psi), W}$$

$(\Delta \cup \Delta', \Psi)$ is an enrichment relative to Σ and W is a finite map taking qualified function names over $\Sigma \cup \Delta$ from Δ to sets of qualified function names over $\Sigma \cup \Delta \cup \Delta'$ from Δ'.

See the beginning of Sect. 2.3.4 for an explanation of the meaning of Δ, Δ', Ψ and W in this part of the semantics.

$$\frac{\begin{array}{c}\Sigma, s \vdash \texttt{ALTERNATIVE}_1 \rhd (\Delta_1, \Delta'_1, \Psi_1), W_1 \\ \cdots \\ \Sigma, s \vdash \texttt{ALTERNATIVE}_n \rhd (\Delta_n, \Delta'_n, \Psi_n), W_n\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{datatype-decl}\ s\ \texttt{ALTERNATIVE}_1\ \texttt{ALTERNATIVE}_2\ \ldots\ \texttt{ALTERNATIVE}_n \rhd \\ \quad ((\{s\}, \emptyset, \emptyset, \emptyset) \cup \Delta_1 \cup \cdots \cup \Delta_n, \Delta'_1 \cup \cdots \cup \Delta'_n, \Psi_1 \cup \cdots \cup \Psi_n), W_1 \cup \cdots \cup W_n\end{array}}$$

Note that s will be a sort in Σ because of non-linear visibility. If a constructor is declared several times for one sort in a `DATATYPE-DECL`, the effect is the same as if only one constructor had been declared. If these multiple declarations involved different selectors, all of them are provided with all selectors for a given argument position of the constructor being semantically equal.

Alternatives

An `ALTERNATIVE` declares a constructor operation. Each component specifies one or more argument sorts for the profile; the result sort is the one declared by the enclosing datatype declaration.

```
ALTERNATIVE         ::= TOTAL-CONSTRUCT | PARTIAL-CONSTRUCT
TOTAL-CONSTRUCT     ::= total-construct   OP-NAME COMPONENTS*
PARTIAL-CONSTRUCT ::= partial-construct OP-NAME COMPONENTS+
```

$$\boxed{\Sigma, s \vdash \texttt{ALTERNATIVE} \rhd (\Delta, \Delta', \Psi), W}$$

s is required to be a sort in Σ. $(\Delta \cup \Delta', \Psi)$ is an enrichment relative to Σ where Δ contains exactly one function and this function has result sort s, and W is a finite map taking this function to a set of qualified function names over $\Sigma \cup \Delta'$ from Δ'.

See the beginning of Sect. 2.3.4 for an explanation of the meaning of Δ, Δ', Ψ and W in this part of the semantics. In this judgement, s is the sort declared by the enclosing `DATATYPE-DECL`, the function in Δ is the constructor for this alternative, and Δ' are its selectors.

$$\frac{\begin{array}{c}
\Sigma, f, ws, 1 \vdash \texttt{COMPONENTS}_1 \rhd \langle s_{11}, \ldots, s_{1m_1} \rangle, (\Delta'_1, \Psi_1) \\
\cdots \\
\Sigma, f, ws, 1 + m_1 + \cdots + m_{n-1} \vdash \texttt{COMPONENTS}_n \rhd \langle s_{n1}, \ldots, s_{nm_n} \rangle, (\Delta'_n, \Psi_n) \\
\mathit{disjoint\text{-}functions}(\Delta'_1, \ldots, \Delta'_n) \\
ws = (\langle s_{11}, \ldots, s_{1m_1}, \ \ldots, \ s_{n1}, \ldots, s_{nm_n} \rangle, s) \\
(S, \mathit{TF}, \mathit{PF}, P) = \Sigma \qquad (S', \mathit{TF}', \mathit{PF}', P') = \Delta'_1 \cup \cdots \cup \Delta'_n
\end{array}}{\begin{array}{l}
\Sigma, s \vdash \texttt{total-construct} \ f \ \texttt{COMPONENTS}_1 \ \ldots \ \texttt{COMPONENTS}_n \rhd \\
\qquad ((\emptyset, \{ws \mapsto \{f\}\}, \emptyset, \emptyset), \Delta'_1 \cup \cdots \cup \Delta'_n, \Psi_1 \cup \cdots \cup \Psi_n), \\
\qquad \{f_{ws} \mapsto \{g_{\langle s \rangle, s'} \mid s' \in S, g \in \mathit{PF}'(\langle s \rangle, s')\}\}
\end{array}}$$

where $\mathit{disjoint\text{-}functions}((S_1, \mathit{TF}_1, \mathit{PF}_1, P_1), \ldots, (S_n, \mathit{TF}_n, \mathit{PF}_n, P_n))$ means

$$\begin{array}{l}
\text{for all } i, j \text{ and } ws \text{ such that } 1 \leq i \neq j \leq n \text{ and} \\
ws \in \mathit{Dom}(\mathit{TF}_i \cup \mathit{PF}_i) \cap \mathit{Dom}(\mathit{TF}_j \cup \mathit{PF}_j), \\
(\mathit{TF}_i \cup \mathit{PF}_i)(ws) \cap (\mathit{TF}_j \cup \mathit{PF}_j)(ws) = \emptyset
\end{array}$$

Note that f will be a total function in Σ because of non-linear visibility.

$$\frac{\begin{array}{c} \Sigma, f, ws, 1 \vdash \mathtt{COMPONENTS}_1 \rhd \langle s_{11},\ldots,s_{1m_1}\rangle, (\Delta'_1,\Psi_1) \\ \ldots \\ \Sigma, f, ws, 1+m_1+\cdots+m_{n-1} \vdash \mathtt{COMPONENTS}_n \rhd \langle s_{n1},\ldots,s_{nm_n}\rangle,(\Delta'_n,\Psi_n) \\ \mathit{disjoint\text{-}functions}(\Delta'_1,\ldots,\Delta'_n) \\ ws = (\langle s_{11},\ldots,s_{1m_1},\ \ldots,\ s_{n1},\ldots,s_{nm_n}\rangle, s) \\ (S,TF,PF,P)=\Sigma \qquad (S',TF',PF',P') = \Delta'_1\cup\cdots\cup\Delta'_n \end{array}}{\begin{array}{l} \Sigma, s \vdash \mathtt{partial\text{-}construct}\ f\ \mathtt{COMPONENTS}_1\ \ldots\ \mathtt{COMPONENTS}_n \rhd \\ \qquad ((\emptyset,\emptyset,\{ws\mapsto\{f\}\},\emptyset),\Delta'_1\cup\cdots\cup\Delta'_n,\Psi_1\cup\cdots\cup\Psi_n), \\ \qquad \{f_{ws}\mapsto\{g_{\langle s\rangle,s'} \mid s'\in S, g\in PF'(\langle s\rangle,s')\}\}\end{array}}$$

where *disjoint-functions* is defined as in the previous rule. Note that f will be a partial function in Σ because of non-linear visibility.

The disjointness premise in both rules implements the requirement that the selectors within each `ALTERNATIVE` must be distinct.

Components

Each `COMPONENTS` construct specifies one or more argument sorts for the constructor operation declared by the enclosing `ALTERNATIVE`, and optionally some selector operations with sentences determining their result on values produced by that constructor. All sorts used must be declared in the local environment.

```
COMPONENTS     ::= TOTAL-SELECT | PARTIAL-SELECT | SORT
TOTAL-SELECT   ::= total-select   OP-NAME+ SORT
PARTIAL-SELECT ::= partial-select OP-NAME+ SORT
```

$$\boxed{\Sigma, f, ws, m \vdash \mathtt{COMPONENTS} \rhd w', (\Delta', \Psi)}$$

f is required to be a function name in Σ with profile $ws = (\langle s_1,\ldots,s_n\rangle, s)$ and $1 \leq m \leq n$. w' is a non-empty sequence of sorts in Σ and (Δ', Ψ) is an enrichment relative to Σ.

See the beginning of Sect. 2.3.4 for an explanation of the meaning of Δ' and Ψ in this part of the semantics. In this judgement, f is the constructor declared by the enclosing `ALTERNATIVE`, s is the sort declared by the enclosing `DATATYPE-DECL`, and m is the first argument position corresponding to these `COMPONENTS`. Then w' are the sorts of these arguments, so $w' = \langle s_m, \ldots, s_{m+|w'|-1}\rangle$.

$$\frac{s' \in S}{(S, TF, PF, P), f, ws, m \vdash s' \rhd \langle s'\rangle, (\emptyset, \emptyset)}$$

$$\frac{s' \in S \qquad (\langle s_1,\ldots,s_n\rangle, s) = ws \qquad x^i \neq x^j \text{ for all } 1 \le i \neq j \le n}{\begin{array}{l}(S, TF, PF, P), f, ws, m \vdash \texttt{total-select}\ f^1\ \ldots\ f^p\ s' \rhd \\ \quad \langle \underbrace{s', \ldots, s'}_{p\ \text{times}} \rangle, ((\emptyset, \{(\langle s\rangle, s') \mapsto \{f^1,\ldots,f^p\}\}, \emptyset, \emptyset), \\ \qquad\qquad \{\forall\{x^1_{s_1},\ldots,x^n_{s_n}\}. D(f_{ws}\langle x^1_{s_1},\ldots,x^n_{s_n}\rangle) \Rightarrow \\ \qquad\qquad\qquad\qquad f^1_{\langle s\rangle, s'}\langle f_{ws}\langle x^1_{s_1},\ldots,x^n_{s_n}\rangle\rangle \overset{\mathrm{s}}{=} x^m_{s_m}, \\ \qquad\qquad \ldots, \\ \qquad\qquad \forall\{x^1_{s_1},\ldots,x^n_{s_n}\}. D(f_{ws}\langle x^1_{s_1},\ldots,x^n_{s_n}\rangle) \Rightarrow \\ \qquad\qquad\qquad\qquad f^p_{\langle s\rangle, s'}\langle f_{ws}\langle x^1_{s_1},\ldots,x^n_{s_n}\rangle\rangle \overset{\mathrm{s}}{=} x^{m+p-1}_{s_{m+p-1}}\})\end{array}}$$

Note that $f^1,\ldots,f^p$ will be in TF because of non-linear visibility. If the constructor f is declared as total then the definedness conditions in the sentences produced are redundant but harmless.

$$\frac{s' \in S \qquad (\langle s_1,\ldots,s_n\rangle, s) = ws \qquad x^i \neq x^j \text{ for all } 1 \le i \neq j \le n}{\begin{array}{l}(S, TF, PF, P), f, ws, m \vdash \texttt{partial-select}\ f^1\ \ldots\ f^p\ s' \rhd \\ \quad \langle \underbrace{s', \ldots, s'}_{p\ \text{times}} \rangle, ((\emptyset, \emptyset, \{(\langle s\rangle, s') \mapsto \{f^1,\ldots,f^p\}\}, \emptyset), \\ \qquad\qquad \{\forall\{x^1_{s_1},\ldots,x^n_{s_n}\}. D(f_{ws}\langle x^1_{s_1},\ldots,x^n_{s_n}\rangle) \Rightarrow \\ \qquad\qquad\qquad\qquad f^1_{\langle s\rangle, s'}\langle f_{ws}\langle x^1_{s_1},\ldots,x^n_{s_n}\rangle\rangle \overset{\mathrm{s}}{=} x^m_{s_m}, \\ \qquad\qquad \ldots, \\ \qquad\qquad \forall\{x^1_{s_1},\ldots,x^n_{s_n}\}. D(f_{ws}\langle x^1_{s_1},\ldots,x^n_{s_n}\rangle) \Rightarrow \\ \qquad\qquad\qquad\qquad f^p_{\langle s\rangle, s'}\langle f_{ws}\langle x^1_{s_1},\ldots,x^n_{s_n}\rangle\rangle \overset{\mathrm{s}}{=} x^{m+p-1}_{s_{m+p-1}}\})\end{array}}$$

Note that $f^1,\ldots,f^p$ will be in PF because of non-linear visibility. If the constructor f is declared as total then the definedness conditions in the sentences produced are redundant but harmless.

Free Datatype Declarations

A `FREE-DATATYPE` construct is only well-formed when its constructors are total. The same sorts, constructors, and selectors are declared as in ordinary datatype declarations. Apart from the sentences defining the values of selectors, additional sentences require the constructors to be injective, the ranges of constructors of the same sort to be disjoint, the declared sorts to be generated by the constructors, and that applying a selector to a constructor for which it has not been declared is undefined.

```
FREE-DATATYPE ::= free-datatype DATATYPE-ITEMS
```

$$\boxed{\Sigma \vdash \texttt{FREE-DATATYPE} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W \\ (S, TF, PF, P) = \Sigma \qquad (S', TF', \emptyset, P') = \Delta \qquad S'' = S \cup S'\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{free-datatype DATATYPE-ITEMS} \rhd \\ \quad (\Delta \cup \Delta', \Psi \cup \{\mathit{injective}(f_{w,s}) \mid w \in \mathit{FinSeq}(S''), s \in S'', f \in \mathit{TF}'_{w,s}\} \\ \qquad\qquad \cup \{\mathit{disjoint\text{-}ranges}(f_{w,s}, g_{w',s}) \\ \qquad\qquad\qquad \mid w, w' \in \mathit{FinSeq}(S''), s \in S'', f \in \mathit{TF}'_{w,s}, g \in \mathit{TF}'_{w',s} \\ \qquad\qquad\qquad\ \ \text{such that } w \neq w' \text{ or } f \neq g\} \\ \qquad\qquad \cup \{\mathit{undefined\text{-}selection}(f_{w,s}, g_{\langle s \rangle, s'}) \\ \qquad\qquad\qquad \mid f_{w,s}, f'_{w',s} \in \mathit{Dom}(W), g_{\langle s \rangle, s'} \in W(f'_{w',s}) \setminus W(f_{w,s})\} \\ \qquad\qquad \cup \{(S', \mathit{complete}(\mathit{TF}', \mathit{FinSeq}(S'') \times S''))\}\,)\end{array}}$$

where:

- *injective*$(f_{w,s})$ is the following $(\Sigma \cup \Delta \cup \Delta')$-sentence which states that $f_{w,s}$ is injective:

$$\begin{array}{l}\forall \{x^1_{s_1}, \ldots, x^n_{s_n}, y^1_{s_1}, \ldots, y^n_{s_n}\}. \\ \qquad f_{w,s}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle \overset{\mathrm{s}}{=} f_{w,s}\langle y^1_{s_1}, \ldots, y^n_{s_n}\rangle \Rightarrow \\ \qquad\qquad x^1_{s_1} \overset{\mathrm{s}}{=} y^1_{s_1} \wedge \cdots \wedge x^n_{s_n} \overset{\mathrm{s}}{=} y^n_{s_n}\end{array}$$

  where $\langle s_1, \ldots, s_n \rangle = w$ and $x^1, \ldots, x^n, y^1, \ldots, y^n$ are distinct variables.
- *disjoint-ranges*$(f_{w,s}, g_{w',s})$ is the following $(\Sigma \cup \Delta \cup \Delta')$-sentence which states that $f_{w,s}$ and $g_{w',s}$ have disjoint ranges:

$$\forall \{x^1_{s_1}, \ldots, x^m_{s_m}, y^1_{s'_1}, \ldots, y^n_{s'_n}\}. \neg (f_{w,s}\langle x^1_{s_1}, \ldots, x^m_{s_m}\rangle \overset{\mathrm{s}}{=} g_{w',s}\langle y^1_{s'_1}, \ldots, y^n_{s'_n}\rangle)$$

  where $\langle s_1, \ldots, s_m \rangle = w$, $\langle s'_1, \ldots, s'_n \rangle = w'$ and $x^1, \ldots, x^m, y^1, \ldots, y^n$ are distinct variables.
- *undefined-selection*$(f_{w,s}, g_{\langle s \rangle, s'})$ is the following $(\Sigma \cup \Delta \cup \Delta')$-sentence which states that the value of applying the selector $g_{\langle s \rangle, s'}$ to values produced by the constructor $f_{w,s}$ (for which it has not been declared) is undefined:

$$\forall \{x^1_{s_1}, \ldots, x^n_{s_n}\}. \neg D(g_{\langle s \rangle, s'}\langle f_{w,s}\langle x^1_{s_1}, \ldots, x^n_{s_n}\rangle\rangle)$$

  where $\langle s_1, \ldots, s_n \rangle = w$ and $x^1, \ldots, x^n$ are distinct variables.

See the beginning of Sect. 2.3.4 for an explanation of Δ, Δ', Ψ and W as produced by the judgement for `DATATYPE-ITEMS`. The third premise imposes the condition that all declared constructors are total. Note that $(S', \mathit{complete}(\mathit{TF}', \mathit{FinSeq}(S'') \times S''))$ in the last line of the rule is a sort generation constraint, and recall that this abbreviates $(S', \mathit{complete}(\mathit{TF}', \mathit{FinSeq}(S'') \times S''), \mathit{id}_{\Sigma \cup \Delta \cup \Delta'})$. This requires that all values of sorts declared by `DATATYPE-ITEMS` are generated by the declared constructors.

The following proposition states that the resulting model class is the same as for a free extension with the datatype declarations.

Proposition 2.19. *Consider a declaration* `free-datatype DATATYPE-ITEMS`*, a signature Σ and a model class $\mathcal{M}$ over Σ, and suppose*

$$\frac{\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W}{\Sigma \vdash \texttt{free-datatype DATATYPE-ITEMS} \rhd (\Delta \cup \Delta', \Psi')}$$

such that `DATATYPE-ITEMS` *fulfills the following conditions (all referring to fully qualified symbols):*

- *The sorts in Δ (and hence the constructors in Δ and the selectors in Δ') are not in the local environment Σ; and*
- *Any selector in Δ' is total only when the same selector is present in all* `ALTERNATIVE`*s for that sort.*

Let $\mathcal{C}$ be the full subcategory of $\mathbf{Mod}(\Sigma \cup \Delta \cup \Delta')$ containing those $(\Sigma \cup \Delta \cup \Delta')$-models M'' such that $M'' \models \psi$ for all $\psi \in \Psi$, and let $\mathcal{M}'$ and $\mathcal{M}''$ be the $(\Sigma \cup \Delta \cup \Delta')$-model classes

$$\begin{aligned}\mathcal{M}' = \{&(\Sigma \cup \Delta \cup \Delta')\text{-}model\ M' \\ &\mid M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} \in \mathcal{M} \text{ and } M' \in \mathcal{C} \text{ is free over } M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} \\ &\quad w.r.t.\ .|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} : \mathcal{C} \to \mathbf{Mod}(\Sigma)\} \\ \mathcal{M}'' = \{&(\Sigma \cup \Delta \cup \Delta')\text{-}model\ M' \\ &\mid M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} \in \mathcal{M} \text{ and } M' \models \psi' \text{ for all } \psi' \in \Psi'\}\end{aligned}$$

Then $\mathcal{M}' = \mathcal{M}''$.

Proof. See Theorem 3.11 in Sect. 3.2.2 for a more general result. □

A few examples should help to clarify the above definitions. Since there is no overloading in these examples, ordinary function names are used instead of qualified function names to reduce clutter, and the usual syntax for variable typing is used.

Here is an example of a free datatype declaration where all alternatives are constants, which corresponds to an unordered enumeration type:

free type *Color* ::= *red* | *blue*

The result is the following enrichment (relative to the empty signature):

$$\begin{aligned}(\Sigma, \{&red\langle\rangle \overset{s}{=} red\langle\rangle \Rightarrow true, \\ &blue\langle\rangle \overset{s}{=} blue\langle\rangle \Rightarrow true, \\ &\neg(red\langle\rangle \overset{s}{=} blue\langle\rangle), \\ &\neg(blue\langle\rangle \overset{s}{=} red\langle\rangle), \\ &(\{Color\}, TF, id_\Sigma)\}\end{aligned}$$

where $\Sigma = (S, TF, PF, P)$ is the signature containing the sort *Color*, the total function symbols *red* and *blue*, and no partial function symbols or predicate symbols. The first two sentences are from the *injective* condition and are tautologous, as always for nullary constructors. The next two sentences are from the *disjoint-ranges* condition and are equivalent (such duplication will always be present but it does no harm). The final sentence is a sort generation constraint which requires every value of sort *Color* to be produced by either $red\langle\rangle$ or $blue\langle\rangle$. Each model has a carrier of sort *Color* containing exactly two values.

Here is the standard example of lists, with selectors:

$$\textbf{free type } List ::= nil \mid cons(first :?Elem;\ rest :?List)$$

The result is the following enrichment (relative to a signature Σ containing just the sort *Elem*):

$$\begin{aligned}
(\Delta, \{&\forall x{:}Elem, x'{:}List.D(cons\langle x, x'\rangle) \Rightarrow first\langle cons\langle x, x'\rangle\rangle \overset{s}{=} x,\\
&\forall x{:}Elem, x'{:}List.D(cons\langle x, x'\rangle) \Rightarrow rest\langle cons\langle x, x'\rangle\rangle \overset{s}{=} x',\\
&nil\langle\rangle \overset{s}{=} nil\langle\rangle \Rightarrow true,\\
&\forall x{:}Elem, x'{:}List, y{:}Elem, y'{:}List.\\
&\qquad\qquad cons\langle x, x'\rangle \overset{s}{=} cons\langle y, y'\rangle \Rightarrow x \overset{s}{=} y \wedge x' \overset{s}{=} y',\\
&\forall x{:}Elem, x'{:}List.\neg(nil\langle\rangle \overset{s}{=} cons\langle x, x'\rangle),\\
&\forall x{:}Elem, x'{:}List.\neg(cons\langle x, x'\rangle \overset{s}{=} nil\langle\rangle),\\
&\neg D(first\langle nil\langle\rangle\rangle),\\
&\neg D(rest\langle nil\langle\rangle\rangle),\\
&(\{List\}, TF, id_{\Sigma\cup\Delta})\}
\end{aligned}$$

where $\Delta = (S, TF, PF, P)$ is the signature extension (relative to Σ) containing the sort *List*, the total function symbols *nil* and *cons*, the partial function symbols *first* and *rest*, and no predicate symbols. The first two sentences are generated by the rules for `COMPONENTS` and specify the relationship between the constructor *cons* and the selectors *first* and *rest*. The next two sentences are from the *injective* condition. The next two sentences are from the *disjoint-ranges* condition; again, they are equivalent. The next two sentences are from the *undefined-selection* condition. The final sentence is a sort generation constraint which requires each value of sort *List* to be produced by a term of the form

$$cons\langle x_1, \ldots, cons\langle x_n, nil\rangle \ldots\rangle.$$

for some assignment of values of sort *Elem* to the variables $x_1, \ldots, x_n$. Models are as one would expect from this specification, with 'no junk' and 'no confusion', and the selectors defined only for values produced by *cons*.

Here is a type containing two copies of the natural numbers, with the same selector for both:

$$\textbf{free type } Twonats ::= left(get : Nat) \mid right(get : Nat)$$

The result is the following enrichment (relative to a signature Σ containing just the sort *Nat*):

$$\begin{aligned}(\Delta, \{&\forall x{:}Nat.D(\mathit{left}\langle x\rangle) \Rightarrow \mathit{get}\langle \mathit{left}\langle x\rangle\rangle \overset{s}{=} x,\\ &\forall x{:}Nat.D(\mathit{right}\langle x\rangle) \Rightarrow \mathit{get}\langle \mathit{right}\langle x\rangle\rangle \overset{s}{=} x,\\ &\forall x{:}Nat, x'{:}Nat.\mathit{left}\langle x\rangle \overset{s}{=} \mathit{left}\langle x'\rangle \Rightarrow x \overset{s}{=} x',\\ &\forall x{:}Nat, x'{:}Nat.\mathit{right}\langle x\rangle \overset{s}{=} \mathit{right}\langle x'\rangle \Rightarrow x \overset{s}{=} x',\\ &\forall x{:}Nat, x'{:}Nat.\neg(\mathit{left}\langle x\rangle \overset{s}{=} \mathit{right}\langle x'\rangle),\\ &\forall x{:}Nat, x'{:}Nat.\neg(\mathit{right}\langle x\rangle \overset{s}{=} \mathit{left}\langle x'\rangle),\\ &(\mathit{Twonats}, \mathit{TF}', \mathit{id}_{\Sigma\cup\Delta})\}\end{aligned}$$

where Δ is the signature extension (relative to Σ) containing the sort *Twonats*, the total function symbols *left*, *right* and *get*, and no partial function symbols or predicate symbols, and TF' contains the total function symbols *left* and *right*. The first two sentences are generated by the rules for `COMPONENTS` and specify the relationship between the constructors *left* and *right* and the selector *get*. The next two sentences are from the *injective* condition. The next two sentences are from the *disjoint-ranges* condition; once more, they are equivalent. The final sentence is a sort generation constraint which requires each value of sort *Twonats* to be produced by either $\mathit{left}\langle x\rangle$ or $\mathit{right}\langle x\rangle$ for some assignment of a value of sort *Nat* to x. Models are as one would expect, with two copies of *Nat* – one produced using *left*, the other produced using *right*. Note that the total selector *get* : *Twonats* → *Nat* which is present in both `ALTERNATIVE`s becomes a single selector in the rule for `DATATYPE-DECL`: we have

$$\Sigma, \mathit{Twonats} \vdash \texttt{total-construct}\ \mathit{left}\ (\texttt{total-select}\ \mathit{get}\ \mathit{Nat}) \rhd (\Delta_1, \Delta_1', \Psi_1), W_1$$

$$\Sigma, \mathit{Twonats} \vdash \texttt{total-construct}\ \mathit{right}\ (\texttt{total-select}\ \mathit{get}\ \mathit{Nat}) \rhd (\Delta_2, \Delta_2', \Psi_2), W_2$$

where Δ_1 contains *left*, Δ_2 contains *right*, $\Delta_1' = \Delta_2'$ contains *get*, Ψ_1 contains $\forall x{:}Nat.D(\mathit{left}\langle x\rangle) \Rightarrow \mathit{get}\langle \mathit{left}\langle x\rangle\rangle \overset{s}{=} x$, Ψ_2 contains $\forall x{:}Nat.D(\mathit{right}\langle x\rangle) \Rightarrow \mathit{get}\langle \mathit{right}\langle x\rangle\rangle \overset{s}{=} x$, and W_1 and W_2 map *left* and *right* respectively to $\emptyset$.

Changing the declaration to

free type *Twonats* ::= *left*(*get* : *Nat*) | *right*(*Nat*)

would cause the sentence $\forall x{:}Nat.D(\mathit{right}\langle x\rangle) \Rightarrow \mathit{get}\langle \mathit{right}\langle x\rangle\rangle \overset{s}{=} x$ to be omitted from the result, but otherwise there would be no difference. Note that the term $\mathit{get}\langle \mathit{right}\langle x\rangle\rangle$ is still required to have some defined value for every x, since *get* and *right* are total function symbols, but that value is unconstrained. This is an example where the equivalence stated by Prop. 2.19 does not hold, and indeed the total selector *get* violates one of its conditions.

Finally, here is what happens when an attempt is made to define an empty type as a free datatype:

free type *Empty* ::= f(*Empty*)

The result is the following enrichment (relative to the empty signature):

$$(\Sigma, \{\forall x{:}\mathit{Empty}, y{:}\mathit{Empty}. f\langle x\rangle \overset{\mathrm{s}}{=} f\langle y\rangle \Rightarrow x \overset{\mathrm{s}}{=} y,\\ (\mathit{Empty}, \mathit{TF}, id_\Sigma)\})$$

where $\Sigma = (S, \mathit{TF}, \mathit{PF}, P)$ is the signature containing the sort *Empty*, the total function symbol f, and no partial function symbols or predicate symbols. The first sentence is from the *injective* condition. The second sentence is a sort generation constraint which requires every value of sort *Empty* to be produced by a Σ-term containing no variables. There are no such terms since there are no constants of sort *Empty*; hence this requires the carrier of sort *Empty* to be empty. But models are required to have non-empty carriers, and therefore there are no models.

2.3.5 Sort Generation

A sort generation `SORT-GEN` determines the same signature elements and sentences as its list of `SIG-ITEMS`s, together with a sort generation constraint requiring the declared sorts to be generated by the declared operations, but excluding operations declared as selectors.

```
SORT-GEN ::= sort-gen SIG-ITEMS+
```

$$\boxed{\Sigma \vdash \texttt{SORT-GEN} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{SIG-ITEMS}_1 \rhd (\Delta_1, \Delta'_1, \Psi_1) \\ \cdots \\ \Sigma \cup \Delta_1 \cup \Delta'_1 \cup \cdots \cup \Delta_{n-1} \cup \Delta'_{n-1} \vdash \texttt{SIG-ITEMS}_n \rhd (\Delta_n, \Delta'_n, \Psi_n) \\ (S, \mathit{TF}, \mathit{PF}, P) = \Delta = \Delta_1 \cup \cdots \cup \Delta_n \qquad \Delta' = \Delta'_1 \cup \cdots \cup \Delta'_n \\ (S', \mathit{TF}', \mathit{PF}', P') = \Sigma \cup \Delta \cup \Delta' \qquad S \neq \emptyset\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{sort-gen SIG-ITEMS}_1 \ \ldots\ \texttt{SIG-ITEMS}_n \rhd \\ \quad (\Delta \cup \Delta', \Psi_1 \cup \cdots \cup \Psi_n \cup \{(S, \mathit{complete}(\mathit{TF} \cup \mathit{PF}, \mathit{FinSeq}(S') \times S'))\})\end{array}}$$

In this rule, Δ represents the signature extension declared by $\texttt{SIG-ITEMS}_1 \ldots \texttt{SIG-ITEMS}_n$, excluding the operations declared as selectors since these do not contribute to the resulting sort generation constraint. The predicate symbols in Δ also make no contribution.

2.4 Variables

Variables for use in terms may be declared globally, locally, or with explicit quantification. Globally or locally declared variables are implicitly universally quantified in subsequent axioms of the enclosing basic specification.

2.4.1 Global Variable Declarations

```
VAR-ITEMS ::= var-items VAR-DECL+
```

$$\boxed{S \vdash \texttt{VAR-ITEMS} \rhd X}$$

X is a valid set of variables over S.

$$\frac{S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \qquad \cdots \qquad S \vdash \texttt{VAR-DECL}_n \rhd X_n}{S \vdash \texttt{var-items VAR-DECL}_1 \ \ldots \ \texttt{VAR-DECL}_n \rhd X_1 + \cdots + X_n}$$

A variable declaration VAR-DECL declares the given variables to be of the given sort for use in subsequent axioms. This adds a universal quantification on those variables to the subsequent axioms of the enclosing basic specification.

```
VAR-DECL ::= var-decl VAR+ SORT
VAR      ::= SIMPLE-ID
```

$$\boxed{S \vdash \texttt{VAR-DECL} \rhd X}$$

X is a valid set of variables over S.

$$\frac{s \in S}{S \vdash \texttt{var-decl}\ x_1 \ \ldots \ x_n \ \ s \rhd \mathit{complete}(\{s \mapsto \{x_1, \ldots, x_n\}\}, S)}$$

A later declaration for a variable overrides an earlier declaration for the same identifier because of the use of + to combine variable sets in the rules for BASIC-SPEC and VAR-ITEMS. Universal quantification over all declared variables, both global and local, is added in the rule for AXIOM, see Sect. 2.5 below.

$$\mathit{Var} = \texttt{SIMPLE-ID}$$

2.4.2 Local Variable Declarations

A LOCAL-VAR-AXIOMS construct declares variables for local use in the given axioms, and adds a universal quantification on those variables to all those axioms.

```
LOCAL-VAR-AXIOMS ::= local-var-axioms VAR-DECL+ AXIOM+
```

$$\boxed{\Sigma, X \vdash \texttt{LOCAL-VAR-AXIOMS} \rhd \Psi}$$

X is required to be a valid set of variables over the sorts of Σ. Ψ is a set of Σ-sentences.

$$\frac{\begin{array}{c}(S, \mathit{TF}, \mathit{PF}, P) = \Sigma \\ S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \qquad \cdots \qquad S \vdash \texttt{VAR-DECL}_m \rhd X_m \\ \Sigma, X + X_1 + \cdots + X_m \vdash \texttt{AXIOM}_1 \rhd \psi_1 \\ \cdots \\ \Sigma, X + X_1 + \cdots + X_m \vdash \texttt{AXIOM}_n \rhd \psi_n\end{array}}{\begin{array}{l}\Sigma, X \vdash \texttt{local-var-axioms VAR-DECL}_1 \ldots \texttt{VAR-DECL}_m \\ \qquad\qquad\qquad\qquad \texttt{AXIOM}_1 \ldots \texttt{AXIOM}_n \rhd \{\psi_1, \ldots, \psi_n\}\end{array}}$$

2.5 Axioms

Each well-formed axiom determines a sentence of the underlying basic specification (closed by universal quantification over all declared variables).

```
AXIOM-ITEMS ::= axiom-items AXIOM+
AXIOM       ::= FORMULA
```

$$\boxed{\Sigma, X \vdash \texttt{AXIOM-ITEMS} \rhd \Psi}$$

X is required to be a valid set of variables over the sorts of Σ. Ψ is a set of Σ-sentences.

$$\frac{\Sigma, X \vdash \texttt{AXIOM}_1 \rhd \psi_1 \qquad \cdots \qquad \Sigma, X \vdash \texttt{AXIOM}_n \rhd \psi_n}{\Sigma, X \vdash \texttt{axiom-items AXIOM}_1 \ldots \texttt{AXIOM}_n \rhd \{\psi_1, \ldots, \psi_n\}}$$

$$\boxed{\Sigma, X \vdash \texttt{AXIOM} \rhd \psi}$$

X is required to be a valid set of variables over the sorts of Σ. ψ is a Σ-sentence.

$$\frac{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi}{\Sigma, X \vdash \texttt{FORMULA}\ \text{qua}\ \texttt{AXIOM} \rhd \forall X . \varphi}$$

All declared variables are universally quantified. Quantification over variables that do not occur free in the axiom has no effect since carriers are assumed to be non-empty.

A formula is constructed from atomic formulas using quantification and the usual logical connectives.

```
FORMULA ::= QUANTIFICATION | CONJUNCTION | DISJUNCTION
          | IMPLICATION | EQUIVALENCE | NEGATION | ATOM
```

$$\boxed{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

Rules elided, except the one for `ATOM` qua `FORMULA` which is near the beginning of Sect. 2.5.3 below to keep it together with subordinate rules for atomic formulas.

2.5.1 Quantifications

Universal, existential and unique-existential quantification are as usual. An inner declaration for a variable with the same identifier as in an outer declaration overrides the outer declaration, regardless of whether the sorts of the variables are the same.

```
QUANTIFICATION ::= quantification QUANTIFIER VAR-DECL+ FORMULA
QUANTIFIER     ::= universal | existential | unique-existential
```

$$\boxed{\Sigma, X \vdash \texttt{QUANTIFICATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \\ S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \qquad \cdots \qquad S \vdash \texttt{VAR-DECL}_n \rhd X_n \\ \Sigma, X + X_1 + \cdots + X_n \vdash \texttt{FORMULA} \rhd \varphi\end{array}}{\begin{array}{l}\Sigma, X \vdash \texttt{quantification universal} \\ \qquad\qquad \texttt{VAR-DECL}_1 \ \ldots \ \texttt{VAR-DECL}_n \ \texttt{FORMULA} \rhd \forall X_1 + \cdots + X_n . \varphi\end{array}}$$

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \\ S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \qquad \cdots \qquad S \vdash \texttt{VAR-DECL}_n \rhd X_n \\ \Sigma, X + X_1 + \cdots + X_n \vdash \texttt{FORMULA} \rhd \varphi\end{array}}{\begin{array}{l}\Sigma, X \vdash \texttt{quantification existential} \\ \qquad\qquad \texttt{VAR-DECL}_1 \ \ldots \ \texttt{VAR-DECL}_n \ \texttt{FORMULA} \rhd \exists X_1 + \cdots + X_n . \varphi\end{array}}$$

$$\frac{\begin{array}{c}(S, TF, PF, P) = \Sigma \\ S \vdash \texttt{VAR-DECL}_1 \rhd X_1 \qquad \cdots \qquad S \vdash \texttt{VAR-DECL}_n \rhd X_n \\ \Sigma, X + X_1 + \cdots + X_n \vdash \texttt{FORMULA} \rhd \varphi\end{array}}{\begin{array}{l}\Sigma, X \vdash \texttt{quantification unique-existential} \\ \qquad\qquad \texttt{VAR-DECL}_1 \ \ldots \ \texttt{VAR-DECL}_n \ \texttt{FORMULA} \rhd \exists! X_1 + \cdots + X_n . \varphi\end{array}}$$

2.5.2 Logical Connectives

The logical connectives are as usual, except that conjunction and disjunction apply to lists of two or more formulas.

Conjunction

```
CONJUNCTION ::= conjunction FORMULA+
```

$$\boxed{\Sigma, X \vdash \texttt{CONJUNCTION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\begin{array}{c}\Sigma, X \vdash \texttt{FORMULA}_1 \rhd \varphi_1 \\ \Sigma, X \vdash \texttt{FORMULA}_2 \rhd \varphi_2 \\ \ldots \\ \Sigma, X \vdash \texttt{FORMULA}_n \rhd \varphi_n\end{array}}{\begin{array}{r}\Sigma, X \vdash \texttt{conjunction FORMULA}_1 \texttt{ FORMULA}_2 \texttt{ ... FORMULA}_n \rhd \\ (\cdots(\varphi_1 \wedge \varphi_2) \wedge \cdots) \wedge \varphi_n\end{array}}$$

Disjunction

```
DISJUNCTION ::= disjunction FORMULA+
```

$$\boxed{\Sigma, X \vdash \texttt{DISJUNCTION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\begin{array}{c}\Sigma, X \vdash \texttt{FORMULA}_1 \rhd \varphi_1 \\ \Sigma, X \vdash \texttt{FORMULA}_2 \rhd \varphi_2 \\ \ldots \\ \Sigma, X \vdash \texttt{FORMULA}_n \rhd \varphi_n\end{array}}{\begin{array}{r}\Sigma, X \vdash \texttt{disjunction FORMULA}_1 \texttt{ FORMULA}_2 \texttt{ ... FORMULA}_n \rhd \\ (\cdots(\varphi_1 \vee \varphi_2) \vee \cdots) \vee \varphi_n\end{array}}$$

Implication

```
IMPLICATION ::= implication FORMULA FORMULA
```

$$\boxed{\Sigma, X \vdash \texttt{IMPLICATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi \qquad \Sigma, X \vdash \texttt{FORMULA}' \rhd \varphi'}{\Sigma, X \vdash \texttt{implication FORMULA FORMULA}' \rhd \varphi \Rightarrow \varphi'}$$

Equivalence

```
EQUIVALENCE ::= equivalence FORMULA FORMULA
```

$$\boxed{\Sigma, X \vdash \texttt{EQUIVALENCE} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi \qquad \Sigma, X \vdash \texttt{FORMULA}' \rhd \varphi'}{\Sigma, X \vdash \texttt{equivalence FORMULA FORMULA}' \rhd \varphi \Leftrightarrow \varphi'}$$

Negation

```
NEGATION ::= negation FORMULA
```

$$\boxed{\Sigma, X \vdash \texttt{NEGATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi}{\Sigma, X \vdash \texttt{negation FORMULA} \rhd \neg\varphi}$$

2.5.3 Atomic Formulas

An atomic formula is well-formed if it is *well-sorted* and expands to a unique atomic formula for constructing sentences. The notions of when an atomic formula is well-sorted, of when a term is *well-sorted for a particular sort*, and of the *expansions* of atomic formulas and terms, are captured by the rules below.

```
ATOM ::= TRUTH | PREDICATION | DEFINEDNESS
       | EXISTL-EQUATION | STRONG-EQUATION
```

(The following rule really belongs just before Sect. 2.5.1 above. It is here in order to keep it together with the subordinate rules for atomic formulas, because of the complications introduced by the 'unique expansion' requirement.)

$$\frac{\begin{array}{c}\text{there is a unique } \varphi \text{ such that } \Sigma, X \vdash \texttt{ATOM} \rhd \varphi \\ \Sigma, X \vdash \texttt{ATOM} \rhd \varphi\end{array}}{\Sigma, X \vdash \texttt{ATOM}\,\text{qua}\,\texttt{FORMULA} \rhd \varphi}$$

The first premise of this rule imposes the requirement that ATOM expands to a unique (fully-qualified) atomic formula. In this premise, the static semantics of ATOM occurs in a negative position (introduced by "there is a unique φ"). This is potentially problematic, especially since there is a circularity: the judgement $\Sigma, X \vdash \texttt{ATOM} \rhd \varphi$ depends on the judgement $\Sigma, X \vdash \texttt{FORMULA} \rhd \varphi'$ if ATOM contains a conditional term. But since FORMULA will then be strictly contained within ATOM, there is no problem: we can (implicitly) impose a stratification on the judgements for FORMULA and ATOM where the semantics of larger formulas/atoms is based on the (fixed) semantics of strictly smaller formulas/atoms.

$$\boxed{\Sigma, X \vdash \texttt{ATOM} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

Rules elided, except for the following one:

$$\frac{\vdash \texttt{TRUTH} \rhd \varphi}{\Sigma, X \vdash \texttt{TRUTH}\ \text{qua}\ \texttt{ATOM} \rhd \varphi}$$

Truth

The atomic formulas for truth and falsity are always well-sorted, and expand to primitive sentences.

```
TRUTH ::= true-atom | false-atom
```

$$\boxed{\vdash \texttt{TRUTH} \rhd \varphi}$$

φ is a Σ-formula over X for any Σ and X.

$$\frac{}{\vdash \texttt{true-atom} \rhd \mathit{true}}$$

$$\frac{}{\vdash \texttt{false-atom} \rhd \mathit{false}}$$

Predicate Application

The application of a predicate symbol is well-sorted when there is a declaration of the predicate name such that all the argument terms are well-sorted for the respective argument sorts. It then expands to an application of the qualified predicate name to the fully-qualified expansions of the argument terms for those sorts.

```
PREDICATION    ::= predication PRED-SYMB TERMS
PRED-SYMB      ::= PRED-NAME | QUAL-PRED-NAME
QUAL-PRED-NAME ::= qual-pred-name PRED-NAME PRED-TYPE
```

$$\boxed{\Sigma, X \vdash \texttt{PREDICATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{PRED-SYMB} \rhd p, \langle s_1, \ldots, s_n \rangle \qquad \Sigma, X \vdash \texttt{TERMS} \rhd \langle t_1, \ldots, t_n \rangle \\ sort(t_1) = s_1 \quad \cdots \quad sort(t_n) = s_n\end{array}}{\Sigma, X \vdash \texttt{predication PRED-SYMB TERMS} \rhd p_{\langle s_1, \ldots, s_n \rangle} \langle t_1, \ldots, t_n \rangle}$$

$$\boxed{\Sigma \vdash \texttt{PRED-SYMB} \rhd p, w}$$

p is a predicate symbol in Σ with profile w.

$$\frac{\{s_1, \ldots, s_n\} \subseteq S \qquad p \in P_{\langle s_1, \ldots, s_n \rangle}}{(S, \mathit{TF}, \mathit{PF}, P) \vdash p \rhd p, \langle s_1, \ldots, s_n \rangle}$$

$$\frac{S \vdash \texttt{PRED-TYPE} \rhd w \qquad p \in P_w}{(S, \mathit{TF}, \mathit{PF}, P) \vdash \texttt{qual-pred-name}\ p\ \texttt{PRED-TYPE} \rhd p, w}$$

Definedness

A definedness formula is well-sorted when the term is well-sorted for some sort. It then expands to a definedness assertion on the fully-qualified expansion of the term.

```
DEFINEDNESS ::= definedness TERM
```

$$\boxed{\Sigma, X \vdash \texttt{DEFINEDNESS} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t}{\Sigma, X \vdash \texttt{definedness TERM} \rhd D(t)}$$

Equations

An equation is well-sorted if both terms are well-sorted for some sort. It then expands to the corresponding equation on the fully-qualified expansions of the terms for that sort.

```
EXISTL-EQUATION ::= existl-equation TERM TERM
STRONG-EQUATION ::= strong-equation TERM TERM
```

$$\boxed{\Sigma, X \vdash \texttt{EXISTL-EQUATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t \qquad \Sigma, X \vdash \texttt{TERM}' \rhd t' \qquad \mathit{sort}(t) = \mathit{sort}(t')}{\Sigma, X \vdash \texttt{existl-equation TERM TERM}' \rhd t \overset{\mathrm{e}}{=} t'}$$

$$\boxed{\Sigma, X \vdash \texttt{STRONG-EQUATION} \rhd \varphi}$$

X is required to be a valid set of variables over the sorts of Σ. φ is a Σ-formula over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t \qquad \Sigma, X \vdash \texttt{TERM}' \rhd t' \qquad \mathit{sort}(t) = \mathit{sort}(t')}{\Sigma, X \vdash \texttt{strong-equation TERM TERM}' \rhd t \overset{\mathrm{s}}{=} t'}$$

2.5.4 Terms

A term is constructed from variables by applications of operations. All names used in terms may be qualified by the intended types, and the intended sort of the term may be specified.

```
TERM ::= SIMPLE-ID | QUAL-VAR | APPLICATION
       | SORTED-TERM | CONDITIONAL
```

$$\boxed{\Sigma, X \vdash \texttt{TERM} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

Rules elided, except for the two rules in the next subsection which are for the case `SIMPLE-ID`.

Identifiers

An unqualified simple identifier in a term may be a variable or a constant, depending on the local environment and the variable declarations. Either is well-sorted for the sort specified in its declaration; a variable expands to the (sorted) variable itself, whereas a constant expands to an application of the qualified symbol to the empty list of arguments.

$$\frac{s \in S \qquad x \in X_s}{(S, \mathit{TF}, \mathit{PF}, P), X \vdash x \rhd x_s}$$

$$\frac{s \in S \qquad f \in \mathit{TF}_{\langle\rangle,s} \cup \mathit{PF}_{\langle\rangle,s}}{(S, \mathit{TF}, \mathit{PF}, P), X \vdash f \rhd f_{\langle\rangle,s}\langle\rangle}$$

Qualified Variables

A qualified variable is well-sorted for the given sort.

```
QUAL-VAR ::= qual-var VAR SORT
```

$$\boxed{\Sigma, X \vdash \texttt{QUAL-VAR} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

$$\frac{s \in S \qquad x \in X_s}{(S, \mathit{TF}, \mathit{PF}, P), X \vdash \texttt{qual-var}\ x\ \ s \rhd x_s}$$

Operation Application

An application is well-sorted for some sort s when there is a declaration of the operation name such that all the argument terms are well-sorted for the respective argument sorts, and the result sort is s. It then expands to an application of the qualified operation name to the fully-qualified expansions of the argument terms for those sorts.

```
APPLICATION  ::= application OP-SYMB TERMS
OP-SYMB      ::= OP-NAME | QUAL-OP-NAME
QUAL-OP-NAME ::= qual-op-name OP-NAME OP-TYPE
TERMS        ::= terms TERM*
```

$$\boxed{\Sigma, X \vdash \texttt{APPLICATION} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{OP-SYMB} \rhd f, (\langle s_1, \ldots, s_n\rangle, s) \qquad \Sigma, X \vdash \texttt{TERMS} \rhd \langle t_1, \ldots, t_n\rangle \\ \mathit{sort}(t_1) = s_1 \quad \cdots \quad \mathit{sort}(t_n) = s_n\end{array}}{\Sigma, X \vdash \texttt{application OP-SYMB TERMS} \rhd f_{\langle s_1,\ldots,s_n\rangle,s}\langle t_1, \ldots, t_n\rangle}$$

$$\boxed{\Sigma \vdash \texttt{OP-SYMB} \rhd f, ws}$$

f is a function symbol in Σ with profile ws.

$$\frac{\{s_1, \ldots, s_n, s\} \subseteq S \qquad f \in TF_{\langle s_1,\ldots,s_n\rangle,s} \cup PF_{\langle s_1,\ldots,s_n\rangle,s}}{(S, TF, PF, P) \vdash f \rhd f, (\langle s_1, \ldots, s_n\rangle, s)}$$

$$\frac{\{s_1, \ldots, s_n, s\} \subseteq S \qquad f \in TF_{\langle s_1,\ldots,s_n\rangle,s}}{\begin{array}{l}(S, TF, PF, P) \vdash \\ \qquad \texttt{qual-op-name}\, f\ (\texttt{total-op-type}\,(\texttt{sort-list}\ s_1\ \ldots\ s_n)\ s) \rhd \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad f, (\langle s_1, \ldots, s_n\rangle, s)\end{array}}$$

$$\frac{\{s_1, \ldots, s_n, s\} \subseteq S \qquad f \in PF_{\langle s_1,\ldots,s_n\rangle,s}}{\begin{array}{l}(S, TF, PF, P) \vdash \\ \qquad \texttt{qual-op-name}\, f\ (\texttt{partial-op-type}\,(\texttt{sort-list}\ s_1\ \ldots\ s_n)\ s) \rhd \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad f, (\langle s_1, \ldots, s_n\rangle, s)\end{array}}$$

$$\boxed{\Sigma, X \vdash \texttt{TERMS} \rhd \langle t_1, \ldots, t_n\rangle}$$

X is required to be a valid set of variables over the sorts of Σ. $t_1, \ldots, t_n$ are fully-qualified Σ-terms over X.

$$\frac{\Sigma, X \vdash \texttt{TERM}_1 \rhd t_1 \quad \cdots \quad \Sigma, X \vdash \texttt{TERM}_n \rhd t_n}{\Sigma, X \vdash \texttt{terms}\ \texttt{TERM}_1\ \ldots\ \texttt{TERM}_n \rhd \langle t_1, \ldots, t_n\rangle}$$

Sorted Terms

A sorted term is well-sorted if the given term is well-sorted for the given sort. It then expands to those fully-qualified expansions of the component term that have the specified sort.

```
SORTED-TERM ::= sorted-term TERM SORT
```

$$\boxed{\Sigma, X \vdash \texttt{SORTED-TERM} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t \qquad sort(t) = s}{\Sigma, X \vdash \texttt{sorted-term}\ \texttt{TERM}\ s \rhd t}$$

Conditional Terms

A conditional term is well-sorted for some sort when both given terms are well-sorted for that sort and the given formula is well-formed. It then expands to a fully-qualified term built from that formula and the fully-qualified expansions of the given terms for that sort.

```
CONDITIONAL ::= conditional TERM FORMULA TERM
```

$$\boxed{\Sigma, X \vdash \texttt{CONDITIONAL} \rhd t}$$

X is required to be a valid set of variables over the sorts of Σ. t is a fully-qualified Σ-term over X.

$$\frac{\Sigma, X \vdash \texttt{TERM} \rhd t \qquad \Sigma, X \vdash \texttt{FORMULA} \rhd \varphi \qquad \Sigma, X \vdash \texttt{TERM}' \rhd t' \qquad \mathit{sort}(t) = \mathit{sort}(t')}{\Sigma, X \vdash \texttt{conditional TERM FORMULA TERM}' \rhd \varphi \to t \mid t'}$$

Conditional terms are interpreted as fully-qualified terms, as explained in Sect. 2.1.3, rather than being handled by transformation of the enclosing atomic formula as is suggested in the CASL Summary; such a transformation would be difficult to define using this style of semantics.

2.6 Identifiers

The internal structure of identifiers ID is insignificant in the context of basic specifications. (ID is extended with compound identifiers, whose structure is significant, in connection with generic specifications in Sect. 4.6.)

```
SIMPLE-ID   ::= WORDS
SORT-ID     ::= WORDS
TOKEN       ::= WORDS | DOT-WORDS | SIGNS | DIGIT | QUOTED-CHAR
ID          ::= id MIX-TOKEN+
MIX-TOKEN   ::= TOKEN | PLACE | BRACED-ID | BRACKET-ID | EMPTY-BRS
BRACED-ID   ::= braced-id ID
BRACKET-ID  ::= bracket-id ID
EMPTY-BRS   ::= empty-braces | empty-brackets
```

3

Subsorting Specification Semantics

The semantics of subsorted specifications is explained largely via reduction to the many-sorted case, already covered in Chap. 2. Section 3.1 defines the underlying concepts for subsorted specifications and explains their relationship to the corresponding concepts of many-sorted specifications. The remaining sections cover the language constructs for subsorted basic specifications, presented as extensions to the constructs for many-sorted specifications.

3.1 Subsorting Concepts

This section extends the institution presented in Sect. 2.1, in order to cope with subsorting.

We represent subsort inclusion by embedding (which is not required to be identity), commuting, as usual in order-sorted approaches, with overloaded operation symbols.

3.1.1 Signatures

A *subsorted signature* Σ consists of a many-sorted signature together with a pre-order of *subsort embedding* on its set of sorts.

$$\leq \; \in \mathit{SortRelation} = \mathit{FinSet}(\mathit{Sort} \times \mathit{Sort})$$

$$(S, \mathit{TF}, \mathit{PF}, P, \leq)$$

$$\text{or } \Sigma \in \mathit{SubSig} =$$

$$\mathit{SortSet} \times \mathit{FunSet} \times \mathit{FunSet} \times \mathit{PredSet} \times \mathit{SortRelation}$$

Requirements on a subsorted signature $(S, \mathit{TF}, \mathit{PF}, P, \leq)$:

- $(S, \mathit{TF}, \mathit{PF}, P)$ is a valid many-sorted signature
- $\leq \; \in \mathit{FinSet}(S \times S)$
- $s \leq s$ for all $s \in S$
- $s \leq s'$ and $s' \leq s''$ implies $s \leq s''$ for all $s, s', s'' \in S$

$\leq$ is extended pointwise to sequences of sorts, that is, $w \leq w'$ iff $w = \langle s_1, \ldots, s_n \rangle$, $w' = \langle s'_1, \ldots, s'_n \rangle$ and $s_j \leq s'_j$ for all $1 \leq j \leq n$.

A *subsorted signature fragment* $(S, \mathit{TF}, \mathit{PF}, P, \leq)$ consists of a (many-sorted) signature fragment $(S, \mathit{TF}, \mathit{PF}, P)$ and a relation $\leq\; \in \mathit{SortRelation}$.

$$(S, \mathit{TF}, \mathit{PF}, P, \leq) \in \mathit{SubsortedSigFragment} = \\ \mathit{SortSet} \times \mathit{FunSet} \times \mathit{FunSet} \times \mathit{PredSet} \times \mathit{SortRelation}$$

The union of subsorted signature fragments requires the resulting subsorting relation to be computed as reflexive and transitive closure, and is defined as follows:

$$(S, \mathit{TF}, \mathit{PF}, P, \leq) \cup (S', \mathit{TF}', \mathit{PF}', P', \leq') = (\overline{S}, \overline{\mathit{TF}}, \overline{\mathit{PF}}, \overline{P}, \mathit{RTClos}(\leq \cup \leq'))$$

where $(\overline{S}, \overline{\mathit{TF}}, \overline{\mathit{PF}}, \overline{P}) = (S, \mathit{TF}, \mathit{PF}, P) \cup (S', \mathit{TF}', \mathit{PF}', P')$ and $\mathit{RTClos} : \mathit{SortRelation} \to \mathit{SortRelation}$ associates each relation with its reflexive and transitive closure, that is, for each $\sim\; \in \mathit{SortRelation}$ such that $\widehat{S} = \{s \mid \exists s'.s \sim s' \vee s' \sim s\}$, the result $\mathit{RTClos}(\sim)$ is the relation $\overset{*}{\sim}$, inductively defined by the following rules:

$$\frac{s \sim s'}{s \overset{*}{\sim} s'}$$

$$\frac{s \in \widehat{S}}{s \overset{*}{\sim} s}$$

$$\frac{s \overset{*}{\sim} s' \qquad s' \overset{*}{\sim} s''}{s \overset{*}{\sim} s''}$$

For each subsorted signature $\Sigma = (S, \mathit{TF}, \mathit{PF}, P, \leq)$ the relation $\leq$ is reflexive and transitive, so that $\leq$ and $\mathit{RTClos}(\leq)$ coincide.

Analogously to the many-sorted case, a *subsorted signature extension* Δ relative to a subsorted signature Σ is a subsorted signature fragment such that $\Sigma \cup \Delta$ is a subsorted signature.

$$(S, \mathit{TF}, \mathit{PF}, P, \leq) \\ \text{or } \Delta \in \mathit{SubsortedExtension} = \mathit{SubsortedSigFragment}$$

It is trivial to extend Prop. 2.1 to the subsorted case.

Proposition 3.1. *If Δ and Δ' are subsorted signature extensions relative to Σ then $\Delta \cup \Delta'$ is a subsorted signature extension relative to Σ.*

Proof. Straightforward. □

As in the many-sorted case, a subsorted signature Σ is a *subsignature* of a subsorted signature Σ' if there is some subsorted extension Δ relative to Σ such that $\Sigma' = \Sigma \cup \Delta$.

For a subsorted signature $(S, \mathit{TF}, \mathit{PF}, P, \leq)$, we define *overloading relations* for operation and predicate symbols. Let $f \in (\mathit{TF}_{w_1,s_1} \cup \mathit{PF}_{w_1,s_1}) \cap$

$(TF_{w_2,s_2} \cup PF_{w_2,s_2})$. Two qualified operation symbols f_{w_1,s_1} and f_{w_2,s_2} are in the overloading relation (written $f_{w_1,s_1} \sim_F f_{w_2,s_2}$) iff there exists a $w \in S^*$ and $s \in S$ such that $w \leq w_1, w_2$ and $s_1, s_2 \leq s$. Similarly, two qualified predicate symbols p_{w_1} and p_{w_2} are in the overloading relation (written $p_{w_1} \sim_P p_{w_2}$) iff there exists a $w \in S^*$ such that $w \leq w_1, w_2$. We say that two profiles of a symbol are in the overloading relation if the corresponding qualified symbols are in the overloading relation.

Let $\Sigma = (S, TF, PF, P, \leq)$ and $\Sigma' = (S', TF', PF', P', \leq')$ be subsorted signatures. Then a *subsorted signature morphism* $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$ from Σ to Σ' is a many sorted signature morphism from (S, TF, PF, P) to (S', TF', PF', P') preserving the subsort relation and the overloading relations, that is:

- $s \leq s'$ implies $\sigma^{\mathsf{S}}(s) \leq' \sigma^{\mathsf{S}}(s')$;
- $f_{ws} \sim_F f'_{ws'}$ implies that one of the following holds:
  - $\sigma^{TF}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)} \sim'_F \sigma^{TF}_{ws'}(f')_{\sigma^{\mathsf{S}}(ws')}$, where $f_{ws} \in TF$ and $f'_{ws'} \in TF$,
  - $\sigma^{TF}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)} \sim'_F \sigma^{PF}_{ws'}(f')_{\sigma^{\mathsf{S}}(ws')}$, where $f_{ws} \in TF$ and $f'_{ws'} \in PF$,
  - $\sigma^{PF}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)} \sim'_F \sigma^{TF}_{ws'}(f')_{\sigma^{\mathsf{S}}(ws')}$, where $f_{ws} \in PF$ and $f'_{ws'} \in TF$, or
  - $\sigma^{PF}_{ws}(f)_{\sigma^{\mathsf{S}}(ws)} \sim'_F \sigma^{PF}_{ws'}(f')_{\sigma^{\mathsf{S}}(ws')}$, where $f_{ws} \in PF$ and $f'_{ws'} \in PF$; and
- $p_w \sim_P p'_{w'}$ implies $\sigma^P_w(p)_{\sigma^{\mathsf{S}}(w)} \sim'_P \sigma^P_{w'}(p')_{\sigma^{\mathsf{S}}(w')}$.

Notice that the preservation of overloading relations is equivalent to the requirement that any two qualified function (predicate) symbols that are in the overloading relation are translated into the same (unqualified) symbol.

If Σ is a subsignature of Σ', we write $\Sigma \hookrightarrow \Sigma'$ for the evident subsorted signature morphism, called a *subsorted signature inclusion.*

Proposition 3.2. *The composition of subsorted signature morphisms does indeed yield a subsorted signature morphism.*

Proof. Straightforward. □

Proposition 3.3. *Subsorted signatures and subsorted signature morphisms form a finitely cocomplete category,* **SubSig**.

Proof. It is easy to see that **SubSig** is a category. Regarding finite cocompleteness, see [38]. □

For a subsorted signature $\Sigma = (S, TF, PF, P, \leq)$, we define $|\Sigma| \subseteq \mathit{SigSym}$ to be $|(S, TF, PF, P)|$, and for a subsorted signature morphism $\sigma : \Sigma \to \Sigma'$, we define $|\sigma| : |\Sigma| \to |\Sigma'|$ as in the many-sorted case.

Proposition 3.4. $|.| :$ **SubSig** $\to$ **Set** *is a faithful functor.*

Proof. Straightforward. □

Proposition 3.5. *A subsorted signature morphism $\sigma : \Sigma \to \Sigma'$ is a signature inclusion iff $|\sigma|$ is an inclusion of $|\Sigma|$ into $|\Sigma'|$.*

Proof. Straightforward. □

Any subsorted signature Σ is associated with a many-sorted signature $\Sigma^{\#}$, where the embeddings of subsorts into their supersorts are explicitly added as operations.

In order to define a reduction of subsorted signatures to many-sorted signatures, we first redefine the universes *FunName* of operation names and *PredName* of predicate names from Sect. 2.3, adding embedding, projection and membership symbols that differ from all those already present:

$$\begin{aligned} \mathit{FunName} &= \mathtt{ID} \uplus \{\mathit{em}\} \uplus \{\mathit{pr}\} \\ \mathit{PredName} &= \mathtt{ID} \uplus \{\mathit{in}(s) \mid s \in \mathit{Sort}\} \end{aligned}$$

Here, '*em*', '*pr*' and '$\mathit{in}(s)$' refer to arbitrary objects, the only requirement being that $\mathit{in}(s) \neq \mathit{in}(s')$ whenever $s \neq s'$.

Then the many-sorted signature $\Sigma^{\#}$ consists of $(S^{\#}, \mathit{TF}^{\#}, \mathit{PF}^{\#}, P^{\#})$, where

$$\begin{aligned} S^{\#} &= S \\ \mathit{TF}^{\#}_{w,s'} &= \begin{cases} \mathit{TF}_{w,s'} \cup \{\mathit{em}\} \text{ if } w = \langle s \rangle \text{ and } s \leq s' \\ \mathit{TF}_{w,s'} \qquad \text{otherwise} \end{cases} \\ \mathit{PF}^{\#}_{w,s} &= \begin{cases} \mathit{PF}_{w,s} \cup \{\mathit{pr}\} \text{ if } w = \langle s' \rangle \text{ and } s \leq s' \\ \mathit{PF}_{w,s} \qquad \text{otherwise} \end{cases} \\ P^{\#}_{w} &= \begin{cases} P_w \cup \{\mathit{in}(s) \mid s \leq s'\} \text{ if } w = \langle s' \rangle \\ P_w \qquad \text{otherwise} \end{cases} \end{aligned}$$

Any subsorted signature morphism $\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$ from Σ to Σ' extends to a many-sorted signature morphism $\sigma^{\#} = ({\sigma^{\#}}^{S}, {\sigma^{\#}}^{TF}, {\sigma^{\#}}^{PF}, {\sigma^{\#}}^{P})$ from $\Sigma^{\#}$ to $\Sigma'^{\#}$ as follows:

$$\begin{aligned} {\sigma^{\#}}^{S} &= \sigma^{\mathsf{S}} \\ {\sigma^{\#}}^{TF}_{w,s'}(f) &= \begin{cases} \mathit{em} \qquad \text{if } w = \langle s \rangle, s \leq s' \text{ and } f = \mathit{em} \\ \sigma^{\mathsf{TF}}_{w,s'}(f) \text{ otherwise} \end{cases} \\ {\sigma^{\#}}^{PF}_{w,s}(f) &= \begin{cases} \mathit{pr} \qquad \text{if } w = \langle s' \rangle, s \leq s' \text{ and } f = \mathit{pr} \\ \sigma^{\mathsf{PF}}_{w,s}(f) \text{ otherwise} \end{cases} \\ {\sigma^{\#}}^{P}_{w}(p) &= \begin{cases} \mathit{in}(\sigma^{\mathsf{S}}(s)) \text{ if } w = \langle s' \rangle, s \leq s' \text{ and } p = \mathit{in}(s) \\ \sigma^{\mathsf{P}}_{w}(p) \qquad \text{otherwise} \end{cases} \end{aligned}$$

Proposition 3.6. *The construction* $(.)^{\#}$ *is a functor from* **SubSig** *to* **Sig**.

Proof. It is easy to see that if σ and σ' are composable subsorted signature morphisms, then $(\sigma \circ \sigma')^{\#} = \sigma^{\#} \circ \sigma'^{\#}$, and that identity is preserved. □

3.1.2 Models

Subsorted models over Σ are the many-sorted $\Sigma^{\#}$-models in which the embedding and projection functions and the membership predicates are well-behaved.

For a subsorted signature $\Sigma = (S, \mathit{TF}, \mathit{PF}, P, \leq)$, the *subsorted models* are ordinary many-sorted models for $\Sigma^{\#}$ satisfying the following axioms:

Identity: $\forall x_s . \mathit{em}_{\langle s \rangle, s}\langle x_s \rangle \stackrel{e}{=} x_s$

Transitivity: $\forall x_s . \mathit{em}_{\langle s' \rangle, s''}\langle \mathit{em}_{\langle s \rangle, s'}\langle x_s \rangle \rangle \stackrel{e}{=} \mathit{em}_{\langle s \rangle, s''}\langle x_s \rangle$ for $s \leq s' \leq s''$

Projection: $\forall x_s . \mathit{pr}_{\langle s' \rangle, s}\langle \mathit{em}_{\langle s \rangle, s'}\langle x_s \rangle \rangle \stackrel{e}{=} x_s$ for $s \leq s'$

Projection-injectivity: $\forall \{x_{s'}, y_{s'}\} . \mathit{pr}_{\langle s' \rangle, s}\langle x_{s'} \rangle \stackrel{e}{=} \mathit{pr}_{\langle s' \rangle, s}\langle y_{s'} \rangle \Rightarrow x_{s'} \stackrel{e}{=} y_{s'}$ for $s \leq s'$

Membership: $\forall x_{s'} . \mathit{in}(s)_{\langle s' \rangle}\langle x_{s'} \rangle \Leftrightarrow D(\mathit{pr}_{\langle s' \rangle, s}\langle x_{s'} \rangle)$ for $s \leq s'$

Function-monotonicity:

$$\forall \{x^1_{\overline{s}_1}, \ldots, x^n_{\overline{s}_n}\} . \mathit{em}_{\langle s \rangle, s''}\langle f_{w,s}\langle \mathit{em}_{\langle \overline{s}_1 \rangle, s_1}\langle x^1_{\overline{s}_1} \rangle, \ldots, \mathit{em}_{\langle \overline{s}_n \rangle, s_n}\langle x^n_{\overline{s}_n} \rangle \rangle \rangle$$
$$\stackrel{s}{=} \mathit{em}_{\langle s' \rangle, s''}\langle f_{w',s'}\langle \mathit{em}_{\langle \overline{s}_1 \rangle, s'_1}\langle x^1_{\overline{s}_1} \rangle, \ldots, \mathit{em}_{\langle \overline{s}_n \rangle, s'_n}\langle x^n_{\overline{s}_n} \rangle \rangle \rangle$$

for $f_{w,s} \sim_F f'_{w',s'}$, where $w' = \langle s'_1, \ldots, s'_n \rangle$ and $w = \langle s_1, \ldots, s_n \rangle$, with $\overline{w} \leq w, w'$ for some $\overline{w} = \langle \overline{s}_1, \ldots, \overline{s}_n \rangle$, and $s, s' \leq s''$

Predicate-monotonicity:

$$\forall \{x^1_{\overline{s}_1}, \ldots, x^n_{\overline{s}_n}\} .\ p_w\langle \mathit{em}_{\langle \overline{s}_1 \rangle, s_1}\langle x^1_{\overline{s}_1} \rangle, \ldots, \mathit{em}_{\langle \overline{s}_n \rangle, s_n}\langle x^n_{\overline{s}_n} \rangle \rangle$$
$$\Leftrightarrow p'_{w'}\langle \mathit{em}_{\langle \overline{s}_1 \rangle, s'_1}\langle x^1_{\overline{s}_1} \rangle, \ldots, \mathit{em}_{\langle \overline{s}_n \rangle, s'_n}\langle x^n_{\overline{s}_n} \rangle \rangle$$

for $p_w \sim_P p'_{w'}$, where $w' = \langle s'_1, \ldots, s'_n \rangle$ and $w = \langle s_1, \ldots, s_n \rangle$, with $\overline{w} \leq w, w'$ for some $\overline{w} = \langle \overline{s}_1, \ldots, \overline{s}_n \rangle$

Proposition 3.7. *In every subsorted model the following axiom holds:*

Embedding-injectivity: $\forall \{x_s, y_s\} . \mathit{em}_{\langle s \rangle, s'}\langle x_s \rangle \stackrel{e}{=} \mathit{em}_{\langle s \rangle, s'}\langle y_s \rangle \Rightarrow x_s \stackrel{e}{=} y_s$ *for all* $s \leq s'$*.*

Proof. Let us assume that $\mathit{em}_{\langle s \rangle, s'}\langle x_s \rangle \stackrel{e}{=} \mathit{em}_{\langle s \rangle, s'}\langle y_s \rangle$ is satisfied by a subsorted model M with respect to some assignment ρ for $\{x_s, y_s\}$; then the equation $\mathit{pr}_{\langle s' \rangle, s}\langle \mathit{em}_{\langle s \rangle, s'}\langle x_s \rangle \rangle \stackrel{e}{=} \mathit{pr}_{\langle s' \rangle, s}\langle \mathit{em}_{\langle s \rangle, s'}\langle y_s \rangle \rangle$ is also satisfied by M w.r.t. ρ. By the projection axioms, both $\mathit{pr}_{\langle s' \rangle, s}\langle \mathit{em}_{\langle s \rangle, s'}\langle x_s \rangle \rangle \stackrel{e}{=} x_s$ and $\mathit{pr}_{\langle s' \rangle, s}\langle \mathit{em}_{\langle s \rangle, s'}\langle y_s \rangle \rangle \stackrel{e}{=} y_s$ must be satisfied by M w.r.t. ρ, since M is a subsorted model. Therefore, $x_s \stackrel{e}{=} y_s$ must be satisfied by M w.r.t. ρ. □

Notice that, as usual, the same class of models may be described by several different axiomatic theories that may be convenient for different purposes; for instance, one might wish to use a certain automatic deduction system that requires a specific form of axioms. For example, we can replace the projection-injectivity axiom by the following axiom:

Defined-projection: $\forall \{x_{s'}\} . D(\mathit{pr}_{\langle s' \rangle, s}\langle x_{s'} \rangle) \Rightarrow \mathit{em}_{\langle s \rangle, s'}\langle \mathit{pr}_{\langle s' \rangle, s}\langle x_{s'} \rangle \rangle \stackrel{e}{=} x_{s'}$ for all $s \leq s'$.

This gives a set of *existentially conditioned equations* (ECE) in the sense of Burmeister [10].

Proposition 3.8. *Projections are always undefined on elements that are not in the image of the corresponding embedding.*

Proof. Follows from the defined-projection axiom. □

Subsorted Σ-model morphisms are ordinary $\Sigma^{\#}$-homomorphisms, that is the *category* of subsorted Σ-models $\mathbf{SubMod}(\Sigma)$ is the full subcategory of $\mathbf{Mod}(\Sigma^{\#})$, whose objects are all the many-sorted models satisfying the above axioms.

Therefore, all notions defined for many-sorted models, and in particular all functors having many-sorted model categories as source, apply to subsorted models as well, via the embedding of $\mathbf{SubMod}(\Sigma)$ into $\mathbf{Mod}(\Sigma^{\#})$.

The *reduct* of a subsorted Σ'-model along a subsorted signature morphism $\sigma : \Sigma \to \Sigma'$ is the many-sorted reduct along the signature morphism $\sigma^{\#}$, and similarly for subsorted Σ'-model morphisms. This defines a functor $\mathbf{SubMod} : \mathbf{SubSig}^{op} \to \mathbf{CAT}$.

Since subsorted signature morphisms preserve overloading relations, the reducts of subsorted models satisfy the above axioms, too, and are, hence, subsorted models.

Notice that **SubMod** is *not* finitely cocontinuous. The following counterexample is from [45]. Let Σ be the signature with sorts s and t and no operations, and let Σ_1 be the extension of Σ by the subsort relation $s \leq t$. Then the pushout

$$\begin{tikzcd}
\Sigma \arrow[r] \arrow[d] & \Sigma_1 \arrow[d] \\
\Sigma_1 \arrow[r] & \Sigma_1
\end{tikzcd}$$

in **SubSig** is not mapped to a pullback in **CAT** since two models of Σ_1 that are compatible w.r.t. the inclusion of Σ may interpret the subsort injection differently.

3.1.3 Sentences

The subsorted Σ-sentences are simply the many-sorted $\Sigma^{\#}$-sentences.

For a subsorted signature Σ, the *subsorted sentences* are the ordinary many-sorted sentences (as defined in Sect. 2.1.3) for the associated many-sorted signature $\Sigma^{\#}$. The subsorted translation of sentences along a subsorted signature morphism σ is the ordinary many-sorted translation along $\sigma^{\#}$. That is, subsorted sentences are given by the composition of the ordinary functor yielding many-sorted sentences with the functor $(.)^{\#}$, i.e. $\mathbf{SubSen} : \mathbf{SubSig} \to \mathbf{Set}$ is defined as $\mathbf{Sen} \circ (.)^{\#}$.

A *subsorted enrichment* (Δ, Ψ) relative to a subsorted signature Σ consists of a subsorted extension Δ relative to Σ and a set Ψ of subsorted sentences over $\Sigma \cup \Delta$.

Satisfaction over Σ is many-sorted satisfaction over $\Sigma^{\#}$.

Subsorted satisfaction for a subsorted signature Σ is ordinary many-sorted satisfaction for the signature $\Sigma^{\#}$. Since reducts and sentence translation are ordinary many-sorted reducts and sentence translation, the satisfaction condition is satisfied for the subsorted case as well.

Theorem 3.9. SubSig*,* **SubMod***,* **SubSen** *and* $\models$ *form an institution with* **SubSig** *being finitely cocomplete.*

Proof. Straightforward. Cocompleteness follows from Prop. 3.3. □

The relationship between formula satisfaction and isomorphism is the same for subsorted as for standard many-sorted models.

Proposition 3.10. *Satisfaction is preserved and reflected by isomorphisms: if M, M' are subsorted Σ-models such that $M \cong M'$ and ψ is a subsorted Σ-sentence, then $M \models \psi$ iff $M' \models \psi$.*

Proof. Straightforward. □

3.2 Signature Declarations

The rest of this chapter gives the abstract syntax of the additional subsorting constructs used in *subsorted* basic specifications, and defines their intended interpretation, extending what was provided for many-sorted specifications in Chap. 2. Unless otherwise stated, the rules for static and model semantics of the constructs given in Chap. 2 are the same, with the convention that the signatures in each rule are enriched by an implicit subsorting relation that is not modified by the rules.

As in the many-sorted case, a well-formed subsorted basic specification `BASIC-SPEC` of the CASL language determines an extension Δ to the local environment Σ together with a set of $\Sigma \cup \Delta$-sentences of the form described in Sect. 3.1.3. This in turn describes the class of all subsorted $\Sigma \cup \Delta$-models that satisfy those sentences.

3.2.1 Sorts

`SORT-ITEM` has three more alternatives than in Chap. 2:

```
SORT-ITEM ::= ... | SUBSORT-DECL | ISO-DECL | SUBSORT-DEFN
```

Subsort Declarations

A subsort declaration declares all the sorts, as well as the embedding of each subsort into the supersort, which must be a different sort.

```
SUBSORT-DECL ::= subsort-decl SORT+ SORT
```

$$\boxed{\Sigma \vdash \texttt{SUBSORT-DECL} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{s_i \neq s \text{ for all } i = 1, \ldots, n \qquad \preceq_S = \{(s_i, s) \mid 1 \leq i \leq n\}}{\begin{array}{r}(S, \mathit{TF}, \mathit{PF}, P, \leq) \vdash \texttt{subsort-decl}\, s_1 \ldots s_n\; s \rhd \\ ((\{s_1, \ldots, s_n, s\}, \emptyset, \emptyset, \emptyset, \preceq_S), \emptyset)\end{array}}$$

The first condition checks that each subsort is distinct from the supersort.

Isomorphism Declarations

An isomorphism declaration declares all the sorts, as well as their embeddings as subsorts of each other.

```
ISO-DECL ::= iso-decl SORT+
```

$$\boxed{\Sigma \vdash \texttt{ISO-DECL} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{c}s_i \neq s_j \text{ for all } i \neq j \qquad n \geq 2 \\ \preceq_S = \{(s_i, s_j) \mid 1 \leq i \leq n \wedge 1 \leq j \leq n\}\end{array}}{(S, \mathit{TF}, \mathit{PF}, P, \leq) \vdash \texttt{iso-decl}\, s_1 \ldots s_n \rhd ((\{s_1, \ldots, s_n\}, \emptyset, \emptyset, \emptyset, \preceq_S), \emptyset)}$$

The first condition checks that each sort occurs once and the second checks that there are at least two sorts.

Subsort Definitions

A subsort definition provides an explicit specification of the values of the subsort, stating which values of the supersort belong to the subsort by means of a formula with one free variable in it.

```
SUBSORT-DEFN ::= subsort-defn SORT VAR SORT FORMULA
```

$$\boxed{\Sigma \vdash \texttt{SUBSORT-DEFN} \rhd (\Delta, \Psi)}$$

(Δ, Ψ) is an enrichment relative to Σ.

$$\frac{\begin{array}{ccc}(S, TF, PF, P, \leq) = \Sigma & s' \in S & FV(F) \subseteq \{v : s'\} \\ \Sigma \vdash \texttt{subsort-decl}\, s\ s' \rhd (\Delta, \emptyset) & & \Sigma \cup \Delta, \emptyset \vdash F' \rhd \varphi\end{array}}{\Sigma \vdash \texttt{subsort-defn}\, s\ v\ s'\ F \rhd (\Delta, \{\varphi\})}$$

where F' is

$$\texttt{quantification}\,\texttt{forall}\,\texttt{var-decl}\, v\ s'(\texttt{equivalence}\, F\ (\texttt{membership}\, v\ s))$$

The second condition checks that s' is already declared, the third that the all variables in F but v are explicitly bound, and the fourth and the fifth[1] conditions are equivalent to expanding the subsort definition into the subsort declaration plus an axiom stating which values of the supersort belong to the subsort.

3.2.2 Datatypes

Alternatives

Datatype declarations have a new kind of alternative, for the embedding of known subsorts into the datatype.

```
ALTERNATIVE ::= ... | SUBSORTS
SUBSORTS    ::= subsorts SORT+
```

$$\boxed{\Sigma, s \vdash \texttt{SUBSORTS} \rhd (\Delta, \Delta', \Psi)}$$

s is required to be a sort in Σ. $(\Delta \cup \Delta', \Psi)$ is an enrichment relative to Σ.

$$\frac{s_i \in S \text{ for all } i = 1, \ldots, n \qquad \preceq_S = \{(s_i, s) \mid 1 \leq i \leq n\}}{(S, TF, PF, P, \leq), s \vdash \texttt{subsort}\ s_1 \ldots s_n \rhd ((\emptyset, \emptyset, \emptyset, \emptyset, \preceq_S), \emptyset, \emptyset), \{\}}$$

The first condition checks that the subsorts are declared elsewhere.

Note that this kind of alternative does not contribute a constructor, in contrast to the other kinds – the subsort embeddings are treated as *implicit* constructors, see below.

[1] Actually, if free variables different from v appear in F, this condition cannot be satisfied, so that the third condition is superfluous and required only for stressing the point.

Free Datatype Declarations

The embeddings of subsorts are treated as implicit constructors.

The rule for free datatypes given in Sect. 2.3.4 has to be modified to treat the embedding of subsorts as constructors as well.

Thus, the rule:

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W \\ (S, \mathit{TF}, \mathit{PF}, P) = \Sigma \qquad (S', \mathit{TF}', \emptyset, P') = \Delta \qquad S'' = S \cup S'\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{free-datatype DATATYPE-ITEMS} \rhd \\ \quad (\Delta \cup \Delta', \Psi \cup \{\mathit{injective}(f_{w,s}) \mid w \in \mathit{FinSeq}(S''), s \in S'', f \in \mathit{TF}'_{w,s}\} \\ \qquad\qquad \cup \{\mathit{disjoint\text{-}ranges}(f_{w,s}, g_{w',s}) \\ \qquad\qquad\qquad \mid w, w' \in \mathit{FinSeq}(S''), s \in S'', f \in \mathit{TF}'_{w,s}, g \in \mathit{TF}'_{w',s} \\ \qquad\qquad\qquad\quad \text{such that } w \neq w' \text{ or } f \neq g\} \\ \qquad\qquad \cup \{\mathit{undefined\text{-}selection}(f_{w,s}, g_{\langle s\rangle,s'}) \\ \qquad\qquad\quad \mid f_{w,s}, f'_{w',s} \in \mathit{Dom}(W), g_{\langle s\rangle,s'} \in W(f'_{w',s}) \setminus W(f_{w,s})\} \\ \qquad\qquad \cup \{(S', \mathit{complete}(\mathit{TF}', \mathit{FinSeq}(S'') \times S''))\}\,)\end{array}}$$

is replaced by:

$$\frac{\begin{array}{c}\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W \\ (S, \mathit{TF}, \mathit{PF}, P, \leq) = \Sigma \qquad (S', \mathit{TF}', \emptyset, P', \prec) = \Delta \qquad S'' = S \cup S' \\ \overline{F} = \mathit{TF}' \cup \{(\langle s\rangle, s') \mapsto \mathit{em} \mid s \prec s'\}\end{array}}{\begin{array}{l}\Sigma \vdash \texttt{free-datatype DATATYPE-ITEMS} \rhd \\ \quad (\Delta \cup \Delta', \Psi \cup \{\mathit{injective}(f_{w,s}) \mid w \in \mathit{FinSeq}(S''), s \in S'', f \in \mathit{TF}'_{w,s}\} \\ \qquad\qquad \cup \{\mathit{disjoint\text{-}ranges}(f_{w,s}, g_{w',s}) \\ \qquad\qquad\qquad \mid w, w' \in \mathit{FinSeq}(S''), s \in S'', f \in \overline{F}_{w,s}, g \in \overline{F}_{w',s} \\ \qquad\qquad\qquad\quad \text{such that } w \neq w' \text{ or } f \neq g\} \\ \qquad\qquad \cup \{\mathit{undefined\text{-}selection}(f_{w,s}, g_{\langle s\rangle,s'}) \\ \qquad\qquad\quad \mid f_{w,s}, f'_{w',s} \in \mathit{Dom}(W), g_{\langle s\rangle,s'} \in W(f'_{w',s}) \setminus W(f_{w,s})\} \\ \qquad\qquad \cup \{\mathit{undefined\text{-}selection}(\mathit{em}_{\langle s''\rangle,s}, g_{\langle s\rangle,s'}) \\ \qquad\qquad\quad \mid s'' \neq s \in S'', s'' \prec s, \\ \qquad\qquad\qquad\quad f_{w,s} \in \mathit{Dom}(W), g_{\langle s\rangle,s'} \in W(f_{w,s})\} \\ \qquad\qquad \cup \{(S', \mathit{complete}(\overline{F}, \mathit{FinSeq}(S'') \times S''))\}\,)\end{array}}$$

where *injective*, *disjoint-ranges* and *undefined-selection* are defined as in Sect. 2.3.4.

Theorem 3.11. *Consider a declaration* `free-datatype DATATYPE-ITEMS`*, a signature Σ and a model class $\mathcal{M}$ over Σ, and suppose*

$$\begin{array}{c}\Sigma \vdash \texttt{DATATYPE-ITEMS} \rhd (\Delta, \Delta', \Psi), W \\ \Sigma \vdash \texttt{free-datatype DATATYPE-ITEMS} \rhd (\Delta \cup \Delta', \Psi')\end{array}$$

such that `DATATYPE-ITEMS` *fulfills the following conditions (all referring to fully qualified symbols):*

- *The sorts in Δ (and hence the constructors in Δ and the selectors in Δ') are not in the local environment Σ;*
- *Any selector in Δ' is total only when the same selector is present in all* `ALTERNATIVE`*s for that sort;*
- *Each constructor in Δ and each selector in Δ' is in the overloading relation of $\Sigma \cup \Delta \cup \Delta'$ only with itself; and*
- *Distinct sorts in Δ have no common subsort in $\Sigma \cup \Delta \cup \Delta'$.*

Let $\mathcal{C}$ be the full subcategory of $\mathbf{Mod}(\Sigma \cup \Delta \cup \Delta')$ *containing those $(\Sigma \cup \Delta \cup \Delta')$-models M'' such that $M'' \models \psi$ for all $\psi \in \Psi$, and let $\mathcal{M}'$ and $\mathcal{M}''$ be the $(\Sigma \cup \Delta \cup \Delta')$-model classes*

$$\begin{aligned}\mathcal{M}' = \{&(\Sigma \cup \Delta \cup \Delta')\text{-}\mathit{model}\ M' \\ &\mid M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} \in \mathcal{M}\ \mathit{and}\ M' \in \mathcal{C}\ \mathit{is\ free\ over}\ M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} \\ &\quad \mathit{w.r.t.}\ .|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} : \mathcal{C} \to \mathbf{Mod}(\Sigma)\} \\ \mathcal{M}'' = \{&(\Sigma \cup \Delta \cup \Delta')\text{-}\mathit{model}\ M' \\ &\mid M'|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} \in \mathcal{M}\ \mathit{and}\ M' \models \psi'\ \mathit{for\ all}\ \psi' \in \Psi'\}\end{aligned}$$

Then $\mathcal{M}' = \mathcal{M}''$.

Proof. Employing the notation of the above rule, let us use S' for the sorts in Δ and $\overline{F}$ for the functions from Δ and the embeddings relative to the subsorts explicitly given in Δ (that is, for the constructors).

Let us first show that $\mathcal{M}'' \subseteq \mathcal{M}'$.

It suffices to show that every model $M \in \mathcal{M}''$, satisfying the axioms in Ψ', is free w.r.t. $.|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'} : \mathcal{C} \to \mathbf{Mod}(\Sigma)$, that is, that it belongs to $\mathcal{C}$ and that any homomorphism h from its Σ-reduct to the Σ-reduct of a model $N \in \mathcal{C}$ extends uniquely to a homomorphism from M to N on the overall signature.

Since, by construction, $\Psi \subseteq \Psi'$, $\mathcal{M}'' \subseteq \mathcal{C}$ hence $M \in \mathcal{C}$. Thus, we only have to prove the existence of the $\Sigma \cup \Delta \cup \Delta'$-homomorphism extending h.

As the sorts in S' are new in the environment, to extend the homomorphism we have to give the new components h_s for each $s \in S'$.

Because of the sort-generation constraint in Ψ' (see the last axiom of the rule), for every element a in the carrier of sort s (with $s \in S'$), there is a term $t = f\langle t_1, \ldots, t_n\rangle$ containing only function symbols from $\overline{F}$ and variables of sorts not in S' such that $[\![t]\!]_\rho = a$ for some assignment ρ into $M|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'}$. Let us show that such a term t is unique (up to variable renaming). Since $t = f\langle t_1, \ldots, t_n\rangle$ and $[\![t]\!]_\rho = a$, we have $a = f^M(a_1, \ldots, a_n)$ for some $f \in \overline{F}$ and $a_i = [\![t_i]\!]_\rho$. Moreover,

- because of the disjointness axioms in Ψ' the leading symbol is unique
- because of injectivity axioms in Ψ' the argument tuple is unique

For each $a_i \in s_i^M$, if $s_i \notin S'$, then t_i is a variable; otherwise, we can recursively apply this argument to a_i (and t_i).

Therefore, for each $s \in S'$ and each $a \in s^M$ there exists a unique term t_a containing only function symbols from $\overline{F}$ and variables of sorts not in S'

such that $[\![t_a]\!]_\rho = a$ for some assignment ρ into $M|_{\Sigma \hookrightarrow \Sigma \cup \Delta \cup \Delta'}$. Hence, we can (inductively) define h_s as $h_s(a) = h_s(f^M(a_1, \ldots, a_n)) = f^N(h(a_1), \ldots, h(a_n))$ and since the decomposition is unique, this equation yields a function. Moreover, such h_s is the unique possible candidate homomorphism because it is defined by the homomorphism condition itself.

Finally, h satisfies the homomorphism condition for each function symbol f in $\Sigma'^{\#}$, where $\Sigma' = \Sigma \cup \Delta \cup \Delta'$, because:

- if f is a function symbol in $\Sigma^{\#}$, as h is a subsorted Σ-homomorphism, that is a many-sorted $\Sigma^{\#}$-homomorphism;
- if f belongs to $\overline{F}$, by construction;
- if f belongs to Δ' (i.e. is a selector), because selectors in M are defined only on the image of their constructor(s), where their value is fixed by the constructor/selector axioms in Ψ, so that they have to be defined in N as well (with compatible values); and
- if f is a projection, because projections are defined only on the image of their embeddings and their value is fixed by the axioms required from M and N to be subsorted models.

The homomorphism condition for predicates is trivially fulfilled, since Δ and Δ' do not contain any new predicates, and for those in $\Sigma^{\#}$, we already know that it holds[2].

Vice versa, let us show now that $\mathcal{M}' \subseteq \mathcal{M}''$; let us fix an arbitrary model $M' \in \mathcal{M}'$. Let us consider the Σ-model $M = M'|_\Sigma$ and directly build an extension M'', satisfying the axioms of Ψ'. Thus, because of the previous point, such an extension M'' is free. Then, as M' and M'' are isomorphic, being both free models over M, and isomorphic models satisfy the same formulas, we get that M' satisfies the axioms of Ψ'. The extension M'' is built as follows:

- For all $s \in S'$ we (simultaneously) inductively define the carriers to consist of formal applications of embeddings and constructors, that is elements of the form $f(a_1, \ldots, a_n)$ where $f \in \overline{F}_{\langle s_1, \ldots, s_n \rangle, s}$ and $a_i \in s_i^M$ if $s_i \notin S'$ and otherwise $a_i \in s_i^{M''}$. Thus, the a_i may be formal applications as well as elements of M.
- The interpretation of functions and predicates from $\Sigma^{\#}$ is as in M.
- The interpretation of embeddings and constructors from Δ yields the formal interpretation (so the resulting model is generated by $\overline{F}$, and the axioms of injectiveness and disjointness hold).

[2] The homomorphism condition for membership predicates, possibly introduced by the new subsorting relations in Δ, is obviously satisfied, because of the axioms required from M and N to be subsorted models that define the validity of the membership predicate.

- The interpretation of any selector from Δ' is defined for values in the image(s) of its corresponding constructor(s) so that the selector-constructor axioms are satisfied[3]. For all other values it is undefined (so that the axioms concerning undefinedness of selectors are satisfied).
- Analogously, the interpretation of projections is defined only on the image of their embeddings (and their composition yields the identity) and the membership predicates are true only on the image of the corresponding embedding (so the axioms required for M'' to be a $\Sigma \cup \Delta \cup \Delta'$-model are satisfied).

Finally such construction yields a subsorted model M''. The identities required for overloaded symbols are satisfied for the functions and predicates in Σ, because M is the reduct of a subsorted model M' on the overall signature, and they are satisfied for the constructors and selectors in $\Delta \cup \Delta'$, because for them the overloading relation is the identity. Moreover, the transitivity axioms are satisfied because the alternatives being sorts do not have common subsorts. □

Below we give an example of a typical free datatype declaration and discuss some ways in which specifications may accidentally become inconsistent.

Here is an example of a free datatype declaration where all alternatives are sorts that are declared beforehand, which corresponds to a declaration of a disjoint union type:

free type *Vehicle* ::= *sort Car* | *sort Bicycle*

The semantics of the free datatype declaration is the following enrichment (relative to a signature Σ containing the two sorts *Car* and *Bicycle*):

$$\begin{aligned}(\Delta, \{&\forall x{:}\mathit{Car}, y{:}\mathit{Bicycle}.\neg(\mathit{em}_{\langle \mathit{Car}\rangle, \mathit{Vehicle}}\langle x\rangle \overset{\mathrm{s}}{=} \mathit{em}_{\langle \mathit{Bicycle}\rangle, \mathit{Vehicle}}\langle y\rangle),\\ &\forall x{:}\mathit{Car}, y{:}\mathit{Bicycle}.\neg(\mathit{em}_{\langle \mathit{Bicycle}\rangle, \mathit{Vehicle}}\langle x\rangle \overset{\mathrm{s}}{=} \mathit{em}_{\langle \mathit{Car}\rangle, \mathit{Vehicle}}\langle y\rangle),\\ &(\{\mathit{Vehicle}\}, \overline{F}, \mathit{id}_{\Sigma\cup\Delta})\})\end{aligned}$$

where Δ is the signature extension (relative to Σ) containing the sort *Vehicle*, no function symbols or predicate symbols, and the subsort relation $\leq$ which is the reflexive and transitive closure of $\{(\mathit{Car}, \mathit{Vehicle}), (\mathit{Bicycle}, \mathit{Vehicle})\}$. Moreover, $\overline{F}$ contains the two embedding function symbols $\mathit{em}_{\langle \mathit{Car}\rangle, \mathit{Vehicle}}$ and $\mathit{em}_{\langle \mathit{Bicycle}\rangle, \mathit{Vehicle}}$. The first two sentences are from the *disjoint-ranges* condition; they are equivalent. The third sentence is a sort generation constraint which requires each value of sort *Vehicle* to be produced by either $\mathit{em}_{\langle \mathit{Car}\rangle, \mathit{Vehicle}}\langle x\rangle$ or $\mathit{em}_{\langle \mathit{Bicycle}\rangle, \mathit{Vehicle}}\langle y\rangle$ for some assignment of a value of sort *Car* to x or a value of sort *Bicycle* to y, respectively. Since there are no explicit constructors and selectors, there are no injectivity sentences (but embeddings that play the same role as constructors are instead required to be

[3] There are no contradictions in these axioms, as elements built from different constructors are different because of the disjointness axioms in Ψ', and the same selector cannot belong to two different components of the same alternative.

injective by the axioms for subsorted models), no sentences relating selectors to their constructors (but projections that play the same role as selectors are instead related to their embeddings by the axioms for subsorted models) and no sentences expressing *undefined-selection* conditions (but the *disjoint-ranges* conditions and the axioms for subsorted models ensure that it is undefined to apply a projection function to a value generated by an embedding function from a different sort, e.g. $\forall x{:}\mathit{Car}.\neg D(pr_{\langle Vehicle\rangle,Bicycle}\langle em_{\langle Car\rangle,\,Vehicle}\langle x\rangle\rangle))$.

Notice that if, after a correct definition of a free datatype s, other basic items are added to the specification that modify the overloading relation or the subsort relation, then the resulting overall specification may be inconsistent because the explicit axioms of the free datatype and the implicit axioms for subsorted models may conflict. To be more specific, the disjointness axioms require the constructors and embeddings of a free datatype to have disjoint images, but at the same time some applications of its constructors and embeddings may be required to yield the same value by the implicit monotonicity axioms or transitivity axioms for subsorted models. This may happen if additional basic items cause a constructor to enter the overloading relation with a function that has as range sort a subsort of one of the alternatives of the free datatype, or if a common subsort of two alternatives (sorts) of the free datatype is added.

Let us consider two instances of this kind of problem. The first case exemplifies the danger of modifying the subsorting relation. The second one demonstrates how problems may arise from an extension of the overloading relation.

Consider the following specification of the union of two sorts s_1 and s_2:

sorts s_1, s_2;
free type *Union* ::= *sort* s_1 | *sort* s_2

Now we add a sort representing an intersection of the two sorts, and declare a constant a of that new sort to guarantee that the intersection is non-empty:

sort *Intersection* $<$ s_1; *Intersection* $<$ s_2;
op a : *Intersection*

Then

$$em_{\langle s_1\rangle,\,Union}\langle em_{\langle Intersection\rangle,s_1}\langle a_{\langle\rangle,Intersection}\rangle\rangle$$

is equal to

$$em_{\langle s_2\rangle,\,Union}\langle em_{\langle Intersection\rangle,s_2}\langle a_{\langle\rangle,Intersection}\rangle\rangle$$

as both are equal to

$$em_{\langle Intersection\rangle,\,Union}\langle a_{\langle\rangle,Intersection}\rangle$$

due to the transitivity axioms for subsorted models. However, the images of $em_{\langle s_1\rangle,\,Union}$ and $em_{\langle s_2\rangle,\,Union}$ are required to be disjoint by the axioms of the free datatype. Hence, the resulting specification is inconsistent.

Now, consider the following specification of non-empty lists of elements taken from a sort s:

free type $\mathit{NEList} ::= \mathit{sort}\ s \mid \mathit{cons}(\mathit{first} : s;\ \mathit{rest} : \mathit{NEList})$

Now, let us refine the sort s into non-empty lists of elements from another sort *Elem*:

free type $s ::= \mathit{sort}\ \mathit{Elem} \mid \mathit{cons}(\mathit{first} : \mathit{Elem};\ \mathit{rest} : s)$

Then, if e is any term of sort *Elem*, we have that

$$\mathit{cons}_{\langle s,\mathit{NEList}\rangle,\mathit{NEList}}\langle \mathit{em}_{\langle \mathit{Elem}\rangle,s}\langle e\rangle, \mathit{em}_{\langle \mathit{Elem}\rangle,\mathit{NEList}}\langle e\rangle\rangle$$

and

$$\mathit{em}_{\langle s\rangle,\mathit{NEList}}\langle \mathit{cons}_{\langle \mathit{Elem},s\rangle,s}\langle e, \mathit{em}_{\langle \mathit{Elem}\rangle,s}\langle e\rangle\rangle\rangle$$

are equal, because of the function-monotonicity axioms for subsorted models. However, the images of $\mathit{cons}_{\langle s,\mathit{NEList}\rangle,\mathit{NEList}}$ and $\mathit{em}_{\langle s\rangle,\mathit{NEList}}$ are required to be disjoint by the axioms of the free datatype. Hence, the resulting specification is inconsistent.

Notice that both of these examples are quite artificial and, though the technical problem may be subtle, intuitively the inconsistencies arise from a change of the expected meaning of the given free datatype specifications.

Consider for instance the first example. Since we are requiring the datatype *Union* to be free, we are actually describing the *disjoint* union of the sorts s_1 and s_2, as with *Vehicle* above. Thus, adding a non-empty sort for their intersection is actually an attempt to change the understanding of what the union is and hence correctly results in an inconsistency.

Analogously, the second example is based on a naive refinement of the sort s where the problem comes from using the same name for the constructors of the two levels of lists. Due to the axiom of function monotonicity, the choice of the same name corresponds to requiring that the two constructors represent the same function on common arguments (up to embedding). This, in turn, means that we are describing (in an overly complex way) lists of elements of sort *Elem*, instead of lists of lists, as one would expect from the structure of the specification (and as it would be if we had used a different name for the lower level constructor).

Sort Generation

The treatment of a sort generation `SORT-GEN` is as in Sect. 2.3.5 except that the embeddings of subsorts are treated as implicit constructors.

In order to treat the embedding operations as declared operations, the following rule, given in Sect. 2.3.5:

$$\frac{\begin{array}{c} \Sigma \vdash \texttt{SIG-ITEMS}_1 \rhd (\Delta_1, \Delta'_1, \Psi_1) \\ \cdots \\ \Sigma \cup \Delta_1 \cup \Delta'_1 \cup \cdots \cup \Delta_{n-1} \cup \Delta'_{n-1} \vdash \texttt{SIG-ITEMS}_n \rhd (\Delta_n, \Delta'_n, \Psi_n) \\ (S, \mathit{TF}, \mathit{PF}, P) = \Delta = \Delta_1 \cup \cdots \cup \Delta_n \qquad \Delta' = \Delta'_1 \cup \cdots \cup \Delta'_n \\ (S', \mathit{TF}', \mathit{PF}', P') = \Sigma \cup \Delta \cup \Delta' \qquad S \neq \emptyset \end{array}}{\begin{array}{l} \Sigma \vdash \texttt{sort-gen SIG-ITEMS}_1 \ \ldots \ \texttt{SIG-ITEMS}_n \rhd \\ \quad (\Delta \cup \Delta', \Psi_1 \cup \cdots \cup \Psi_n \cup \{(S, \mathit{complete}(\mathit{TF} \cup \mathit{PF}, \mathit{FinSeq}(S') \times S'))\}\,) \end{array}}$$

is replaced by:

$$\frac{\begin{array}{c} \Sigma \vdash \texttt{SIG-ITEMS}_1 \rhd (\Delta_1, \Delta'_1, \Psi_1) \\ \cdots \\ \Sigma \cup \Delta_1 \cup \Delta'_1 \cup \cdots \cup \Delta_{n-1} \cup \Delta'_{n-1} \vdash \texttt{SIG-ITEMS}_n \rhd (\Delta_n, \Delta'_n, \Psi_n) \\ (S, \mathit{TF}, \mathit{PF}, P, \leq) = \Delta = \Delta_1 \cup \cdots \cup \Delta_n \qquad \Delta' = \Delta'_1 \cup \cdots \cup \Delta'_n \\ (S', \mathit{TF}', \mathit{PF}', P', \leq') = \Sigma \cup \Delta \cup \Delta' \qquad S \neq \emptyset \\ \mathit{TF}'' = \mathit{TF} \cup \{(\langle s \rangle, s') \mapsto \mathit{em} \mid s \leq s'\} \end{array}}{\begin{array}{l} \Sigma \vdash \texttt{sort-gen SIG-ITEMS}_1 \ \ldots \ \texttt{SIG-ITEMS}_n \rhd \\ \quad (\Delta \cup \Delta', \Psi_1 \cup \cdots \cup \Psi_n \cup \{(S, \mathit{complete}(\mathit{TF}'' \cup \mathit{PF}, \mathit{FinSeq}(S') \times S'))\}\,) \end{array}}$$

Note that projections, like selectors, are not included in the set of functions of the sort generation constraint, since the axioms of subsorted models guarantee that projections can never generate new elements.

3.3 Axioms

3.3.1 Atomic Formulas

`ATOM` has one more alternative than in Chap. 2:

```
ATOM ::= ... | MEMBERSHIP
```

As for many-sorted specifications, an atomic formula is well-formed (with respect to the current declarations) if it is well-sorted and expands to a unique atomic formula for constructing sentences of the underlying institution – but now for subsorted specifications, uniqueness is required only up to an *equivalence* on atomic formulas and terms. This equivalence is the least one including fully-qualified terms that are the same up to profiles of operation symbols in the overloading relation $\sim_F$ and embedding, and fully-qualified atomic formulas that are the same up to the profiles of predicate symbols in the overloading relation $\sim_P$ and embedding.

The relaxation of the well-formedness requirement for subsorted specifications means that the rule for well-formedness of atomic formulas given in Sect. 2.5.3 has to be modified, requiring the existence of a unique expansion *up to the equivalence relation* defined below.

Thus, the following rule

$$\frac{\text{there is a unique } \varphi \text{ such that } \Sigma, X \vdash \texttt{ATOM} \rhd \varphi \qquad \Sigma, X \vdash \texttt{ATOM} \rhd \varphi}{\Sigma, X \vdash \texttt{ATOM}\,\text{qua}\,\texttt{FORMULA} \rhd \varphi}$$

is replaced by

$$\frac{\varphi \simeq \varphi' \text{ for all } \varphi' \text{ such that } \Sigma, X \vdash \texttt{ATOM} \rhd \varphi' \qquad \Sigma, X \vdash \texttt{ATOM} \rhd \varphi}{\Sigma, X \vdash \texttt{ATOM}\,\text{qua}\,\texttt{FORMULA} \rhd \varphi}$$

where $\simeq$ is defined below. In the first premise – which is equivalent to $\forall \varphi'.(\Sigma, X \vdash \texttt{ATOM} \rhd \varphi' \text{ implies } \varphi \simeq \varphi')$ – the static semantics of ATOM occurs in a negative position. This potential problem can be eliminated in the same way as in the many-sorted case.

The equivalence between fully-qualified terms is the congruence generated by the following axioms:

$$\begin{array}{l}
em_{\langle s\rangle,s}\langle t\rangle \simeq t \\
em_{\langle s'\rangle,s''}\langle em_{\langle s\rangle,s'}\langle t\rangle\rangle \simeq em_{\langle s\rangle,s''}\langle t\rangle \text{ for } s \leq s' \leq s'' \\
pr_{\langle s'\rangle,s}\langle em_{\langle s\rangle,s'}\langle t\rangle\rangle \simeq t \text{ for } s \leq s' \\
em_{\langle s\rangle,s''}\langle f_{w,s}\langle em_{\langle \overline{s}_1\rangle,s_1}\langle x^1_{\overline{s}_1}\rangle, \ldots, em_{\langle \overline{s}_n\rangle,s_n}\langle x^n_{\overline{s}_n}\rangle\rangle\rangle \simeq \\
\qquad\qquad \simeq em_{\langle s'\rangle,s''}\langle f_{w',s'}\langle em_{\langle \overline{s}_1\rangle,s'_1}\langle x^1_{\overline{s}_1}\rangle, \ldots, em_{\langle \overline{s}_n\rangle,s'_n}\langle x^n_{\overline{s}_n}\rangle\rangle\rangle \\
\qquad \text{for } f_{w,s} \sim_F f'_{w',s'}, w = \langle s_1, \ldots, s_n\rangle, w' = \langle s'_1, \ldots, s'_n\rangle, \text{with} \\
\qquad \overline{w} \leq w, w' \text{ for } \overline{w} = \langle \overline{s}_1, \ldots, \overline{s}_n\rangle, \text{and } s, s' \leq s''
\end{array}$$

The equivalence between atomic formulas is the natural extension of the equivalence between fully qualified terms and is inductively defined by:

Equivalence:

$$\frac{}{\varphi \simeq \varphi} \qquad\qquad \frac{\varphi \simeq \varphi'}{\varphi' \simeq \varphi} \qquad\qquad \frac{\varphi \simeq \varphi' \qquad \varphi' \simeq \varphi''}{\varphi \simeq \varphi''}$$

Replacement of equivalent terms:

$$\frac{t \simeq t' \qquad sort(t) = sort(t')}{D(t) \simeq D(t')}$$

$$\frac{t_1 \simeq t'_1 \qquad t_2 \simeq t'_2 \qquad sort(t_1) = sort(t_2) = sort(t'_1) = sort(t'_2)}{t_1 \overset{\text{e}}{=} t_2 \simeq t'_1 \overset{\text{e}}{=} t'_2}$$

$$\frac{t_1 \simeq t'_1 \qquad t_2 \simeq t'_2 \qquad sort(t_1) = sort(t_2) = sort(t'_1) = sort(t'_2)}{t_1 \overset{\text{s}}{=} t_2 \simeq t'_1 \overset{\text{s}}{=} t'_2}$$

$$\frac{t \simeq t' \qquad sort(t) = s' = sort(t') \qquad s \le s'}{in(s)_{\langle s'\rangle}\langle t\rangle \simeq in(s)_{\langle s'\rangle}\langle t'\rangle}$$

$$\frac{t_i \simeq t'_i \qquad sort(t_i) = s_i = sort(t'_i) \text{ for } 1 \le i \le n \qquad w = \langle s_1, \ldots, s_n\rangle}{p_w\langle t_1, \ldots, t_n\rangle \simeq p_w\langle t'_1, \ldots, t'_n\rangle}$$

Embedding and projection:

$$\frac{sort(t) = s = sort(t') \qquad s \le s'}{em_{\langle s\rangle,s'}\langle t\rangle \overset{\mathrm{e}}{=} em_{\langle s\rangle,s'}\langle t'\rangle \simeq t \overset{\mathrm{e}}{=} t'}$$

$$\frac{sort(t) = s = sort(t') \qquad s \le s'}{em_{\langle s\rangle,s'}\langle t\rangle \overset{\mathrm{s}}{=} em_{\langle s\rangle,s'}\langle t'\rangle \simeq t \overset{\mathrm{s}}{=} t'}$$

$$\frac{\begin{array}{c} pr_{\langle s'\rangle,s}\langle t'\rangle \simeq pr_{\langle s''\rangle,s}\langle t''\rangle \\ sort(t') = s' \qquad sort(t'') = s'' \qquad s \le s' \qquad s \le s'' \end{array}}{in(s)_{\langle s'\rangle}\langle t'\rangle \simeq in(s)_{\langle s''\rangle}\langle t''\rangle}$$

$$\frac{sort(t) = s \qquad s \le s' \qquad s \le s''}{D(em_{\langle s\rangle,s'}\langle t\rangle) \simeq D(em_{\langle s\rangle,s''}\langle t\rangle)}$$

Predicate overloading:

$$\frac{\begin{array}{c} sort(t_i) = \overline{s}_i \qquad p_w \sim_P p'_{w'} \\ w = \langle s_1, \ldots, s_n\rangle \qquad w' = \langle s'_1, \ldots, s'_n\rangle \qquad \langle \overline{s}_1, \ldots, \overline{s}_n\rangle \le w, w' \end{array}}{p_w\langle em_{\langle \overline{s}_1\rangle,s_1}\langle t_1\rangle, \ldots, em_{\langle \overline{s}_n\rangle,s_n}\langle t_n\rangle\rangle \simeq p'_{w'}\langle em_{\langle \overline{s}_1\rangle,s'_1}\langle t_1\rangle, \ldots, em_{\langle \overline{s}_n\rangle,s'_n}\langle t_n\rangle\rangle}$$

Membership

A membership formula is well-sorted if the term is well-sorted for a supersort of the specified sort. It expands to an application of the pre-declared predicate symbol for testing values of the sort of the given term for membership in the image of the embedding of the given sort.

```
MEMBERSHIP ::= membership TERM SORT
```

$$\frac{(S, TF, PF, P, \le), X \vdash \mathtt{TERM} \rhd t \qquad sort(t) = s' \qquad s \le s'}{(S, TF, PF, P, \le), X \vdash \mathtt{membership}\,\mathtt{TERM}\, s \rhd in(s)_{\langle s'\rangle}\langle t\rangle}$$

3.3.2 Terms

Term formation is extended by letting a well-sorted term of a subsort be regarded as a well-sorted term of a supersort.

$$\frac{(S, \mathit{TF}, \mathit{PF}, P, \leq), X \vdash \texttt{TERM} \rhd t \qquad \mathit{sort}(t) = s \qquad s \leq s'}{(S, \mathit{TF}, \mathit{PF}, P, \leq), X \vdash \texttt{TERM} \rhd \mathit{em}_{\langle s \rangle, s'}\langle t \rangle}$$

Analogously for sorted terms we have

$$\frac{(S, \mathit{TF}, \mathit{PF}, P, \leq), X \vdash \texttt{TERM} \rhd t \qquad \mathit{sort}(t) = s \qquad s \leq s'}{(S, \mathit{TF}, \mathit{PF}, P, \leq), X \vdash \texttt{sorted-term}\,\texttt{TERM}\, s' \rhd \mathit{em}_{\langle s \rangle, s'}\langle t \rangle}$$

Casts

Terms have one more alternative, representing the (partial) projection of values from a supersort onto a subsort. A cast term is well-sorted if the term is well-sorted for a supersort of the sort. It expands to an application of the pre-declared operation symbol for projecting the sort of the term to the given sort.

```
TERM ::= ... | CAST
CAST ::= cast TERM SORT
```

$$\frac{(S, \mathit{TF}, \mathit{PF}, P, \leq), X \vdash \texttt{TERM} \rhd t \qquad \mathit{sort}(t) = s' \qquad s \leq s'}{(S, \mathit{TF}, \mathit{PF}, P, \leq), X \vdash \texttt{cast}\,\texttt{TERM}\, s \rhd \mathit{pr}_{\langle s' \rangle, s}\langle t \rangle}$$

4

Structured Specification Semantics

The semantics of a well-formed structured specification is of the same form as that of a basic specification: a signature Σ together with a class of Σ-models. While the model class of a basic specification can be characterized by a set of sentences, this is not possible for structured specifications, due to the presence of constructs such as hiding and freeness. Hence, the semantics of structured specifications is essentially based on model classes.

The structure of a specification is *not* reflected in its models: it is used only to present the specification in a modular style. (Specification of the *architecture* of models in the CoFI framework is addressed by architectural specifications, see Chap. I:5, with the semantics given in Chap. 5.)

Within a structured specification, the *current signature* may vary. It is also called the *local environment*. On the other hand, the current association between names and the specifications that they reference is called the *global environment*.

For the semantics of structured specifications (in particular, those involving hiding) the axiom of choice is assumed. Note that hiding is as expressive as second-order existential quantification, the semantics of which may depend on properties of the background set theory, like the axiom of choice. For instance, one can express the proposition that every vector space has a basis as a view in CASL (see Chap. V:9), and indeed the well-formedness of this view is equivalent to the axiom of choice.

4.1 Structuring Concepts

The CASL structuring concepts and constructs and their semantics do not depend on the choice of institution used to write basic specifications. This means that Chaps. 2 and 3 are orthogonal to Chap. 4 (and also to Chaps. 5 and 6). Therefore, CASL basic specifications as given in Chaps. 2 and 3 can be restricted to sublanguages or extended in various ways without the need to reconsider or to change Chaps. 4, 5, and 6.

The concepts defined in Chaps. 2 and 3 lead to a CASL institution, here formalized as an *institution with qualified symbols* (Sect. 4.1.1). Institution independence of the semantics of structured specifications is achieved by introducing a vocabulary of *derived notions* (Sect. 4.1.2) that can be defined over an arbitrary institution with qualified symbols, and writing the semantics in terms of this vocabulary. At various places, we detail what these notions mean in the CASL institution.

The derived notions are used mainly in the computation of *signature morphisms* out of symbol maps, which is addressed in Sect. 4.1.3. Signature morphisms (and the corresponding reducts on models) occur at many places in the semantics of structured specifications. For instance, hiding some symbols in a specification involves a signature morphism that injects the non-hidden symbols into the original signature; the models, after hiding the symbols, are the reducts of the original models along this morphism. Translation goes the other way: the reducts of models over the translated signature back along the morphism give the original models. CASL uses symbol maps to denote signature morphisms; the semantics of symbol maps is of course institution specific (Sect. 4.5).

Finally, in Sect. 4.1.4, we slightly modify (in an institution independent way) the institution, equipping so-called *extended signatures* with a set of symbols as a further component.

Let us stress that, though the following details are a bit complicated, once the reader has accepted the concept of signature unions and has learned that signature morphisms are generated by maps between symbols in a more or less expected way, the semantics of structured and architectural specifications may be understood in terms of an arbitrary institution, with the qualified symbol structure put aside. For the first reading of this chapter, it may therefore be a good idea to continue with the institution independent structuring concepts in Sect. 4.1.5, and refer to the sections before that only when needed.

4.1.1 Institution Independence and the CASL Institution

> The CASL structuring concepts and constructs and their semantics do not depend on a specific institution; hence, they are given here in an institution-independent way.

In order to achieve institution independence, below we introduce a minimal vocabulary of notions required for the semantics of structured specifications. Together they form the CASL *institution with qualified symbols.* In order to change the framework of basic specifications, one just has to change the institution with qualified symbols. While Chaps. 2 and 3 are completely institution-specific, in Chap. 4 *we explicitly indicate those parts that are institution-specific.* These are:

- the definition of the CASL institution with qualified symbols;
- some propositions about what some institution-independent derived notions mean in the CASL institution (Props. 4.1, 4.2, 4.4, 4.5, and 4.6);
- the semantic rule for treating a basic specification as a structured specification (see page 204); and
- some of the rules for symbol lists and maps in Sect. 4.5.

A more formal treatment of this issue can be found in [39].

We now come to the components of the CASL institution with qualified symbols. We first recall the components of the CASL institution with symbols that have been introduced in Chaps. 2 and 3.

Category of signatures and signature morphisms: A category **SubSig** of signatures with subsorting was introduced in Chap. 3.

Sentences: A sentence functor $\mathbf{Sen}\colon \mathbf{Sig} \to \mathbf{Set}$ was introduced in Chap. 2, see page 135, and extended to subsorted signatures via composition with the functor $(.)^{\#}\colon \mathbf{SubSig} \to \mathbf{Sig}$, yielding a functor $\mathbf{SubSen}\colon \mathbf{SubSig} \to \mathbf{Set}$.

Models: A model functor $\mathbf{Mod}\colon \mathbf{Sig}^{op} \to \mathbf{CAT}$ was introduced in Chap. 2, see page 130, and extended to subsorted signatures via composition with the functor $(.)^{\#}\colon \mathbf{SubSig} \to \mathbf{Sig}$ and further restriction by axioms, yielding a functor $\mathbf{SubMod}\colon \mathbf{SubSig}^{op} \to \mathbf{CAT}$.

Satisfaction: A satisfaction relation $\models_{\Sigma} \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ was introduced in Chap. 2, see page 135, and extended to subsorted signatures, models and sentences via composition with the functor $(.)^{\#}\colon \mathbf{SubSig} \to \mathbf{Sig}$.

Signature symbols: A set of signature symbols *SigSym* and a faithful functor $|.|\colon \mathbf{Sig} \to \mathbf{Set}$ giving, for each signature Σ, a set of signature symbols $|\Sigma| \subseteq \mathit{SigSym}$, and for each signature morphism $\sigma\colon \Sigma \to \Sigma'$, a translation of signature symbols $|\sigma|\colon |\Sigma| \to |\Sigma'|$, were introduced in Chap. 2, pages 126 and 128, and extended to subsorted signatures in Chap. 3, page 171.

We now extend this institution with more notions, mainly regarding the computation of signature morphisms out of symbol maps.

Symbols:

$$\begin{array}{rll}
k \in & \mathit{SymKind} = & \{\mathit{implicit}, \mathit{sort}, \mathit{fun}, \mathit{pred}\} \\
SY \in & \mathit{Sym} = & \\
s \in & & \mathit{Sort} \uplus \\
f^{\mathsf{t}}_{ws} \in & & \mathit{QualFunName} \uplus \\
f^{\mathsf{p}}_{ws} \in & & \mathit{QualFunName} \uplus \\
p_w \in & & \mathit{QualPredName} \uplus \\
(k, \mathit{Ident}) \in & & (\mathit{SymKind} \times \mathtt{ID})
\end{array}$$

Identifiers can be regarded as symbols using the injection

$$IDAsSym\colon \mathtt{ID} \to Sym$$

defined by $IDAsSym(Ident) = (implicit, Ident)$.

For simplicity, we regard `SORT-ID` as a subset of `ID`, although there is only an embedding between the two sets. The embedding maps `WORDS` to `id WORDS` and `comp-sort-id WORDS ID+` to `id` (`comp-mix-token ID+`).

Factorization of signature symbol functor: There is a partial function $SymAsSigSym : Sym \rightharpoonup SigSym$, such that the object part of the signature symbol functor $|.|$: $\mathbf{SubSig} \to \mathbf{Set}$ can be can be factorized through a symbol function

$$||.|| :: |\mathbf{SubSig}| \to |\mathbf{Set}|,$$

such that $||\Sigma|| \subseteq Sym$ and for each $\Sigma \in |\mathbf{SubSig}|$, $SymAsSigSym(||\Sigma||)$ is defined and equal to $|\Sigma|$.

We define $SymAsSigSym$ as follows:

$$\begin{array}{ll}
SymAsSigSym(s) = s & s \in Sort \\
SymAsSigSym(f^{\mathsf{t}}_{ws}) = f_{ws} & f_{ws} \in QualFunName \\
SymAsSigSym(f^{\mathsf{p}}_{ws}) = f_{ws} & f_{ws} \in QualFunName \\
SymAsSigSym(p_w) = p_w & p_w \in QualPredName \\
SymAsSigSym(k, Ident) = undefined &
\end{array}$$

If $\Sigma = (S, TF, PF, P, \leq)$, we define $||\Sigma|| \subseteq Sym$ as follows:

$$\begin{aligned}
||\Sigma|| = S &\cup \{f^{\mathsf{t}}_{ws} \mid ws \in FinSeq(S) \times S, f \in TF_{ws}\} \\
&\cup \{f^{\mathsf{p}}_{ws} \mid ws \in FinSeq(S) \times S, f \in PF_{ws}\} \\
&\cup \{p_w \mid w \in FinSeq(S), p \in P_w\}
\end{aligned}$$

Signature symbols matching symbols: The matching relation

$$matches \subseteq SigSym \times Sym$$

between signature symbols and symbols is the least relation satisfying

- s matches $(implicit, s)$ for $s \in Sort$, f_{ws} matches $(implicit, f)$ for $f_{ws} \in QualFunName$ and $f \in \mathtt{ID}$, and p_w matches $(implicit, p)$ for $p_w \in QualPredName$ and $p \in \mathtt{ID}$,
- s matches $(sort, s)$ for $s \in Sort$, f_{ws} matches (fun, f) for $f_{ws} \in QualFunName$ and $f \in \mathtt{ID}$, and p_w matches $(pred, p)$ for $p_w \in QualPredName$ and $p \in \mathtt{ID}$,
- s matches s for $s \in Sort$,
- f_{ws} matches f^{p}_{ws} for $f_{ws} \in QualFunName$,
- f_{ws} matches f^{t}_{ws} for $f_{ws} \in QualFunName$,
- p_w matches p_w for $p_w \in QualPredName$.

Names of signature symbols: There is a function $nameSigSym \rightharpoonup \mathtt{ID}$ assigning a name $name(SSY)$ to each signature symbol SSY, such that, whenever $name(SSY)$ is defined,

$$\mathit{SSY} \text{ matches } \mathit{IDAsSym}(\mathit{name}(\mathit{SSY}))$$

It is defined as follows:

$\mathit{name}(s) = s$, for $s \in \mathit{Sort}$,

$$\mathit{name}(f_{ws}) = \begin{cases} f, & \text{if } f_{ws} \in \mathit{QualFunName} \text{ and } f \in \mathtt{ID} \\ \text{undefined}, & \text{otherwise} \end{cases},$$

$$\mathit{name}(p_{w}) = \begin{cases} p, & \text{if } p_{w} \in \mathit{QualPredName} \text{ and } p \in \mathtt{ID} \\ \text{undefined}, & \text{otherwise} \end{cases}.$$

Note that *name* is defined on the actual signature symbols of a signature, since these exclude the special embedding, projection and membership symbols introduced in Sect. 3.1.1, and the latter are the only ones that are not based on an `ID`.

Empty signature: The empty signature, denoted by $\emptyset$, has been defined in Sect. 1.1. We define $\mathcal{M}_{\perp} = \mathbf{Mod}(\emptyset)$. $\mathcal{M}_{\perp}$ consists of exactly one object.

Signature unions: The union of signatures has been defined in Sect. 3.1.1, see page 170. The union of Σ_1 and Σ_2, written $\Sigma_1 \cup \Sigma_2$, comes with two injections $\iota_{\Sigma_1 \subseteq \Sigma_1 \cup \Sigma_2}$ and $\iota_{\Sigma_2 \subseteq \Sigma_1 \cup \Sigma_2}$.

In the Casl institution, unions are always defined, but in other frameworks, this need not be the case. In the rules for the semantics below, when we use a union inside a condition (e.g. within the premises of a rule) it is implicitly assumed that the definedness of the union is added with a conjunction to this condition (e.g. yielding an extra premise).

Generating signature morphisms: A signature morphism

$$\sigma = (\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}}) : (S, \mathit{TF}, \mathit{PF}, P, \leq) \to (S', \mathit{TF}', \mathit{PF}', P', \leq')$$

is said to be *generating* if

- $|\sigma|$ is surjective,
- σ detects totality (i.e. $f' \in \mathit{TF}'_{w's'}$ implies that there is some ws with $\sigma^{\mathsf{S}}(ws) = ws$ and some $f \in \mathit{TF}_{ws}$ with $\sigma^{\mathit{TF}}_{ws}(f) = f'$), and
- $\leq'$ is the least pre-order on S' satisfying

$$\sigma^{\mathsf{S}}(s_1) \leq' \sigma^{\mathsf{S}}(s_2) \text{ if } s_1 \leq s_2.$$

It can be shown that generating signature morphisms $\sigma\colon \Sigma_1 \to \Sigma_2$ are *final*, that is, any function $h\colon |\Sigma_2| \to |\Sigma_3|$ is a signature morphism provided that $h \circ |\sigma|\colon |\Sigma_1| \to |\Sigma_3|$ is[1].

It is possible to replace the Casl institution with a different one, provided that it comes equipped with the components listed above.

4.1.2 Derived Notions

We now introduce, in an institution independent way, some further notions, derived from those introduced above, that are needed for the semantics of

[1] Here, $h\colon |\Sigma_2| \to |\Sigma_3|$ is said to be a signature morphism if there is a signature morphism $\theta\colon \Sigma_2 \to \Sigma_3$ with $|\theta| = h$.

structured specifications. At various places, we detail what these notions mean in the CASL institution.

Subsignatures, signature inclusions and extensions: We say that a signature morphism $\iota\colon \Sigma \to \Sigma'$ is a *signature inclusion* if $|\iota|$ is an inclusion (of $|\Sigma|$ into $|\Sigma'|$). If there exists a signature inclusion from Σ to Σ', we call Σ a *subsignature* of Σ' and write $\Sigma \subseteq \Sigma'$. Notice that in this case the signature inclusion is unique, and we denote it by $\iota_{\Sigma\subseteq\Sigma'}$. Σ' is then called an *extension* of Σ.

Reducts along signature inclusions: Given a subsignature Σ of a signature Σ' and a Σ'-model M, we write $M|_\Sigma$ for $M|_{\iota_{\Sigma\subseteq\Sigma'}}$. Similarly, given a Σ'-homomorphism $h\colon M \to M'$, we write $h|_\Sigma$ for $h|_{\iota_{\Sigma\subseteq\Sigma'}}$.

Final signature unions: A *sink* is a pair of morphisms with common codomain. A sink $(\sigma_1\colon \Sigma_1 \to \Sigma, \sigma_2\colon \Sigma_2 \to \Sigma)$ is called *final*, if for each function $h\colon |\Sigma| \to |\Sigma'|$, h is a signature morphism[2] provided that $h \circ |\sigma_1|\colon |\Sigma_1| \to |\Sigma'|$ and $h \circ |\sigma_2|\colon |\Sigma_2| \to |\Sigma'|$ are. A union is said to be *final* if the sink consisting of the two inclusions is final.

Proposition 4.1 (CASL-specific). *Let $\equiv_F$ and $\equiv_P$ denote the transitive closures of the overloading relations $\sim_F$ and $\sim_P$, respectively. Then signature morphisms preserve $\equiv_F$ and $\equiv_P$.*

Proof. By induction on the transitive closure, using the fact that signature morphisms preserve $\sim_F$ and $\sim_P$ as basis for the induction. □

Proposition 4.2 (CASL-specific). *For a union $\Sigma_1 \cup \Sigma_2$, the following are equivalent:*

1. *The union is final.*
2. *The relation $\equiv_F^{\Sigma_1\cup\Sigma_2}$ of $\Sigma_1 \cup \Sigma_2$ is the transitive closure of the union of the relations $\equiv_F^{\Sigma_1}$ and $\equiv_F^{\Sigma_2}$ (and similarly for $\equiv_P$).*

Proof. $\neg(2) \Rightarrow \neg(1)$: We here argue for $\equiv_F$ only, the argument for $\equiv_P$ being entirely analogous. Let $\equiv_F$ be the transitive closure of the union of the relations $\equiv_F^{\Sigma_1}$ and $\equiv_F^{\Sigma_2}$. Suppose without loss of generality that $f\colon w \to s \equiv_F^{\Sigma_1\cup\Sigma_2} f\colon w' \to s'$, but not $f\colon w \to s \equiv_F^{\Sigma_1} f\colon w' \to s'$. Extend $\Sigma_1 \cup \Sigma_2$ to Σ by adding, for any w'', s'' such that $f\colon w'' \to s'' \equiv_F f\colon w \to s$, an operation symbol $g\colon w'' \to s''$ not in $\Sigma_1 \cup \Sigma_2$. Let $\sigma\colon |\Sigma_1 \cup \Sigma_2| \to |\Sigma|$ be the identity, except that $f\colon w'' \to s''$ is mapped to $g\colon w'' \to s''$ for any $f\colon w'' \to s'' \equiv_F f\colon w \to s$. Since $\sigma(f\colon w' \to s') = f\colon w' \to s' \not\equiv_F^\Sigma g\colon w \to s = \sigma(f\colon w \to s)$, σ does not preserve the overloading relations and therefore is not a signature morphism. But $\sigma \circ \iota_{\Sigma_1\subseteq\Sigma_1\cup\Sigma_2}$ and $\sigma \circ \iota_{\Sigma_2\subseteq\Sigma_1\cup\Sigma_2}$ are signature morphisms (any two function symbols in the overloading relation of $\Sigma_1 \cup \Sigma_2$ are either both $\equiv_F f\colon w \to s$ or both $\not\equiv_F f\colon w \to s$). Thus, the union is not final.

[2] See the previous footnote.

(2) ⇒ (1): In order to show finality of the union, consider a function $\sigma\colon |\Sigma_1 \cup \Sigma_2| \to |\Sigma|$ such that $\sigma \circ \iota_{\Sigma_1 \subseteq \Sigma_1 \cup \Sigma_2}$ and $\sigma \circ \iota_{\Sigma_2 \subseteq \Sigma_1 \cup \Sigma_2}$ are signature morphisms. Now the subsorting relation and the transitive closure of overloading relations are the transitive closure of the respective component-wise union. Therefore, σ preserves these relations (and also profiles of symbols) because $\sigma \circ \iota_{\Sigma_1 \subseteq \Sigma_1 \cup \Sigma_2}$ and $\sigma \circ \iota_{\Sigma_2 \subseteq \Sigma_1 \cup \Sigma_2}$ do so. Thus, σ is a signature morphism. □

Signature symbol maps: A signature symbol map is a binary relation on the set of signature symbols[3]:

$$h \in \mathit{SigSymMap} = \mathit{FinSet}(\mathit{SigSym} \times \mathit{SigSym})$$

Fully qualified symbols: A symbol SY is said to be *fully qualified* if $SY \in \mathit{Dom}(\mathit{SymAsSigSym})$).

Symbol maps:

$$r \in \mathit{SymMap} = \mathit{Set}(\mathit{Sym} \times \mathit{Sym})^4$$

Symbol map induced by a signature morphism: Given a signature morphism $\sigma\colon \Sigma_1 \to \Sigma_2$, the symbol map induced by σ is defined to be

$$\begin{aligned} ||\sigma|| = \{(SY_1, SY_2) \mid\ & SY_i \in ||\Sigma_i|| \text{ fully qualified for } i = 1, 2, \text{ and} \\ & |\sigma|(\mathit{SymAsSigSym}(SY_1)) \text{ matches } SY_2\} \end{aligned}$$

Signature morphisms matching symbol maps: Given a signature symbol SSY and a symbol map r, we say that SSY *is not directly mapped by* r if $\mathit{SymAsSigSym}^{-1}(SSY) \cap \mathit{dom}(r) = \emptyset$.

Given a signature morphism $\sigma\colon \Sigma \to \Sigma'$ and a symbol map $r \subseteq \mathit{Sym} \times \mathit{Sym}$, we say that σ *matches* r if:

- for all $(SY, SY') \in r$ with SY fully qualified, we have $SY \in ||\Sigma||$ and $|\sigma|(\mathit{SymAsSigSym}(SY))$ matches SY', and
- for all $(SY, SY') \in r$ such that SY is not fully qualified,
  - $SY \in ||\Sigma||$,
  - there exists some $SSY \in |\Sigma|$ not directly mapped by r that matches SY, and moreover
  - for all such $SSY \in |\Sigma|$ not directly mapped by r and matching SY, we have $|\sigma|(SSY)$ matches SY'.

Signature morphisms leaving names unchanged: Given a signature morphism $\sigma\colon \Sigma \to \Sigma'$ and a symbol map $r \subseteq \mathit{Sym} \times \mathit{Sym}$, we say that σ *leaves names unchanged outside* r if for any $SSY \in |\Sigma|$ matching no $SY \in \mathit{dom}(r)$, $\mathit{name}(|\sigma|(SSY)) = \mathit{name}(SSY)$.

Compatibility of signature morphisms: A pair of signature morphisms $\sigma_1\colon \Sigma_1 \to \Sigma_1'$ and $\sigma_2\colon \Sigma_2 \to \Sigma_2'$ are *compatible* if their signature symbol maps $|\sigma_1|$ and $|\sigma_2|$ coincide on the intersection $|\Sigma_1| \cap |\Sigma_2|$ of their domains (i.e., $\mathit{graph}(|\sigma_1|) \cup \mathit{graph}(|\sigma_2|)$ is a function).

[3] A motivation of this choice can be found in [39].

[4] We also need infinite symbol mappings because of the function *Ext*, see page 196.

Unions of signature morphisms: Given a pair of signature morphisms $\sigma_1 \colon \Sigma_1 \to \Sigma'_1$ and $\sigma_2 \colon \Sigma_2 \to \Sigma'_2$, if $\mathit{graph}(|\sigma_1|) \cup \mathit{graph}(|\sigma_2|)$ is (the graph of) a signature morphism from $\Sigma_1 \cup \Sigma_2$ to $\Sigma'_1 \cup \Sigma'_2$, then $\sigma_1 \cup \sigma_2$ is defined to be this signature morphism (it is unique since $|.|$ is faithful), otherwise, it is undefined.

$$\begin{tikzcd}
\Sigma_1 \arrow[rr, "\sigma_1"] \arrow[dr, hook] & & \Sigma'_1 \arrow[dr, hook] & \\
& \Sigma_1 \cup \Sigma_2 \arrow[rr, "\sigma_1 \cup \sigma_2"] & & \Sigma'_1 \cup \Sigma'_2 \\
\Sigma_2 \arrow[rr, "\sigma_2"] \arrow[ur, hook] & & \Sigma'_2 \arrow[ur, hook] &
\end{tikzcd}$$

Proposition 4.3. *Given signature morphisms $\sigma_1 \colon \Sigma_1 \to \Sigma'_1$ and $\sigma_2 \colon \Sigma_2 \to \Sigma'_2$, consider the following conditions:*

1. $\Sigma_1 \cup \Sigma_2$ is final and σ_1 and σ_2 are compatible.
2. $\sigma_1 \cup \sigma_2$ is defined.
3. σ_1 and σ_2 are compatible.

We have that (1) ⇒ (2) ⇒ (3) *(but the converse implications do not hold in general).*

Proof. (1) ⇒ (2): Compatibility of σ_1 and σ_2 means that $\mathit{graph}(|\sigma_1|) \cup \mathit{graph}(|\sigma_2|)$ is a function. By finality of the union $\Sigma_1 \cup \Sigma_2$, it is also a signature morphism.
(2) ⇒ (3): If $\sigma_1 \cup \sigma_2$ exists, $\mathit{graph}(|\sigma_1|) \cup \mathit{graph}(|\sigma_2|)$ is a function. Thus, σ_1 and σ_2 are compatible. □

Extension of symbol maps: There is a function

$$\mathit{Ext} \colon \mathit{FinSet}(\mathit{Sym} \times \mathit{Sym}) \to \mathit{Set}(\mathit{Sym} \times \mathit{Sym})$$

giving for each symbol map $r \subseteq \mathit{Sym} \times \mathit{Sym}$ an extension $\mathit{Ext}(r) \subseteq \mathit{Sym} \times \mathit{Sym}$ of r, i.e. $r \subseteq \mathit{Ext}(r)$. At this point, we take $\mathit{Ext}(r)$ to be just r. This will be modified in Sect. 4.6 when giving the semantics of compound identifiers.

4.1.3 Signature Morphisms

A set *SSYs* of signature symbols in $|\Sigma|$ determines the inclusion of the *smallest* subsignature $\Sigma|_{\mathit{SSYs}}$ of Σ that contains these symbols. (When an operation or predicate symbol is included, all the sorts in its profile have to be included too.)

A subsignature Σ' of a signature Σ is said to be *full* if every subsignature of Σ with the same set of names as Σ' is a subsignature of Σ'.

We call a set of signature symbols $SSYs \subseteq |\Sigma|$ *closed* in Σ if there is a subsignature Σ' of Σ with the set of signature symbols $SSYs$, i.e. such that $|\Sigma'| = SSYs$.

Given a set $SSYs \subseteq |\Sigma|$, if there is a unique full subsignature Σ' of Σ such that $|\Sigma'|$ is the smallest set containing $SSYs$ and closed in Σ, then Σ' is called the *signature generated in Σ by SSYs* and is denoted by $\Sigma|_{SSYs}$.

Proposition 4.4 (Casl-specific). *The following are equivalent for a subsignature Σ of Σ':*

1. *Σ is a full subsignature of Σ'.*
2. *Σ inherits the subsort relation from Σ' (i.e. $\leq \, = \, \leq' \cap (S \times S)$) and totality of function symbols (i.e. a function symbol of Σ that is total in Σ' must already be total in Σ – formally this means $PF_{ws} \cap TF'_{ws} = \emptyset$ for each $ws \in S^* \times S$).*

Proof. (1) ⇒ (2): Let $\Sigma = (S, TF, PF, P, \leq)$ be a full subsignature of $\Sigma' = (S', TF', PF', P', \leq')$. Then $\Sigma'' = (S, TF'', PF'', P, \leq'')$ with

- $\leq'' = \leq' \cap (S \times S))$,
- $TF''_{ws} = TF'_{ws} \cap (TF_{ws} \cup PF_{ws})$,
- $PF''_{ws} = PF'_{ws} \cap (TF_{ws} \cup PF_{ws})$

is a subsignature of Σ' with $|\Sigma| = |\Sigma''|$. By fullness, $\Sigma'' \subseteq \Sigma$, hence, $\leq' \cap (S \times S) \subseteq \leq$ (while the converse inclusion holds since $\Sigma \subseteq \Sigma'$). Moreover, by $\Sigma'' \subseteq \Sigma$, $PF''_{ws} = PF'_{ws} \cap (TF_{ws} \cup PF_{ws}) \subseteq PF_{ws}$, which implies $TF_{ws} \cap PF'_{ws} = \emptyset$ (note that TF_{ws} and PF_{ws} are disjoint).

(2) ⇒ (1): Let $\Sigma = (S, TF, PF, P, \leq' \cap (S \times S))$ be a subsignature of $\Sigma' = (S', TF', PF', P', \leq')$. Given any subsignature $\Sigma'' = (S'', TF'', PF'', P'', \leq'')$ of Σ' with $|\Sigma| = |\Sigma''|$, Σ and Σ'' can differ only in the subsort relation and in totality of function symbols. Since $\leq'' \subseteq \leq'$ by the subsignature property, and $S = S''$ by $|\Sigma| = |\Sigma''|$, we have $\leq'' \subseteq \leq' \cap (S \times S)$. Moreover, if $f \in TF''_{ws}$, by the subsignature property, also $f \in TF'_{ws}$. Since $TF_{ws} \cup PF_{ws} = TF''_{ws} \cup PF''_{ws}$, $TF_{ws} \cap PF_{ws} = \emptyset$ and moreover, $PF_{ws} \cap TF'_{ws} = \emptyset$ by assumption, we get $f \in TF_{ws}$. Thus, $\Sigma'' \subseteq \Sigma$. □

Proposition 4.5 (Casl-specific). *For any set $SSYs \subseteq |\Sigma|$, $\Sigma|_{SSYs}$ exists.*

Proof. Let $\Sigma = (S, TF, PF, P, \leq)$. Given $SSYs \subseteq |\Sigma|$, let $SSYs'$ be the union of $SSYs$ with all the sort symbols occurring in profiles of signature symbols in $SSYs$. Obviously, $SSYs'$ is the smallest set containing $SSYs$ and closed in Σ. Let $\Sigma|_{SSYs} = (S', TF', PF', P', \leq) \cap (S' \times S'))$ where

$$\begin{aligned} S' &= \{s \in S \mid s \in SSYs'\}, \\ TF'_{ws} &= \{f \in TF_{ws} \mid f_{ws} \in SSYs'\}, \\ PF'_{ws} &= \{f \in PF_{ws} \mid f_{ws} \in SSYs'\}, \\ P'(w) &= \{p \in P(w) \mid p_w \in SSYs'\}. \end{aligned}$$

By Prop. 4.4, $\Sigma|_{SSYs}$ is a full subsignature of Σ with set of signature symbols $SSYs'$. Clearly, $\Sigma|_{SSYs}$ is unique with this property. Together with the property that $SSYs'$ is the smallest set containing $SSYs$ and closed in Σ, this gives the desired result. □

A set of signature symbols $SSYs$ in $|\Sigma|$ also determines the inclusion of the *largest* subsignature $\Sigma|^{SSYs}$ of Σ that does not contain any of these signature symbols. (When a sort is not included, no operation or predicate symbol with that sort in its profile can be included either.)

Given a set $SSYs \subseteq |\Sigma|$, if there is a unique full subsignature Σ' of Σ such that $|\Sigma'|$ is the largest set disjoint from $SSYs$ and closed in Σ, then Σ' is called the *signature co-generated in Σ by SSYs* and is denoted by $\Sigma|^{SSYs}$.

Proposition 4.6 (CASL-specific). *For any set $SSYs \subseteq |\Sigma|$, $\Sigma|^{SSYs}$ exists.*

Proof. Analogous to the proof of Prop. 4.5. □

A mapping r of symbols in $||\Sigma||$ determines the morphism $r|^{\Sigma}$ from Σ that extends this mapping with identity maps for all the remaining names in $||\Sigma||$. In case such a signature morphism does not exist, the enclosing construct is ill-formed.

Given a signature Σ and a symbol map $r \subseteq Sym \times Sym$, a generating signature morphism $\sigma\colon \Sigma \to \Sigma'$ matching r and leaving names unchanged outside r is called *the signature morphism from Σ induced by r*, provided that σ is unique with these properties. If it exists, we will denote it by $r|^{\Sigma}$.

Given signatures Σ and Σ', a mapping r of symbols in $||\Sigma||$ to symbols in $||\Sigma'||$ determines the unique signature morphism $r|^{\Sigma}_{\Sigma'}$ from Σ to Σ' that extends the given mapping, and then is the identity, as far as possible, on common names of Σ and Σ'. (Mapping an operation or predicate symbol implies mapping the sorts in the profile consistently.) In case such a signature morphism does not exist or is not unique, the enclosing construct is ill-formed.

Let signatures Σ and Σ' and a symbol map $r \subseteq Sym \times Sym$ be given.

Now r determines the set of all signature morphisms $\sigma\colon \Sigma \to \Sigma'$ such that there is some set $SSYs$ of signature symbols with

1. σ matches r and leaves names unchanged on $SSYs$ (where the latter means that $name(|\sigma|(SSY)) = name(SSY)$ for each $SSY \in SSYs$), and
2. $SSYs$ is maximal with the property that for some signature morphism $\theta : \Sigma \to \Sigma'$, θ matches r and leaves names unchanged on $SSYs$ (the maximality here implements the 'as far as possible' requirement on page 35 of the Summary).

If this set is a singleton, its unique element (also called $r|^{\Sigma}_{\Sigma'}$) is *the signature morphism from Σ to Σ' induced by r*.

When a generic specification (given by the inclusion $\Delta\colon \Sigma \rightarrow \Sigma'$ of its formal parameters into the body) is instantiated, the fitting arguments yield a signature morphism $\sigma\colon \Sigma \rightarrow \Sigma_A$, which is then extended to a signature morphism $\sigma(\Delta)\colon \Sigma' \rightarrow \Sigma_A \cup \Sigma_A(\Delta)$ that is applicable to the signature Σ' of the body of the generic specification. The resulting signature $\Sigma_A \cup \Sigma_A(\Delta)$ is the union of the fitting arguments with the translated body.
An instantiation of a generic specification is not well-formed if the result signature is not a pushout of the body and argument signatures.

At the level of symbols, the construction is basically a set-theoretic union, where some side conditions ensure that this gives a set-theoretic pushout. Since we use final signature morphisms and final unions to lift this to the level of signatures, and final lifts of colimits are again colimits, we can show that the construction, if defined, always yields a pushout.

Given a signature extension $\Delta\colon \Sigma \rightarrow \Sigma'$ and a signature morphism $\sigma\colon \Sigma \rightarrow \Sigma_A$, if:

- the signature morphism $r|^{\Sigma'}$ from Σ' induced by
$$r = \mathit{Ext}(||\sigma||) \cap (||\Sigma'|| \times \mathit{Sym})$$
exists[5] (let its target be denoted by $\Sigma_A(\Delta)$);
- the union $\Sigma_A \cup \Sigma_A(\Delta)$ exists, and moreover, is final[6];
- $|\Sigma_A| \cap |\Sigma_A(\Delta)| \subseteq |\sigma|(|\Sigma|)$ [7]; and
- $\mathit{ker}(|(r|^{\Sigma'})|) \subseteq \mathit{ker}(|\sigma|)$ [8],

then $\iota_{\Sigma_A(\Delta)\subseteq\Sigma_A\cup\Sigma_A(\Delta)} \circ r|^{\Sigma'}$ is called the *extension of σ along Δ*, denoted by $\sigma(\Delta)\colon \Sigma' \rightarrow \Sigma_A \cup \Sigma_A(\Delta)$.

Proposition 4.7. *If the extension of $\sigma\colon \Sigma \rightarrow \Sigma_A$ along $\Delta\colon \Sigma \rightarrow \Sigma'$ exists, then*

[5] This may fail to exist for several reasons. One is that the symbol map r is not a function for reasons that are discussed in footnotes 10 and 11 of Sect. 4.6; another is that r is a function, but there is no signature morphism matching it. In Casl, the latter can happen for example if r does not preserve the overloading relations. An example is given in [38].

[6] The union may fail to be final in Casl if symbols newly enter the overloading relation, cf. Prop. 4.2.

[7] This property may fail if the actual parameter and the body share symbols that are in neither the formal parameter nor the import.

[8] This property may fail if the fitting morphism σ is not injective (say, it maps both *elem*1 and *elem*2 to *nat*) and this leads to new identifications in the extension (say, both *list*[*elem*1] and *list*[*elem*2] occur in the body, so $\sigma(\Delta)$ maps both to *list*[*nat*]), see Sect. 4.6.

$$\begin{tikzcd}
\Sigma \arrow[r, phantom, "\subseteq"] \arrow[d, "\sigma"] & \Sigma' \arrow[d, "\sigma(\Delta)"] \\
\Sigma_A \arrow[r, phantom, "\subseteq"] & \Sigma_A \cup \Sigma_A(\Delta)
\end{tikzcd}$$

is a pushout in **Sig**.

Proof. The diagram commutes because $||\sigma|| \subseteq r \subseteq ||\sigma(\Delta)||$. Given any cocone $(\sigma_A \colon \Sigma_A \to \widehat{\Sigma}, \sigma' \colon \Sigma' \to \widehat{\Sigma})$ define $\theta \colon |\Sigma_A| \cup |\Sigma_A(\Delta)| \to \widehat{\Sigma}$ as follows:

$$\theta(sy) = |\sigma_A|(sy), \text{ if } sy \in |\Sigma_A|$$

$$\theta(sy) = |\sigma'|(sy'), \text{ if } sy \in |\Sigma_A(\Delta)|, |\sigma(\Delta)|(sy') = sy$$

The second line is well-defined because $ker(|(r|^{\Sigma'})|) \subseteq ker(|\sigma|)$, and the two lines agree on their overlapping part because $|\Sigma_A| \cap |\Sigma_A(\Delta)| \subseteq |\sigma|(|\Sigma|)$ and the cocone commutes.

Clearly, θ is the unique function from $|\Sigma_A| \cup |\Sigma_A(\Delta)|$ to $|\widehat{\Sigma}|$ with $\theta \circ |\iota_{\Sigma_A \subseteq \Sigma_A \cup \Sigma_A(\Delta)}| = |\sigma_A|$ and $\theta \circ |\sigma(\Delta)| = |\sigma'|$.

Now $\theta \circ \iota_{\Sigma_A(\Delta) \subseteq \Sigma_A \cup \Sigma_A(\Delta)} \circ r|^{\Sigma'} = \sigma'$ is a signature morphism. By finality of $r|^{\Sigma'}$, $\theta \circ \iota_{\Sigma_A(\Delta) \subseteq \Sigma_A \cup \Sigma_A(\Delta)}$ is also a signature morphism. Since $\theta \circ \iota_{\Sigma_A \subseteq \Sigma_A \cup \Sigma_A(\Delta)} = \sigma_A$ is a signature morphism as well, by finality of the union, θ is also a signature morphism. □

4.1.4 Extended Signatures

Any symbol declared explicitly in the parameter (and not only in the import) must be mapped to a symbol declared explicitly in the argument specification (cf. Sect. 4.3.2 below).

This requirement eases the use of the default mechanism for symbol maps occurring in instantiations of generic specifications. The reason is that argument specifications are regarded as extensions of the imports. While the latter are always instantiated with an identity map, the fitting map for the former may be computed by the default mechanism. However, in the presence of imports, in most cases ambiguities will arise, since e.g. the imports as well as the actual parameter will declare sorts symbols. By concentrating on the symbols declared explicitly in the parameter and the argument specification, respectively, and excluding the symbols from the import, there are fewer potential ambiguities.

However, a prerequisite for realizing this is the ability to distinguish between symbols in the argument specification that only come from the import and those that are explicitly declared or re-declared in the argument specification. As mentioned on page 127, when abstracting signature fragments Δ to signature inclusions $\Sigma \hookrightarrow \Sigma \cup \Delta$, information about any re-declaration in Δ of symbols in Σ is lost. In order to regain this information, we now extend signatures with an additional signature symbol set, called the set of *explicitly declared signature symbols*.

Technically, we replace the institution with symbols above by a new one that can be obtained from the old one in a generic (i.e. institution-independent) way. References below to the derived notions from Sect. 4.1.2 above should be taken to refer to this new institution.

Category of signatures and signature morphisms: Signatures are pairs (Σ^{basic}, SSY), where Σ^{basic} is a signature from **SubSig** and $SSY \subseteq |\Sigma^{basic}|$ is a set of signature symbols. Signature morphisms $\sigma : (\Sigma_1^{basic}, SSY_1) \rightarrow (\Sigma_2^{basic}, SSY_2)$ are such that $\sigma\colon \Sigma_1^{basic} \rightarrow \Sigma_2^{basic}$ is a signature morphism from **SubSig** and $|\sigma|(SSY_1) \subseteq SSY_2$.

Sentences: The sentence functor is just **SubSen**, acting on the first component of signatures.

Models: The model functor is just **SubMod**, acting on the first component of signatures.

Satisfaction: Satisfaction is just as before.

Signature symbols: The signature symbol functor is just the signature symbol functor from before, acting on the first component of signatures.

Symbols: Symbols are defined as before.

Signature symbols matching symbols: The matching relation between signature symbols and symbols is defined as before.

Factorization of signature symbol functor: The factorization of the signature symbol functor is defined as before (the symbol function acting on the first component of signatures).

Names of signature symbols: Names of signature symbols are defined as before.

Empty signature: The empty signature is defined as before, except that it is paired with the empty set of signature symbols.

Signature unions: Signature unions are defined component-wise.

Generating signature morphisms: A signature morphism

$$\sigma\colon (\Sigma_1^{basic}, SSY_1) \rightarrow (\Sigma_2^{basic}, SSY_2)$$

is said to be generating if $\sigma\colon \Sigma_1^{basic} \rightarrow \Sigma_2^{basic}$ is generating in the earlier sense, and moreover $|\sigma|\colon SSY_1 \rightarrow SSY_2$ is surjective.

We additionally define the function *EmptyExplicit* on extended signatures, given by

$$\mathit{EmptyExplicit}(\Sigma^{basic}, SSY) = (\Sigma^{basic}, \emptyset).$$

4.1.5 Institution Independent Structuring Concepts

Abusing the notation somewhat, in the rest of the semantics of Casl we will work with the institution with fully qualified symbols defined in the preceding section. The category of extended signatures will be denoted simply by **Sig**;

the same notation will be used for the class of its objects, with Σ used as the main meta-variable ranging over it, and σ as the main meta-variable ranging over signature morphisms. Similarly, the sentence and model functors of this institutions will be denoted by **Sen** and **Mod**, respectively, with the rest of the components of the institution with symbols denoted as they have been so far.

The use of this generic notation is to remind the reader that everywhere except for a number of rules concerned with the details of symbol maps in Sect. 4.5, no specific assumptions are made about these concepts, except that they form an institution with fully qualified symbols (satisfying the properties listed in Sect. 4.1.1).

Specification Morphisms

> For a specification morphism, it is required that the reduct of each model of the target specification is a model of the source specification; otherwise the semantics is undefined.

Given specifications $\mathtt{SPEC}_1$ and $\mathtt{SPEC}_2$ with signatures Σ_1 and Σ_2 and model classes $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively, a *specification morphism* $\sigma\colon \mathtt{SPEC}_1 \rightarrow \mathtt{SPEC}_2$ is a signature morphism $\sigma\colon \Sigma_1 \rightarrow \Sigma_2$ such that $\mathcal{M}_2|_\sigma \subseteq \mathcal{M}_1$.

Generic Specifications

The static semantics GS_s of a generic specification is a *generic signature* consisting of two signatures (the import and the body) and a sequence of signatures (the formal parameters). (If there is more than one import specification, the imports can be united to a single import. This is not possible for the parameter specifications: they have to be instantiated individually.)

$$(\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle, \Sigma_B)$$
$$\text{or } GS_s \in \mathit{GenSig} = \mathbf{Sig} \times \mathit{FinSeq}(\mathbf{Sig}) \times \mathbf{Sig}$$

The requirements on a generic signature $GS_s = (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle, \Sigma_B) \in \mathit{GenSig}$ are:

- $\Sigma_I \subseteq \Sigma_i$ for $1 \leq i \leq n$, and
- $\Sigma_1 \cup \cdots \cup \Sigma_n \subseteq \Sigma_B$.

Let pairs (Σ_i^A, σ_i) be given where $\sigma_i\colon \Sigma_i \rightarrow \Sigma_i^A$ is a signature morphism, for $1 \leq i \leq n$. We assume that $\sigma_f = \sigma_1 \cup \cdots \cup \sigma_n \cup id_{\Sigma_I}$ is defined, which implies by Prop. 4.3 that all the σ_i are compatible with each other and compatible with the identity id_{Σ_I}. Let Δ denote the signature extension given by the inclusion of $\Sigma_1 \cup \cdots \cup \Sigma_n$ into Σ_B. If $\Sigma_A = \Sigma_1^A \cup \ldots \cup \Sigma_n^A$ is defined then:

$$GS_s((\Sigma_1^A, \sigma_1), \ldots, (\Sigma_n^A, \sigma_n)) = (\Sigma_A \cup \Sigma_A(\Delta), \sigma_f(\Delta)).$$

The notations $\Sigma_A(\Delta)$ and $\sigma_f(\Delta)$ are defined on page 199.

The model semantics GS_m of a generic specification consists of a model class for the import, a sequence of model classes for the formal parameters, and a model class for the body of the generic specification.

$$(\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_{n'} \rangle, \mathcal{M}_B) \quad \text{or } GS_m \in \mathbf{GenSpec} = \mathbf{ModelClass} \times \mathit{FinSeq}(\mathbf{ModelClass}) \times \mathbf{ModelClass}$$

A generic signature $GS_s = (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle, \Sigma_B) \in \mathit{GenSig}$ and an element $GS_m = (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_{n'} \rangle, \mathcal{M}_B)$ of **GenSpec** are *compatible* if

- $n = n'$,
- $\mathcal{M}_I$ is a class of Σ_I-models,
- each $\mathcal{M}_i$ is a class of Σ_i-models and each element of $\mathcal{M}_i$ extends some model of $\mathcal{M}_I$ for $1 \leq i \leq n$, and
- $\mathcal{M}_B$ is a class of Σ_B-models, each model of which extends some model of $\mathcal{M}_i$ for each $1 \leq i \leq n$.

Let pairs $(\mathcal{M}_i^A, \sigma_i)$ be given where $\mathcal{M}_i^A$ is a class of models over Σ_i^A and σ_i is a signature morphism from Σ_i to Σ_i^A, for $1 \leq i \leq n$. Provided that

$$(\Sigma, \sigma'_f) = GS_s((\Sigma_1^A, \sigma_1), \ldots, (\Sigma_n^A, \sigma_n))$$

is defined and $\mathcal{M}_i^A|_{\sigma_i} \subseteq \mathcal{M}_i$ for all $1 \leq i \leq n$ then

$$GS_m((\mathcal{M}_1^A, \sigma_1), \ldots, (\mathcal{M}_n^A, \sigma_n)) = \mathcal{M}$$

where $\mathcal{M}$ is the class of models over Σ given by

$$\mathcal{M} = \{ M \in \mathbf{Mod}(\Sigma) \mid M|_{\Sigma_i^A} \in \mathcal{M}_i^A,\ 1 \leq i \leq n,\ M|_{\sigma'_f} \in \mathcal{M}_B \}.$$

Views

The static semantics of a view consists of a signature (the source signature), a signature morphism, and a generic signature (the target signature of the view).

$$(\Sigma_s, \sigma, GS_s) \quad \text{or } V_s \in \mathit{ViewSig} = \mathbf{Sig} \times \mathit{SignatureMorphism} \times \mathit{GenSig}$$

For an element (Σ_s, σ, GS_s) in *ViewSig* we require that σ is a signature morphism from Σ_s to Σ_B, where $GS_s = (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle, \Sigma_B)$.

The model semantics of a view consists of a model class and the model semantics of a generic specification.

$$(\mathcal{M}_s, GS_m) \quad \text{or } V_m \in \mathbf{ViewSpec} = \mathbf{ModelClass} \times \mathbf{GenSpec}$$

An element $V_s = (\Sigma_s, \sigma, GS_s)$ in *ViewSig* and an element $V_m = (\mathcal{M}_s, GS_m)$ in **ViewSpec** are *compatible* if

- $\mathcal{M}_s$ is a class of Σ_s-models,
- $GS_s = (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle, \Sigma_B)$ and $GS_m = (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle, \mathcal{M}_B)$ are compatible, and
- $\mathcal{M}_B|_\sigma \subseteq \mathcal{M}_s$.

Local and Global Environment

Within a structured specification, the *current signature*, also called the *local environment*, may vary. The current association between names and the specifications that they reference is called the *global environment.*

As introduced elsewhere (cf. Sect. 6.1), model (resp., static) *global environments* Γ_m (resp., Γ_s) contain a generic specification component $\mathcal{G}_m : \mathit{SpecName} \overset{\text{fin}}{\to} \mathbf{GenSpec}$ (resp. $\mathcal{G}_s : \mathit{SpecName} \overset{\text{fin}}{\to} \mathit{GenSig}$), as well as a view component $\mathcal{V}_m : \mathit{ViewName} \overset{\text{fin}}{\to} \mathbf{ViewSpec}$ (resp. $\mathcal{V}_s : \mathit{ViewName} \overset{\text{fin}}{\to} \mathit{ViewSig}$).

A static global environment and a model global environment are *compatible* if their components are compatible, see Sect. 6.1. Compatibility for the generic specification and view components has been defined above.

The rest of this chapter indicates the semantics of the constructs of structured specifications.

4.2 Structured Specifications

The static semantics of a specification has been given as a signature extension Δ of the local environment Σ in Chap. 2, where signature extension referred to a signature fragment. At the institution independent level, where we do not have signature fragments, this is abstracted to the signature inclusion $\Sigma \hookrightarrow \Sigma \cup \Delta$. As mentioned on page 127, information about any re-declaration in Δ of symbols in Σ is lost by this abstraction. Therefore, in Sect. 4.1.4 we provide a mechanism that keeps the symbols of Δ together with $\Sigma \cup \Delta$.

In structured specifications, a specification `SPEC` may occur in a context where it is to *extend* other specifications, providing itself only part of a signature. Hence, its interpretation determines an extended signature Σ', given a signature Σ (the local environment), together with a model class over Σ' (when defined), given a model class over Σ.
Translations and reductions in a `SPEC` are not allowed to affect symbols that are already in the local environment that is being extended.

```
SPEC ::= BASIC-SPEC | TRANSLATION | REDUCTION
       | UNION | EXTENSION | FREE-SPEC | LOCAL-SPEC
       | CLOSED-SPEC
```

Casl-specific rules for basic specifications

$$\boxed{\Sigma, \Gamma_s \vdash \texttt{SPEC} \rhd \Sigma' \qquad\qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}'}$$

Γ_s and Γ_m are compatible global environments and $\mathcal{M}$ is a class of models over Σ; then Σ' is an extension of Σ, and $\mathcal{M}'$ is a class of models over Σ' with each model extending some model in $\mathcal{M}$. Recall that signatures Σ are pairs (Σ^{basic}, SSY) with $SSY \subseteq |\Sigma^{basic}|$.

$$\frac{\Sigma^{basic} \vdash \texttt{BASIC-SPEC} \rhd (\Delta, \Psi)}{(\Sigma^{basic}, SSY), \Gamma_s \vdash \texttt{BASIC-SPEC}\,\text{qua}\,\texttt{SPEC} \rhd (\Sigma^{basic} \cup \Delta, SSY \cup |\Delta|)}$$

$$\frac{\Sigma, \mathcal{M} \vdash \texttt{BASIC-SPEC} \Rightarrow \mathcal{M}'}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{BASIC-SPEC}\,\text{qua}\,\texttt{SPEC} \Rightarrow \mathcal{M}'}$$

end of Casl-specific rules

Other rules elided (see Sect. 1.3).

4.2.1 Translations

The symbols mapped by SYMB-MAP-ITEMS+ must be among those declared by SPEC. The signature Σ given by SPEC and the mapping SYMB-MAP-ITEMS+ then determine a signature morphism to a signature Σ', as explained in Sect. 4.1, which must not affect the symbols already declared in the local environment. The class of models of the translation consists exactly of those models over Σ' whose reducts along the morphism are models of SPEC.

```
TRANSLATION ::= translation SPEC RENAMING
RENAMING    ::= renaming  SYMB-MAP-ITEMS+
```

$$\boxed{\Sigma \vdash \texttt{RENAMING} \rhd \sigma\colon \Sigma \rightarrow \Sigma'}$$

Σ is a signature; then $\sigma\colon \Sigma \rightarrow \Sigma'$ is a final signature morphism.

$$\frac{\vdash \texttt{SYMB-MAP-ITEMS+} \rhd r}{\Sigma \vdash \texttt{renaming}\,\texttt{SYMB-MAP-ITEMS+} \rhd r|^{\Sigma}}$$

$$\boxed{\Sigma, \Gamma_s \vdash \texttt{TRANSLATION} \rhd \Sigma' \qquad\qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{TRANSLATION} \Rightarrow \mathcal{M}'}$$

Γ_s and Γ_m are compatible global environments, and $\mathcal{M}$ is a class of models over Σ; then Σ' is an extension of Σ and $\mathcal{M}'$ is a class of models over Σ' with each model extending some model in $\mathcal{M}$.

$$\frac{\begin{array}{c} \Sigma, \Gamma_s \vdash \mathtt{SPEC} \rhd \Sigma' \\ \Sigma' \vdash \mathtt{RENAMING} \rhd \sigma\colon \Sigma' \to \Sigma'' \\ |\sigma| \text{ is the identity on signature symbols in } |\Sigma| \end{array}}{\Sigma, \Gamma_s \vdash \mathtt{translation}\ \mathtt{SPEC}\ \mathtt{RENAMING} \rhd \Sigma''}$$

Note that $\Sigma \subseteq \Sigma''$ because $|\sigma|$ is the identity on signature symbols in $|\Sigma|$.

$$\frac{\begin{array}{c} \Sigma, \Gamma_s \vdash \mathtt{SPEC} \rhd \Sigma' \\ \Sigma' \vdash \mathtt{RENAMING} \rhd \sigma\colon \Sigma' \to \Sigma'' \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC} \Rightarrow \mathcal{M}' \end{array}}{\begin{array}{r} \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{translation}\ \mathtt{SPEC}\ \mathtt{RENAMING} \Rightarrow \\ \{M \in \mathbf{Mod}(\Sigma'') \mid M|_\sigma \in \mathcal{M}'\} \end{array}}$$

4.2.2 Reductions

In the case of a hiding reduction, the signature Σ given by SPEC and the set of symbols listed by SYMB-ITEMS+ determine the inclusion of the largest subsignature Σ' of Σ that does *not* contain any of the listed symbols, as explained in Sect. 4.1.
In the case of a revealing reduction, the signature Σ given by SPEC and the set of symbols mapped by SYMB-MAP-ITEMS+ determine the inclusion of the smallest subsignature Σ' of Σ that contains all of the listed symbols, as explained in Sect. 4.1. This signature then may be further translated.
A reduction must not affect the symbols already declared in the local environment.

```
REDUCTION   ::= reduction SPEC RESTRICTION
RESTRICTION ::= HIDDEN | REVEALED
HIDDEN      ::= hidden SYMB-ITEMS+
REVEALED    ::= revealed SYMB-MAP-ITEMS+
```

$$\boxed{(\Sigma, \Sigma') \vdash \mathtt{RESTRICTION} \rhd \sigma\colon \Sigma_1 \to \Sigma''}$$

Σ' is an extension of Σ; then $\Sigma \subseteq \Sigma_1 \subseteq \Sigma'$, and $|\sigma|$ is the identity on signature symbols in $|\Sigma|$.

$$\frac{\begin{array}{c} |\Sigma'| \vdash \mathtt{SYMB\text{-}ITEMS+} \rhd SSYs \\ SSYs \cap |\Sigma| = \emptyset \qquad \Sigma_1 = \Sigma'|^{SSYs} \end{array}}{(\Sigma, \Sigma') \vdash \mathtt{hidden}\ \mathtt{SYMB\text{-}ITEMS+} \rhd id_{\Sigma_1}}$$

The second premise ensures that no symbols from Σ are hidden.

$$\frac{\begin{array}{c} \vdash \mathtt{SYMB\text{-}MAP\text{-}ITEMS+} \rhd r \\ SSYs = \{SY \in |\Sigma'| \mid SSY \text{ matches some } SY \in dom(r)\} \\ \Sigma_1 = \Sigma'|_{SSYs \cup |\Sigma|} \qquad \sigma\colon \Sigma_1 \to \Sigma'' = r|^{\Sigma_1} \\ |\sigma| \text{ is the identity on signature symbols in } |\Sigma| \end{array}}{(\Sigma, \Sigma') \vdash \mathtt{revealed}\ \mathtt{SYMB\text{-}MAP\text{-}ITEMS+} \rhd \sigma\colon \Sigma_1 \to \Sigma''}$$

The third premise generates all symbols from the symbol map and also the symbols from the local environment into the revealed signature. The fifth premise ensures that the local environment is not renamed. Note that $\Sigma \subseteq \Sigma_1$.

$$\boxed{\Sigma, \Gamma_s \vdash \mathtt{REDUCTION} \rhd \Sigma' \qquad\qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{REDUCTION} \Rightarrow \mathcal{M}'}$$

Γ_s and Γ_m are compatible global environments and $\mathcal{M}$ is a class of models over Σ; then Σ' is an extension of Σ, and $\mathcal{M}'$ is a class of models over Σ' with each model extending some model in $\mathcal{M}$.

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \mathtt{SPEC} \rhd \Sigma' \\ (\Sigma, \Sigma') \vdash \mathtt{RESTRICTION} \rhd \sigma\colon \Sigma_1 \to \Sigma''\end{array}}{\Sigma, \Gamma_s \vdash \mathtt{reduction}\ \mathtt{SPEC}\ \mathtt{RESTRICTION} \rhd \Sigma''}$$

Note that $\Sigma \subseteq \Sigma''$ because $\Sigma \subseteq \Sigma_1$ and $|\sigma|$ is the identity on signature symbols in $|\Sigma|$.

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \mathtt{SPEC} \rhd \Sigma' \\ (\Sigma, \Sigma') \vdash \mathtt{RESTRICTION} \rhd \sigma\colon \Sigma_1 \to \Sigma'' \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC} \Rightarrow \mathcal{M}'\end{array}}{\begin{array}{l}\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{reduction}\ \mathtt{SPEC}\ \mathtt{RESTRICTION} \Rightarrow \\ \qquad \{M'' \in \mathbf{Mod}(\Sigma'') \mid \text{ there is some } M' \in \mathcal{M}' \text{ with } M'|_{\Sigma_1} = M''|_\sigma\}\end{array}}$$

4.2.3 Unions

The signature of the union is the union of the signatures of each `SPEC`. Thus all occurrences of a symbol in the `SPEC`s are interpreted uniformly. The models are those models of the union signature for which the reduct to the signature of the i^{th} `SPEC` is a model of that `SPEC`.

```
UNION ::= union SPEC+
```

$$\boxed{\Sigma, \Gamma_s \vdash \mathtt{UNION} \rhd \Sigma' \qquad\qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{UNION} \Rightarrow \mathcal{M}'}$$

Γ_s and Γ_m are compatible global environments and $\mathcal{M}$ is a class of models over Σ; then Σ' is an extension of Σ, and $\mathcal{M}'$ is a class of models over Σ' with each model extending some model in $\mathcal{M}$.

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma_1 \\ \ldots \\ \Sigma, \Gamma_s \vdash \mathtt{SPEC}_n \rhd \Sigma_n\end{array}}{\Sigma, \Gamma_s \vdash \mathtt{union}\ \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \rhd \Sigma_1 \cup \ldots \cup \Sigma_n}$$

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma_1 \\ \ldots \\ \Sigma, \Gamma_s \vdash \mathtt{SPEC}_n \rhd \Sigma_n \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_1 \Rightarrow \mathcal{M}_1 \\ \ldots \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_n \Rightarrow \mathcal{M}_n\end{array}}{\begin{array}{l}\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{union}\,\mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \Rightarrow \\ \qquad \{M \in \mathbf{Mod}(\Sigma_1 \cup \ldots \cup \Sigma_n) \mid M|_{\Sigma_i} \in \mathcal{M}_i,\ 1 \leq i \leq n\}\end{array}}$$

4.2.4 Extensions

$\mathtt{SPEC}_1$ determines an extension from the local environment to a signature Σ_1. For $i = 2, \ldots, n$ each $\mathtt{SPEC}_i$ determines an extension from Σ_{i-1} to a signature Σ_i. The signature determined by the entire extension is then Σ_n. Models are determined similarly, with each $\mathtt{SPEC}_i$ determining a class $\mathcal{M}_i$ of Σ_i-models whose Σ_{i-1}-reducts are in $\mathcal{M}_{i-1}$.

```
EXTENSION ::= extension SPEC+
```

$$\boxed{\Sigma, \Gamma_s \vdash \mathtt{EXTENSION} \rhd \Sigma' \qquad\qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{EXTENSION} \Rightarrow \mathcal{M}'}$$

Γ_s and Γ_m are compatible global environments and $\mathcal{M}$ is a class of models over Σ; then Σ' is an extension of Σ, and $\mathcal{M}'$ is a class of models over Σ' with each model extending some model in $\mathcal{M}$.

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma_1 \\ \ldots \\ \Sigma_{n-1}, \Gamma_s \vdash \mathtt{SPEC}_n \rhd \Sigma_n\end{array}}{\Sigma, \Gamma_s \vdash \mathtt{extension}\,\mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \rhd \Sigma_n}$$

$$\frac{\begin{array}{c}\Sigma, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma_1 \\ \ldots \\ \Sigma_{n-1}, \Gamma_s \vdash \mathtt{SPEC}_n \rhd \Sigma_n \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_1 \Rightarrow \mathcal{M}_1 \\ \ldots \\ \Sigma_{n-1}, \mathcal{M}_{n-1}, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_n \Rightarrow \mathcal{M}_n\end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{extension}\,\mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \Rightarrow \mathcal{M}_n}$$

A semantic annotation can occur at any point in the list of extensions and then concerns only the extension at that point. If the annotation is attached to the $i - 1^{th}$ extension (i.e. between $\mathtt{SPEC}_{i-1}$ and $\mathtt{SPEC}_i$, where $2 \leq i \leq n$)), then it holds under the following conditions.

- %**cons** (*conservative*): each model $M \in \mathcal{M}_{i-1}$ has an $\mathcal{M}_i$*-extension*, i.e. a model $M' \in \mathcal{M}_i$ such that $M'|_{\Sigma_{i-1}} = M$.
- %**mono** (*monomorphic*): each model $M \in \mathcal{M}_{i-1}$ has a unique $\mathcal{M}_i$-extension up to isomorphism, i.e. any two $\mathcal{M}_i$-extensions are isomorphic.
- %**def** (*definitional*): each model in $\mathcal{M}_{i-1}$ has a unique $\mathcal{M}_i$-extension
- %**implies** (*implicational*): $\Sigma_i = \Sigma_{i-1}$ and $\mathcal{M}_i = \mathcal{M}_{i-1}$.

4.2.5 Free Specifications

`SPEC` determines an extension from the local environment to a signature Σ'. `free-spec SPEC` determines the class of Σ'-models that are free extensions for `SPEC` of their own reducts to models of the current local environment.

```
FREE-SPEC ::= free-spec SPEC
```

$\Sigma, \Gamma_s \vdash \texttt{FREE-SPEC} \rhd \Sigma' \qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{FREE-SPEC} \Rightarrow \mathcal{M}'$

Γ_s and Γ_m are compatible global environments and $\mathcal{M}$ is a class of models over Σ; then Σ' is an extension of Σ, and $\mathcal{M}'$ is a class of models over Σ' with each model extending some model in $\mathcal{M}$.

$$\frac{\Sigma, \Gamma_s \vdash \texttt{SPEC} \rhd \Sigma'}{\Sigma, \Gamma_s \vdash \texttt{free-spec SPEC} \rhd \Sigma'}$$

$$\frac{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}_1}{\begin{array}{l}\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{free-spec SPEC} \Rightarrow \\ \qquad \{M \in \mathcal{M}_1 \mid M \text{ is a free extension of } M|_\Sigma \text{ w.r.t. } \mathcal{M}_1\}\end{array}}$$

A model M is a free extension of $M|_\Sigma$ w.r.t. a class of models $\mathcal{M}_1$ if for all elements M_1 of $\mathcal{M}_1$ and homomorphisms $h : M|_\Sigma \to M_1|_\Sigma$ there exists a unique homomorphism $h^\# : M \to M_1$ with $h^\#|_\Sigma = h$.

If Σ is the empty local environment $\emptyset$ then $M|_\emptyset$ is the only element of **Mod**$(\emptyset)$ and for each M_1 in $\mathcal{M}_1$ the identity *id* on $M|_\Sigma$ is a homomorphism from $M|_\Sigma$ to $M_1|_\Sigma$. If M is a free extension of $M|_\emptyset$ w.r.t. $\mathcal{M}_1$ then *id* extends to a unique homomorphism $id^\#$ from M to M_1, which makes M an initial object of $\mathcal{M}_1$.

In the Casl institution, under some minor restrictions the basic specification written:

free types DD_1; … DD_n;

has the same interpretation as the free structured specification written:

free { types DD_1; … DD_n; **}**

See Theorem 3.11 in Sect. 3.2.2 for an equivalence between free types as Casl basic specifications and structured free datatypes, and for details of what the minor restrictions are.

4.2.6 Local Specifications

Declaring $\mathtt{SPEC}_1$ as local to $\mathtt{SPEC}_2$ is equivalent to an extension of $\mathtt{SPEC}_1$ by $\mathtt{SPEC}_2$, followed by a hiding of all the symbols declared by $\mathtt{SPEC}_1$ that are not already in the current local environment.

```
LOCAL-SPEC ::= local-spec SPEC SPEC
```

$$\boxed{\Sigma, \Gamma_s \vdash \mathtt{LOCAL\text{-}SPEC} \rhd \Sigma' \qquad\qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{LOCAL\text{-}SPEC} \Rightarrow \mathcal{M}'}$$

Γ_s and Γ_m are compatible global environments and $\mathcal{M}$ is a class of models over Σ; then Σ' is an extension of Σ, and $\mathcal{M}'$ is a class of models over Σ' with each model extending some model in $\mathcal{M}$.

$$\frac{\begin{array}{cc} \Sigma, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma' & \Sigma', \Gamma_s \vdash \mathtt{SPEC}_2 \rhd \Sigma'' \\ \Sigma_1 = \Sigma''|^{|\Sigma'| \setminus |\Sigma|} & |\Sigma''| \setminus |\Sigma'| \subseteq |\Sigma_1| \end{array}}{\Sigma, \Gamma_s \vdash \mathtt{local\text{-}spec}\ \mathtt{SPEC}_1\ \mathtt{SPEC}_2 \rhd \Sigma_1}$$

The last premise ensures that symbols newly introduced in $\mathtt{SPEC}_2$ are not hidden.

$$\frac{\begin{array}{c} \Sigma, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma' \qquad \Sigma', \Gamma_s \vdash \mathtt{SPEC}_2 \rhd \Sigma'' \\ \Sigma_1 = \Sigma''|^{|\Sigma'| \setminus |\Sigma|} \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_1 \Rightarrow \mathcal{M}' \qquad \Sigma', \mathcal{M}', \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_2 \Rightarrow \mathcal{M}'' \end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{local\text{-}spec}\ \mathtt{SPEC}_1\ \mathtt{SPEC}_2 \Rightarrow \{M|_{\Sigma_1} \mid M \in \mathcal{M}''\}}$$

4.2.7 Closed Specifications

A closed specification determines the same signature and class of models as SPEC determines in the empty local environment.

```
CLOSED-SPEC ::= closed-spec SPEC
```

$$\boxed{\Sigma, \Gamma_s \vdash \mathtt{CLOSED\text{-}SPEC} \rhd \Sigma' \qquad\qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{CLOSED\text{-}SPEC} \Rightarrow \mathcal{M}'}$$

$\mathit{\Gamma}_s$ and $\mathit{\Gamma}_m$ are compatible global environments and $\mathcal{M}$ is a class of models over $\mathit{\Sigma}$; then $\mathit{\Sigma}'$ is an extension of $\mathit{\Sigma}$, and $\mathcal{M}'$ is a class of models over $\mathit{\Sigma}'$ with each model extending some model in $\mathcal{M}$.

$$\frac{\emptyset, \mathit{\Gamma}_s \vdash \texttt{SPEC} \rhd \mathit{\Sigma}'}{\mathit{\Sigma}, \mathit{\Gamma}_s \vdash \texttt{closed-spec}\,\texttt{SPEC} \rhd \mathit{\Sigma} \cup \mathit{\Sigma}'}$$

$$\frac{\begin{array}{c}\emptyset, \mathit{\Gamma}_s \vdash \texttt{SPEC} \rhd \mathit{\Sigma}' \\ \emptyset, \mathcal{M}_\perp, \mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}' \\ \mathcal{M}'' = \{M \in \mathbf{Mod}(\mathit{\Sigma} \cup \mathit{\Sigma}') \mid M|_{\mathit{\Sigma}} \in \mathcal{M},\ M|_{\mathit{\Sigma}'} \in \mathcal{M}'\}\end{array}}{\mathit{\Sigma}, \mathcal{M}, \mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{closed-spec}\,\texttt{SPEC} \Rightarrow \mathcal{M}''}$$

The union $\mathit{\Sigma} \cup \mathit{\Sigma}'$ and the construction of $\mathcal{M}''$ ensure the invariant that the resulting signature extends the local environment signature $\mathit{\Sigma}$, and that each resulting model extends one from $\mathcal{M}$.

4.3 Named and Generic Specifications

4.3.1 Specification Definitions

A generic specification definition defines the name *SN* to refer to the specification that has parameter and import specifications as indicated by `GENERICITY`, and body specification `SPEC`. This extends the global environment (which must not already include a definition for *SN*).
The declared parameters show just which parts of the generic specification are *intended* to vary between different references to it. The imports, in contrast, are fixed, and common to the parameters, body, and arguments.

```
SPEC-DEFN   ::= spec-defn SPEC-NAME GENERICITY SPEC
GENERICITY  ::= genericity PARAMS IMPORTED
PARAMS      ::= params SPEC*
IMPORTED    ::= imported SPEC*
SPEC-NAME   ::= SIMPLE-ID
```

Though possible in the abstract syntax, the concrete syntax does not allow an import to be specified for non-generic specifications. Thus, if the import is not empty for a non-generic specifications, the static semantics will be undefined. This is captured in the rule for `GENERICITY` below. As a consequence, the static semantics of a non-generic specification is always of the form $(\emptyset, \langle\rangle, \mathit{\Sigma}_B)$ and the model semantics is of the form $(\mathcal{M}_\perp, \langle\rangle, \mathcal{M}_B)$ where $\mathcal{M}_B$ is a class of $\mathit{\Sigma}_B$-models.

$$\boxed{\mathit{\Gamma}_s \vdash \texttt{SPEC-DEFN} \rhd \mathit{\Gamma}_s' \qquad\qquad \mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{SPEC-DEFN} \Rightarrow \mathit{\Gamma}_m'}$$

Γ_s and Γ_m are compatible global environments; then Γ'_s and Γ'_m are compatible environments extending Γ_s and Γ_m, respectively.

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\\ \mathit{SN} \notin \mathit{Dom}(\mathcal{G}_s) \cup \mathit{Dom}(\mathcal{V}_s) \cup \mathit{Dom}(\mathcal{A}_s) \cup \mathit{Dom}(\mathcal{T}_s)\\ \Gamma_s \vdash \mathtt{GENERICITY} \rhd (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle)\\ \Sigma_1 \cup \cdots \cup \Sigma_n, \Gamma_s \vdash \mathtt{SPEC} \rhd \Sigma_B\end{array}}{\begin{array}{l}\Gamma_s \vdash \mathtt{spec\text{-}defn}\ \mathit{SN}\ \mathtt{GENERICITY}\ \mathtt{SPEC} \rhd\\ \qquad (\mathcal{G}_s \cup \{\mathit{SN} \mapsto (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle, \Sigma_B)\}, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\end{array}}$$

$$\frac{\begin{array}{c}\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)\\ \Gamma_s \vdash \mathtt{GENERICITY} \rhd (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle)\\ \Gamma_s, \Gamma_m \vdash \mathtt{GENERICITY} \Rightarrow (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle)\\ \mathcal{M}_P = \{M \in \mathbf{Mod}(\Sigma_1 \cup \ldots \cup \Sigma_n) \mid M|_{\Sigma_I} \in \mathcal{M}_I,\ M|_{\Sigma_i} \in \mathcal{M}_i,\ 1 \le i \le n\}\\ \Sigma_1 \cup \cdots \cup \Sigma_n, \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC} \Rightarrow \mathcal{M}_B\end{array}}{\begin{array}{l}\Gamma_s, \Gamma_m \vdash \mathtt{spec\text{-}defn}\ \mathit{SN}\ \mathtt{GENERICITY}\ \mathtt{SPEC} \Rightarrow\\ \qquad\qquad (\mathcal{G}_m \cup \{\mathit{SN} \mapsto (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle, \mathcal{M}_B)\}, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)\end{array}}$$

$$\boxed{\begin{array}{l}\Gamma_s \vdash \mathtt{GENERICITY} \rhd (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle)\\ \qquad\qquad\qquad\qquad \Gamma_s, \Gamma_m \vdash \mathtt{GENERICITY} \Rightarrow (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle)\end{array}}$$

Γ_s and Γ_m are compatible global environments; then $\mathcal{M}_I$ is a class of models over Σ_I, Σ_i is an extension of Σ_I and $\mathcal{M}_i$ is a class of models over Σ_i with each model extending some model in $\mathcal{M}_I$ for $1 \le i \le n$.

$$\frac{\begin{array}{c}\Gamma_s \vdash \mathtt{IMPORTS} \rhd \Sigma_I\\ \Sigma_I, \Gamma_s \vdash \mathtt{PARAMS} \rhd \langle \Sigma_1, \ldots, \Sigma_n \rangle\\ n \ge 1\end{array}}{\Gamma_s \vdash \mathtt{genericity}\ \mathtt{PARAMS}\ \mathtt{IMPORTS} \rhd (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle)}$$

$$\frac{\begin{array}{c}\Gamma_s \vdash \mathtt{IMPORTS} \rhd \Sigma_I\\ \Gamma_s, \Gamma_m \vdash \mathtt{IMPORTS} \Rightarrow \mathcal{M}_I\\ \Sigma_I, \mathcal{M}_I, \Gamma_s, \Gamma_m \vdash \mathtt{PARAMS} \Rightarrow \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle\end{array}}{\Gamma_s, \Gamma_m \vdash \mathtt{genericity}\ \mathtt{PARAMS}\ \mathtt{IMPORTS} \Rightarrow (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle)}$$

In case there are no parameters, the import should be empty. If not, the semantics of `GENERICITY` is undefined:

$$\frac{\begin{array}{c}\Gamma_s \vdash \mathtt{IMPORTS} \rhd \emptyset\\ \emptyset, \Gamma_s \vdash \mathtt{PARAMS} \rhd \langle\rangle\end{array}}{\Gamma_s \vdash \mathtt{genericity}\ \mathtt{PARAMS}\ \mathtt{IMPORTS} \rhd (\emptyset, \langle\rangle)}$$

$$\frac{\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \mathtt{PARAMS} \Rightarrow \langle\rangle}{\Gamma_s, \Gamma_m \vdash \mathtt{genericity}\ \mathtt{PARAMS}\ \mathtt{IMPORTS} \Rightarrow (\mathcal{M}_\perp, \langle\rangle)}$$

$$\boxed{\begin{array}{ll} \Sigma, \Gamma_s \vdash \mathtt{PARAMS} \rhd \langle \Sigma_1, \ldots, \Sigma_n \rangle & \\ & \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{PARAMS} \Rightarrow \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle \end{array}}$$

Γ_s and Γ_m are compatible global environments and $\mathcal{M}$ is a class of models over Σ; then Σ_i is an extension of Σ and $\mathcal{M}_i$ is a class of models over Σ_i with each model extending some model in $\mathcal{M}$ for $1 \leq i \leq n$.

$$\frac{\begin{array}{c} \Sigma, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma_1 \\ \ldots \\ \Sigma, \Gamma_s \vdash \mathtt{SPEC}_n \rhd \Sigma_n \end{array}}{\Sigma, \Gamma_s \vdash \mathtt{params}\ \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \rhd \langle \Sigma_1, \ldots, \Sigma_n \rangle}$$

$$\frac{\begin{array}{c} \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_1 \Rightarrow \mathcal{M}_1 \\ \ldots \\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_n \Rightarrow \mathcal{M}_n \end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{params}\ \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \Rightarrow \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle}$$

$$\boxed{\Gamma_s \vdash \mathtt{IMPORTS} \rhd \Sigma \qquad\qquad \Gamma_s, \Gamma_m \vdash \mathtt{IMPORTS} \Rightarrow \mathcal{M}}$$

Γ_s and Γ_m are compatible global environments; then $\mathcal{M}$ is a class of models over Σ.

If the list of imported specifications is empty then the semantics of the import construct is the empty signature $\emptyset$ with $\mathcal{M}_\perp$ as its class of models.

$$\frac{}{\Gamma_s \vdash \mathtt{imports} \rhd \emptyset}$$

$$\frac{}{\Gamma_m \vdash \mathtt{imports} \Rightarrow \mathcal{M}_\perp}$$

For a non-empty list of imported specifications, the semantics of

$$\mathtt{imports}\ \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n$$

is the same as $\mathtt{union}\ \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n$ with respect to the empty local environment.

$$\frac{\emptyset, \Gamma_s \vdash \mathtt{union}\ \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \rhd \Sigma}{\Gamma_s \vdash \mathtt{imports}\ \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \rhd \mathit{EmptyExplicit}(\Sigma)}$$

For the definition of *EmptyExplicit*, see Sect. 4.1.4.

$$\frac{\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \mathtt{union}\ \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \Rightarrow \mathcal{M}}{\Gamma_m \vdash \mathtt{imports}\ \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \Rightarrow \mathcal{M}}$$

4.3.2 Specification Instantiation

An instantiation `SPEC-INST` of a generic specification with some fitting argument refers to the specification named *SN* in the global environment, providing a fitting argument $\texttt{FIT-ARG}_i$ for each declared parameter (in the same order).
When there is more than one parameter, the separate fitting argument morphisms have to be *compatible*, and their union has to yield a single morphism from the union of the parameters to the union of the arguments.
Each model of a fitting argument, when reduced by the signature morphism for that fitting argument, is required to be a model of the corresponding parameter specification, otherwise the instantiation is undefined.

```
SPEC      ::= ... | SPEC-INST
SPEC-INST ::= spec-inst SPEC-NAME FIT-ARG*
FIT-ARG  ::= FIT-SPEC
FIT-SPEC ::= fit-spec SPEC SYMB-MAP-ITEMS*
```

$$\boxed{\Sigma, \Gamma_s \vdash \texttt{SPEC-INST} \rhd \Sigma' \qquad\qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SPEC-INST} \Rightarrow \mathcal{M}'}$$

Γ_s and Γ_m are compatible global environments and $\mathcal{M}$ is a class of models over Σ; then Σ' is an extension of Σ and $\mathcal{M}'$ is a class of models over Σ' with each model extending some model in $\mathcal{M}$.

First, we study the case where the specification name refers to a non-generic specification with static semantics $(\emptyset, \langle\rangle, \Sigma_B)$ and model semantics $(\mathcal{M}_\perp, \langle\rangle, \mathcal{M}_B)$.

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\\ \mathcal{G}_s(SN) = (\emptyset, \langle\rangle, \Sigma_B))\end{array}}{\Sigma, \Gamma_s \vdash \texttt{spec-inst}\ SN \rhd \Sigma \cup \Sigma_B}$$

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\\ \mathcal{G}_s(SN) = (\emptyset, \langle\rangle, \Sigma_B)\\ \Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)\\ \mathcal{G}_m(SN) = (\mathcal{M}_\perp, \langle\rangle, \mathcal{M}_B))\end{array}}{\begin{array}{l}\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{spec-inst}\ SN \Rightarrow\\ \qquad \{M \in \mathbf{Mod}(\Sigma \cup \Sigma_B) \mid M|_\Sigma \in \mathcal{M},\ M|_{\Sigma_B} \in \mathcal{M}_B\}\end{array}}$$

Now the generic case, i.e., the a generic specification with static semantics $(\Sigma_I, \langle\Sigma_1, \ldots, \Sigma_n\rangle, \Sigma_B)$ and model semantics $(\mathcal{M}_I, \langle\mathcal{M}_1, \ldots, \mathcal{M}_n\rangle, \mathcal{M}_B)$, where $n \geq 1$.

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\\ \mathcal{G}_s(SN) = GS_s = (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n\rangle, \Sigma_B), \quad n \geq 1\\ \Sigma_I, \Sigma_1, \Gamma_s \vdash \mathtt{FIT\text{-}ARG}_1 \rhd \sigma_1, \Sigma_1^A\\ \ldots\\ \Sigma_I, \Sigma_n, \Gamma_s \vdash \mathtt{FIT\text{-}ARG}_n \rhd \sigma_n, \Sigma_n^A\\ (\Sigma', \sigma_f) = GS_s((\Sigma_1^A, \sigma_1), \ldots, (\Sigma_n^A, \sigma_n)) \text{ is defined}^9\end{array}}{\Sigma, \Gamma_s \vdash \mathtt{spec\text{-}inst}\ SN\ \mathtt{FIT\text{-}ARG}_1 \ldots \mathtt{FIT\text{-}ARG}_n \rhd \Sigma \cup \Sigma'}$$

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\\ \mathcal{G}_s(SN) = GS_s = (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n\rangle, \Sigma_B), \quad n \geq 1\\ \Sigma_I, \Sigma_1, \Gamma_s \vdash \mathtt{FIT\text{-}ARG}_1 \rhd \sigma_1, \Sigma_1^A\\ \ldots\\ \Sigma_I, \Sigma_n, \Gamma_s \vdash \mathtt{FIT\text{-}ARG}_n \rhd \sigma_n, \Sigma_n^A\\ (\Sigma', \sigma_f) = GS_s((\Sigma_1^A, \sigma_1), \ldots, (\Sigma_n^A, \sigma_n)) \text{ is defined}\\ \Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)\\ \mathcal{G}_m(SN) = GS_m = (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_n\rangle, \mathcal{M}_B)\\ \Sigma_I, \Sigma_1, \mathcal{M}_I, \mathcal{M}_1, \Gamma_s, \Gamma_m \vdash \mathtt{FIT\text{-}ARG}_1 \Rightarrow \mathcal{M}_1^A\\ \ldots\\ \Sigma_I, \Sigma_n, \mathcal{M}_I, \mathcal{M}_n, \Gamma_s, \Gamma_m \vdash \mathtt{FIT\text{-}ARG}_n \Rightarrow \mathcal{M}_n^A\end{array}}{\begin{array}{l}\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \mathtt{spec\text{-}inst}\ SN\ \mathtt{FIT\text{-}ARG}_1 \ldots \mathtt{FIT\text{-}ARG}_n \Rightarrow\\ \quad \{M \in \mathbf{Mod}(\Sigma \cup \Sigma') \mid M|_\Sigma \in \mathcal{M},\ M|_{\Sigma'} \in GS_m((\mathcal{M}_1^A, \sigma_1), \ldots, (\mathcal{M}_n^A, \sigma_n))\}\end{array}}$$

For a fitting argument specification, the signature Σ_A given by its argument specification, the signature Σ_P given by the corresponding parameter specification, and the symbol mapping SYMB-MAP-ITEMS+ determine a signature morphism from Σ_P to Σ_A, as explained in Sect. 4.1.5.

$\Sigma_I, \Sigma_P, \Gamma_s \vdash \mathtt{FIT\text{-}ARG} \rhd \sigma, \Sigma_A$

$\Sigma_I, \Sigma_P, \mathcal{M}_I, \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \mathtt{FIT\text{-}ARG} \Rightarrow \mathcal{M}_A$

Γ_s and Γ_m are compatible global environments, Σ_P is an extension of Σ_I, $\mathcal{M}_I$ a class of models over Σ_I and $\mathcal{M}_P$ is a class of models over Σ_P with each model extending some model in $\mathcal{M}_I$; then Σ_A is an extension of Σ_I, σ is a signature morphism from Σ_P to Σ_A which is the identity when restricted to Σ_I, and $\mathcal{M}_A$ is a class of models over Σ_A. Furthermore, the σ-reduct of each model in $\mathcal{M}_A$ is a model of $\mathcal{M}_P$.

$(\Sigma_P, \mathcal{M}_P)$ is one formal parameter of a generic specification. $(\Sigma_A, \mathcal{M}_A)$ is the corresponding actual parameter and σ is the fitting morphism between formal and actual parameter.

[9] See Sect. 4.1.5 for an explanation of the notation $GS_s(\ldots)$.

Rules elided, including the case of FIT-VIEW added in Sect. 4.4.2 below.

$$\boxed{\begin{array}{l} \Sigma_I, \Sigma_P, \Gamma_s \vdash \texttt{FIT-SPEC} \rhd \sigma, \Sigma_A \\ \hfill \Sigma_I, \Sigma_P, \mathcal{M}_I, \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \texttt{FIT-SPEC} \Rightarrow \mathcal{M}_A \end{array}}$$

Γ_s and Γ_m are compatible global environments, Σ_P is an extension of Σ_I, $\mathcal{M}_I$ a class of models over Σ_I and $\mathcal{M}_P$ is a class of models over Σ_P with each model extending some model in $\mathcal{M}_I$; then Σ_A is an extension of Σ_I, σ is a signature morphism from Σ_P to Σ_A which is the identity when restricted to Σ_I, and $\mathcal{M}_A$ is a class of models over Σ_A. Furthermore, the σ-reduct of each model in $\mathcal{M}_A$ is a model of $\mathcal{M}_P$.

$$\frac{\begin{array}{c} \Sigma_I, \Gamma_s \vdash \texttt{SPEC} \rhd \Sigma_A \\ \vdash \texttt{SYMB-MAP-ITEMS*} \rhd r \\ \sigma = (r \cup \{(SY, SY) | SY \in ||\Sigma_I||\})|^{\Sigma_P}_{\Sigma_A} \end{array}}{\Sigma_I, \Sigma_P, \Gamma_s \vdash \texttt{fit-spec SPEC SYMB-MAP-ITEMS*} \rhd \sigma, \Sigma_A}$$

See Sect. 4.1.3 for the definition of $r|^{\Sigma_P}_{\Sigma_A}:\Sigma_P \rightarrow \Sigma_A$, the signature morphism induced by the symbol map r.

$$\frac{\begin{array}{c} \Sigma_I, \Gamma_s \vdash \texttt{SPEC} \rhd \Sigma_A \\ \vdash \texttt{SYMB-MAP-ITEMS*} \rhd r \\ \sigma = (r \cup \{(SY, SY) | SY \in ||\Sigma_I||\})|^{\Sigma_P}_{\Sigma_A} \\ \Sigma_I, \mathcal{M}_I, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}_A \\ \mathcal{M}_A|_\sigma \subseteq \mathcal{M}_P \end{array}}{\Sigma_I, \Sigma_P, \mathcal{M}_I, \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \texttt{fit-spec SPEC SYMB-MAP-ITEMS*} \Rightarrow \mathcal{M}_A}$$

4.4 Views

4.4.1 View Definitions

A view definition declares the view name *VN* to have the type of specification morphisms from SPEC$_1$ to SPEC$_2$, parameter and import specifications as given by GENERICITY, and defines it by the symbol mapping SYMB-MAP-ITEMS+.

SPEC$_1$ gets the empty local environment. The well-formedness conditions for SPEC$_2$ are as if SPEC$_2$ were the body of a generic specification with formal parameter and import specifications as given by GENERICITY.

It is required that the reduct by the specification morphism of each model of the target is a model of the source; otherwise the semantics is undefined.

```
VIEW-DEFN ::= view-defn VIEW-NAME GENERICITY VIEW-TYPE SYMB-MAP-ITEMS*
VIEW-TYPE ::= view-type SPEC SPEC
VIEW-NAME ::= SIMPLE-ID
```

$$\boxed{\Gamma_s \vdash \texttt{VIEW-DEFN} \rhd \Gamma'_s \qquad\qquad \Gamma_s, \Gamma_m \vdash \texttt{VIEW-DEFN} \Rightarrow \Gamma'_m}$$

Γ_s and Γ_m are compatible global environments; then Γ'_s and Γ'_m are compatible environments extending Γ_s and Γ_m, respectively.

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\\ VN \notin Dom(\mathcal{V}_s) \cup Dom(\mathcal{G}_s) \cup Dom(\mathcal{A}_s) \cup Dom(\mathcal{T}_s)\\ \Gamma_s \vdash \texttt{GENERICITY} \rhd (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle)\\ \Sigma_1 \cup \cdots \cup \Sigma_n, \Gamma_s \vdash \texttt{VIEW-TYPE} \rhd (\Sigma_s, \Sigma_t)\\ \vdash \texttt{SYMB-MAP-ITEMS*} \rhd r \qquad \sigma = r|^{\Sigma_s}_{\Sigma_t}\end{array}}{\begin{array}{l}\Gamma_s \vdash \texttt{view-defn}\ VN\ \texttt{GENERICITY}\ \texttt{VIEW-TYPE}\ \texttt{SYMB-MAP-ITEMS*} \rhd\\ \qquad\qquad (\mathcal{G}_s, \mathcal{V}_s \cup \{\, VN \mapsto (\Sigma_s, \sigma, (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle, \Sigma_t))\}, \mathcal{A}_s, \mathcal{T}_s)\end{array}}$$

See Sect. 4.1.3 for the definition of $r|^{\Sigma_s}_{\Sigma_t}{:}\Sigma_s \to \Sigma_t$, the signature morphism induced by the symbol map r.

$$\frac{\begin{array}{c}\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)\\ \Gamma_s \vdash \texttt{GENERICITY} \rhd (\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle)\\ \Sigma_1 \cup \cdots \cup \Sigma_n, \Gamma_s \vdash \texttt{VIEW-TYPE} \rhd (\Sigma_s, \Sigma_t)\\ \vdash \texttt{SYMB-MAP-ITEMS*} \rhd r \qquad \sigma = r|^{\Sigma_s}_{\Sigma_t}\\ \Gamma_s, \Gamma_m \vdash \texttt{GENERICITY} \Rightarrow (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle)\\ \mathcal{M}_P = \{M \in \mathbf{Mod}(\Sigma_P) \mid M|_{\Sigma_I} \in \mathcal{M}_I,\ M|_{\Sigma_i} \in \mathcal{M}_i,\ 1 \leq i \leq n\}\\ \Sigma_1 \cup \cdots \cup \Sigma_n, \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \texttt{VIEW-TYPE} \Rightarrow (\mathcal{M}_s, \mathcal{M}_t)\\ \mathcal{M}_t|_\sigma \subseteq \mathcal{M}_s\end{array}}{\begin{array}{l}\Gamma_s, \Gamma_m \vdash \texttt{view-defn}\ VN\ \texttt{GENERICITY}\ \texttt{VIEW-TYPE}\ \texttt{SYMB-MAP-ITEMS*} \Rightarrow\\ \qquad\qquad (\mathcal{G}_m, \mathcal{V}_m \cup \{\, VN \mapsto (\mathcal{M}_s, (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle, \mathcal{M}_t))\}, \mathcal{A}_m, \mathcal{T}_m)\end{array}}$$

$$\boxed{\Sigma, \Gamma_s \vdash \texttt{VIEW-TYPE} \rhd (\Sigma_s, \Sigma_t) \qquad \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{VIEW-TYPE} \Rightarrow (\mathcal{M}_s, \mathcal{M}_t)}$$

Γ_s and Γ_m are compatible global environments and $\mathcal{M}$ is a class of models over Σ; then Σ_t is an extension of Σ, $\mathcal{M}_t$ is a class of models over Σ_t with each model extending some model of $\mathcal{M}$, and $\mathcal{M}_s$ is a class of models over Σ_s.

$$\frac{\begin{array}{c}\emptyset, \Gamma_s \vdash \texttt{SPEC}_1 \rhd \Sigma_s\\ \Sigma, \Gamma_s \vdash \texttt{SPEC}_2 \rhd \Sigma_t\end{array}}{\Sigma, \Gamma_s \vdash \texttt{view-type}\ \texttt{SPEC}_1\ \texttt{SPEC}_2 \rhd (\Sigma_s, \Sigma_t)}$$

$$\frac{\begin{array}{c}\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \texttt{SPEC}_1 \Rightarrow \mathcal{M}_s\\ \Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SPEC}_2 \Rightarrow \mathcal{M}_t\end{array}}{\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{view-type}\ \texttt{SPEC}_1\ \texttt{SPEC}_2 \Rightarrow (\mathcal{M}_s, \mathcal{M}_t)}$$

4.4.2 Fitting Views

A reference to a fitting argument view refers to the current global environment, and is well-formed as an argument for a parameter $\texttt{SPEC}_i$ only when the global environment includes a (possibly generic) view definition for *VN* (possibly with parameters that can be instantiated by the indicated fitting arguments `FIT-ARG+`) to give a view of type from `SPEC` to $\texttt{SPEC}'$, such that the signatures of `SPEC` and of $\texttt{SPEC}_i$ are the same. The view definition then provides the fitting morphism from the parameter $\texttt{SPEC}_i$ to the argument specification given by the target $\texttt{SPEC}'$ of the view.
Each model of `SPEC` is required to be a model of $\texttt{SPEC}_i$, otherwise the instantiation is undefined. The instantiation of a generic view with some fitting arguments is not well-formed if the instantiation of the target $\texttt{SPEC}'$ of the view with the same fitting arguments is not well-formed.

```
FIT-ARG  ::= ... | FIT-VIEW
FIT-VIEW ::= fit-view VIEW-NAME FIT-ARG*
```

$\Sigma_I, \Sigma_P, \Gamma_s \vdash \texttt{FIT-VIEW} \rhd \sigma, \Sigma_A$

$\Sigma_I, \Sigma_P, \mathcal{M}_I, \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \texttt{FIT-VIEW} \Rightarrow \mathcal{M}_A$

Γ_s and Γ_m are compatible global environments, Σ_P is an extension of Σ_I, $\mathcal{M}_I$ a class of models over Σ_I and $\mathcal{M}_P$ is a class of models over Σ_P with each model extending some model in $\mathcal{M}_I$; then Σ_A is an extension of Σ_I, σ is a signature morphism from Σ_P to Σ_A which is the identity when restricted to Σ_I, and $\mathcal{M}_A$ is a class of models over Σ_A. Furthermore, the σ-reduct of each model in $\mathcal{M}_A$ is a model of $\mathcal{M}_P$.

First we study the situation where *VN* denotes a non-generic view.

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \\ \mathcal{V}_s(\mathit{VN}) = (\Sigma_s, \sigma, (\emptyset, \langle\rangle, \Sigma_t))) \\ \Sigma_s \cup \Sigma_I = \Sigma_P\end{array}}{\Sigma_I, \Sigma_P, \Gamma_s \vdash \texttt{fit-view}\ \mathit{VN} \rhd \sigma \cup id_{\Sigma_I}, \Sigma_I \cup \Sigma_t}$$

Here, Σ_I is the import signature, Σ_P is the parameter signature, which is the source of the view, and Σ_A is the argument signature, which is the target of the view.

$$\begin{tikzcd}
 & \Sigma_s \arrow[r, "\sigma"] \arrow[d, hook] & \Sigma_t \arrow[d, hook] \\
\Sigma_I \arrow[r, hook] & \Sigma_s \cup \Sigma_I \arrow[r, "\sigma \cup id_{\Sigma_I}"] \arrow[d, equal] & \Sigma_I \cup \Sigma_t \\
 & \Sigma_P &
\end{tikzcd}$$

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \\ \mathcal{V}_s(\mathit{VN}) = (\Sigma_s, \sigma, (\emptyset, \langle\rangle, \Sigma_t))) \\ \Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m) \\ \mathcal{V}_m(\mathit{VN}) = (\mathcal{M}_s, (\mathcal{M}_\perp, \langle\rangle, \mathcal{M}_t)) \\ \{M \in \mathbf{Mod}(\Sigma_P) \mid M|_{\Sigma_s} \in \mathcal{M}_s,\ M|_{\Sigma_I} \in \mathcal{M}_I\} \subseteq \mathcal{M}_P\end{array}}{\begin{array}{l}\Sigma_I, \Sigma_P, \mathcal{M}_I, \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \texttt{fit-view}\ \mathit{VN} \Rightarrow \\ \qquad \{M \in \mathbf{Mod}(\Sigma_I \cup \Sigma_t) \mid M|_{\Sigma_I} \in \mathcal{M}_I,\ M|_{\Sigma_t} \in \mathcal{M}_t\}\end{array}}$$

Note that $\mathcal{M}_A|_{\sigma \cup id_{\Sigma_I}} \subseteq \mathcal{M}_P$ follows from $\mathcal{M}_A|_{\sigma \cup id_{\Sigma_I}} \subseteq \mathcal{M}'_s$ and $\mathcal{M}'_s \subseteq \mathcal{M}_P$, where $\mathcal{M}'_s = \{M \in \mathbf{Mod}(\Sigma_P) \mid M|_{\Sigma_s} \in \mathcal{M}_s,\ M|_{\Sigma_I} \in \mathcal{M}_I\}$.

To show that $\mathcal{M}_A|_{\sigma \cup id_{\Sigma_I}} \subseteq \mathcal{M}'_s$ holds, consider $M \in \mathcal{M}_A$. We have to show that $M|_{\sigma \cup id_{\Sigma_I}} \in \mathcal{M}'_s$, that is, $(M|_{\sigma \cup id_{\Sigma_I}})|_{\Sigma_s} \in \mathcal{M}_s$ and $(M|_{\sigma \cup id_{\Sigma_I}})|_{\Sigma_I} \in \mathcal{M}_I$. The following equality holds by the definition of $\sigma \cup id_{\Sigma_I}$:

$$\sigma \cup id_{\Sigma_I} \circ \iota_{\Sigma_I \subseteq \Sigma_I \cup \Sigma_s} = \iota_{\Sigma_I \subseteq \Sigma_I \cup \Sigma_t}$$

Then we have:

$$(M|_{\sigma \cup id_{\Sigma_I}})|_{\Sigma_s} = (M|_{\Sigma_t})|_\sigma \ \in\ \mathcal{M}_s$$

since $M \in \mathcal{M}_A$ implies $M|_{\Sigma_t} \in \mathcal{M}_t$ by the definition of M_A and because $\mathcal{M}_t|_\sigma \subseteq \mathcal{M}_s$ by the properties of view morphisms.

Further, the following equality holds by the above diagram:

$$\iota_{\Sigma_t \subseteq \Sigma_I \cup \Sigma_t} \circ \sigma = \sigma \cup id_{\Sigma_I} \circ \iota_{\Sigma_s \subseteq \Sigma_I \cup \Sigma_s}$$

This implies

$$(M|_{\sigma \cup id_{\Sigma_I}})|_{\Sigma_I} = M|_{\Sigma_I} \ \in\ \mathcal{M}_I$$

since $M \in \mathcal{M}_A$ implies $M|_{\Sigma_I} \in \mathcal{M}_I$ by the definition of $\mathcal{M}_A$.

Now we come to the generic case:

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \\ \mathcal{V}_s(\mathit{VN}) = (\Sigma_s, \sigma, GS_s)) \\ \Sigma_s \cup \Sigma_I = \Sigma_P \\ GS_s = (\Sigma'_I, \langle \Sigma_1, \ldots, \Sigma_n\rangle, \Sigma_B), \quad n \geq 1 \\ \Sigma'_I, \Sigma_1, \Gamma_s \vdash \texttt{FIT-ARG}_1 \rhd \sigma_1, \Sigma_1^A \\ \ldots \\ \Sigma'_I, \Sigma_n, \Gamma_s \vdash \texttt{FIT-ARG}_n \rhd \sigma_n, \Sigma_n^A \\ (\Sigma_A, \sigma'_f) = GS_s((\Sigma_1^A, \sigma_1), \ldots, (\Sigma_n^A, \sigma_n)) \text{ is defined}\end{array}}{\begin{array}{l}\Sigma_I, \Sigma_P, \Gamma_s \vdash \texttt{fit-view}\ \mathit{VN}\ \texttt{FIT-ARG}_1 \ldots \texttt{FIT-ARG}_n \rhd \\ \qquad\qquad\qquad\qquad (\sigma'_f \circ \sigma) \cup id_{\Sigma_I}, \Sigma_I \cup \Sigma_A\end{array}}$$

$$\begin{tikzcd}
 & & \Sigma'_I \arrow[d, hook] & \\
 & & \Sigma_i \arrow[r, "\sigma_i"] \arrow[d, hook] & \Sigma_i^A \arrow[d, hook] \\
 & \Sigma_s \arrow[r, "\sigma"] \arrow[d, hook] & \Sigma_B \arrow[r, "\sigma'_f"] & \Sigma_A \arrow[d, hook] \\
\Sigma_I \arrow[r, hook] & \Sigma_s \cup \Sigma_I \arrow[rr, "(\sigma'_f \circ \sigma) \cup id_{\Sigma_I}"] \arrow[d, equal] & & \Sigma_A \cup \Sigma_I \\
 & \Sigma_P & &
\end{tikzcd}$$

$$\begin{array}{c}
\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \\
\mathcal{V}_s(\mathit{VN}) = (\Sigma_s, \sigma, \mathit{GS}_s)) \\
\mathit{GS}_s = (\Sigma'_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle, \Sigma_B), \quad n \geq 1 \\
\Sigma'_I, \Sigma_1, \Gamma_s \vdash \texttt{FIT-ARG}_1 \rhd \sigma_1, \Sigma_1^A \\
\ldots \\
\Sigma'_I, \Sigma_n, \Gamma_s \vdash \texttt{FIT-ARG}_n \rhd \sigma_n, \Sigma_n^A \\
\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m) \\
\mathcal{V}_m(\mathit{VN}) = (\mathcal{M}_s, \mathit{GS}_m)) \\
\{M \in \mathbf{Mod}(\Sigma_P) \mid M|_{\Sigma_s} \in \mathcal{M}_s,\ M|_{\Sigma_I} \in \mathcal{M}_I\} \subseteq \mathcal{M}_P \\
\mathit{GS}_m = (\mathcal{M}'_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_n \rangle, \mathcal{M}_B) \\
\Sigma'_I, \Sigma_1, \mathcal{M}'_I, \mathcal{M}_1, \Gamma_s, \Gamma_m \vdash \texttt{FIT-ARG}_1 \Rightarrow \mathcal{M}_1^A \\
\ldots \\
\Sigma'_I, \Sigma_n, \mathcal{M}'_I, \mathcal{M}_n, \Gamma_s, \Gamma_m \vdash \texttt{FIT-ARG}_n \Rightarrow \mathcal{M}_n^A \\
\mathcal{M}_A = \mathit{GS}_m((\mathcal{M}_1^A, \sigma_1), \ldots, (\mathcal{M}_n^A, \sigma_n)) \\
\hline
\Sigma_I, \Sigma_P, \mathcal{M}_I, \mathcal{M}_P, \Gamma_s, \Gamma_m \vdash \texttt{fit-view}\ \mathit{VN}\ \texttt{FIT-ARG}_1 \ldots \texttt{FIT-ARG}_n \Rightarrow \\
\qquad\qquad \{M \in \mathbf{Mod}(\Sigma_I \cup \Sigma_A) \mid M|_{\Sigma_I} \in \mathcal{M}_I,\ M|_{\Sigma_A} \in \mathcal{M}_A\}
\end{array}$$

A similar argument as for the non-generic case shows that $\mathcal{M}'|_{(\sigma'_f \circ \sigma) \cup id_{\Sigma_I}} \subseteq \mathcal{M}'_s$, where $\mathcal{M}' = \{M \in \mathbf{Mod}(\Sigma_I \cup \Sigma_A) \mid M|_{\Sigma_I} \in \mathcal{M}_I,\ M|_{\Sigma_A} \in \mathcal{M}_A\}$. Then, $\mathcal{M}'|_{(\sigma'_f \circ \sigma) \cup id_{\Sigma_I}} \subseteq \mathcal{M}_P$ follows from $\mathcal{M}'|_{(\sigma'_f \circ \sigma) \cup id_{\Sigma_I}} \subseteq \mathcal{M}'_s$ and $\mathcal{M}'_s \subseteq \mathcal{M}_P$.

4.5 Symbol Lists and Mappings

The semantics of a part of this section is necessarily dependent on the underlying basic specification framework formalized as an institution with symbols [39].

4.5.1 Symbol Lists

Symbol lists are used in hiding reductions. They can be identifiers (matching to any symbol with the identifier as name) or fully-qualified symbols. Sublists may also be qualified with kinds (sort, function, predicate).

```
SYMB-ITEMS ::= symb-items SYMB-KIND SYMB+
SYMB-KIND  ::= implicit | sorts-kind | ops-kind | preds-kind
SYMB       ::= ID | QUAL-ID
QUAL-ID    ::= qual-id ID TYPE
TYPE       ::= OP-TYPE | PRED-TYPE
```

Note that the described behavior is achieved by the definitions from Sect. 4.1.3; here, only symbol lists are assembled.

Casl-specific rules for symbol lists

$$\boxed{k \vdash \texttt{SYMB} \rhd SY}$$

$k \in$ `SYMB-KIND`; then $SYs \in Sym$ is a symbol.

$$\frac{Ident \in \texttt{ID}}{\texttt{implicit} \vdash Ident \rhd (implicit, Ident)}$$

$$\frac{}{\texttt{implicit} \vdash \texttt{qual-id}\ f\ (\texttt{total-op-type}(\texttt{sort-list}\ s_1 \ldots s_n)\ s) \rhd f^{\mathsf{t}}_{\langle s_1,\ldots,s_n\rangle,s}}$$

$$\frac{}{\texttt{implicit} \vdash \texttt{qual-id}\ f\ (\texttt{partial-op-type}(\texttt{sort-list}\ s_1 \ldots s_n)\ s) \rhd f^{\mathsf{p}}_{\langle s_1,\ldots,s_n\rangle,s}}$$

$$\frac{}{\texttt{implicit} \vdash \texttt{qual-id}\ p\ (\texttt{pred-type}(\texttt{sort-list}\ s_1 \ldots s_n)) \rhd p_{\langle s_1,\ldots,s_n\rangle}}$$

$$\frac{s \in \texttt{ID}}{\texttt{sorts-kind} \vdash s \rhd s}$$

$$\frac{}{\texttt{ops-kind} \vdash f \rhd (fun, f)}$$

$$\frac{}{\texttt{ops-kind} \vdash \texttt{qual-id}\ f\ (\texttt{total-op-type}(\texttt{sort-list}\ s_1 \ldots s_n)\ s) \rhd f^{\mathsf{t}}_{\langle s_1,\ldots,s_n\rangle,s}}$$

$$\frac{}{\texttt{ops-kind} \vdash \texttt{qual-id}\ f\ (\texttt{partial-op-type}(\texttt{sort-list}\ s_1 \ldots s_n)\ s) \rhd f^{\mathsf{p}}_{\langle s_1,\ldots,s_n\rangle,s}}$$

$$\frac{}{\texttt{preds-kind} \vdash p \rhd (pred, p)}$$

$$\frac{}{\texttt{preds-kind} \vdash \texttt{qual-id}\ p\ (\texttt{pred-type}(\texttt{sort-list}\ s_1 \ldots s_n)) \rhd p_{\langle s_1,\ldots,s_n\rangle}}$$

end of Casl-specific rules

$$\boxed{\vdash \texttt{SYMB-ITEMS} \rhd SYs}$$

$SYs \subseteq \mathit{Sym}$ is a set of symbols.

$$\frac{k \vdash \texttt{SYMB}_1 \rhd SY_1 \qquad \cdots \qquad k \vdash \texttt{SYMB}_n \rhd SY_n}{\vdash \texttt{symb-items}\ k\ \texttt{SYMB}_1\ \cdots\ \texttt{SYMB}_n \rhd \{SY_1, \ldots, SY_n\}}$$

A symbol list determines a set of qualified symbols, obtained from the listed symbols with reference to a given signature.

$$\boxed{SSYs \vdash \texttt{SYMB-ITEMS+} \rhd SSYs'}$$

$SSYs \subseteq \mathit{SigSym}$ is a set of signature symbols; then $SSYs' \subseteq SSYs$.

$$\frac{\vdash \texttt{SYMB-ITEMS}_1 \rhd SYs_1 \qquad \cdots \qquad \vdash \texttt{SYMB-ITEMS}_n \rhd SYs_n}{\begin{array}{l} SSYs \vdash \texttt{SYMB-ITEMS}_1\ \cdots\ \texttt{SYMB-ITEMS}_n \rhd \\ \qquad \{SY \in SSYs \mid SSY \text{ matches some } SY \in SYs_1 \cup \cdots \cup SYs_n\} \end{array}}$$

4.5.2 Symbol Mappings

Symbol mappings are used in translations, revealing reductions, fitting arguments, and views. They can map identifiers (matching to any symbol with the identifier as name) or fully-qualified symbols. Sub-lists in the mapping may also be qualified with kinds (sort, function, predicate).

```
SYMB-MAP-ITEMS ::= symb-map-items SYMB-KIND SYMB-OR-MAP+
SYMB-OR-MAP    ::= SYMB | SYMB-MAP
SYMB-MAP       ::= symb-map SYMB SYMB
```

Note that the described behavior is achieved by the definitions from Sect. 4.1.3; here, only symbol maps are assembled.

Casl-specific rules for symbol maps

$$\boxed{k \vdash \texttt{SYMB-OR-MAP} \rhd r}$$

$k \in \mathit{SymKind}$ is a qualification kind; then $r \in \mathit{SymMap}$ is a symbol map.

$$\frac{k \vdash \texttt{SYMB} \rhd SY}{k \vdash \texttt{SYMB}\ \text{qua}\ \texttt{SYMB-OR-MAP} \rhd \{(SY, SY)\}}$$

$$\frac{\begin{array}{c} k \vdash \texttt{SYMB} \rhd SY \\ k \vdash \texttt{SYMB'} \rhd SY' \end{array}}{k \vdash \texttt{symb-map}\ \texttt{SYMB}\ \texttt{SYMB'} \rhd \{(SY, SY')\}}$$

$$\boxed{\vdash \texttt{SYMB-MAP-ITEMS} \rhd r}$$

$r \in \mathit{SymMap}$ is a symbol map.

$$\frac{k \vdash \texttt{SYMB-OR-MAP}_1 \rhd r_1 \qquad \cdots \qquad k \vdash \texttt{SYMB-OR-MAP}_n \rhd r_n}{\vdash \texttt{symb-map-items}\ k\ \texttt{SYMB-OR-MAP}_1\ \cdots\ \texttt{SYMB-OR-MAP}_n \rhd r_1 \cup \cdots \cup r_n}$$

end of C*ASL-specific rules*

A list of symbol maps determines a set of qualified symbols, obtained from the first components of the listed symbol maps with reference to a given signature, together with a mapping of these symbols to qualified symbols obtained from the second components of the listed symbol maps.

$$\boxed{\vdash \texttt{SYMB-MAP-ITEMS*} \rhd r}$$

$r \in \mathit{SymMap}$ is a symbol map.

$$\frac{\vdash \texttt{SYMB-MAP-ITEMS}_1 \rhd r_1 \qquad \cdots \qquad \vdash \texttt{SYMB-MAP-ITEMS}_n \rhd r_n}{\vdash \texttt{SYMB-MAP-ITEMS}_1\ \cdots\ \texttt{SYMB-MAP-ITEMS}_n \rhd r_1 \cup \cdots \cup r_n}$$

Note that SYMB-MAP-ITEMS+, as used in a RENAMING and a REVEALED, is just the case where $n \geq 1$.

The transition to the level of qualified symbols is performed in the semantics rules for translations, instantiation etc, using the definitions of Sect. 4.1.3.

4.6 Compound Identifiers

The components of a compound identifier may (but need not) themselves identify symbols that are specified in the declared parameters of a generic specification.

```
SORT-ID        ::= ... | COMP-SORT-ID
MIX-TOKEN      ::= ... | COMP-MIX-TOKEN
COMP-SORT-ID   ::= comp-sort-id WORDS ID+
COMP-MIX-TOKEN ::= comp-mix-token ID+
```

Note that by the above additions to the grammar of the abstract syntax, the definitions of SORT-ID and ID have changed.

This also influences the semantic domains introduced in Sect. 2.3 and 3.1.1:

$$\begin{aligned} \mathit{Sort} &= \mathtt{SORT\text{-}ID} \\ \mathit{FunName} &= \mathtt{ID} \uplus \{\mathit{em}\} \uplus \{\mathit{pr}\} \\ \mathit{PredName} &= \mathtt{ID} \uplus \{\mathit{in}(s) \mid s \in \mathit{Sort}\} \end{aligned}$$

Also note that in the definition of the set of symbols of a signature in Sect. 4.1.1, we have adopted the convention that `SORT-ID` is regarded as a subset of `ID`; hence `comp-sort-id WORDS ID+` is identified with `id` (`comp-mix-token ID+`).

> When a compound identifier is used to name a symbol in the body of a generic specification, the translation determined by fitting arguments to parameters applies to the components of the compound identifier as well.

Given an identifier map $h \subseteq \mathtt{ID} \times \mathtt{ID}$, define $\mathit{ExtID}(h) \subseteq \mathtt{ID} \times \mathtt{ID}$ and $\mathit{ExtMIX\text{-}TOKEN}(h) \subseteq \mathtt{MIX\text{-}TOKEN} \times \mathtt{MIX\text{-}TOKEN}$ as the least relations satisfying

- $h \subseteq \mathit{ExtID}(h)$,
- $(\mathtt{id}\ \mathit{mixt}_1 \ldots \mathit{mixt}_n, \mathtt{id}\ \mathit{mixt}'_1 \ldots \mathit{mixt}'_n) \in \mathit{ExtID}(h)$ if $(\mathit{mixt}_i, \mathit{mixt}'_i) \in \mathit{ExtMIX\text{-}TOKEN}(h)$ or $\mathit{mixt}_i = \mathit{mixt}'_i$ for $i = 1, \ldots, n$ and $\mathit{mixt}_j \neq \mathit{mixt}'_j$ for some $1 \leq j \leq n$,
- $((\mathtt{comp\text{-}mix\text{-}token}\ \mathit{ci}_1 \ldots \mathit{ci}_n), (\mathtt{comp\text{-}mix\text{-}token}\ \mathit{ci}'_1 \ldots \mathit{ci}'_n)) \in \mathit{ExtMIX\text{-}TOKEN}(h)$, provided that $(\mathit{ci}_i, \mathit{ci}'_i) \in \mathit{ExtID}(h)$ or $\mathit{ci}_i = \mathit{ci}'_i$ for $i = 1, \ldots, n$.

The definition of *Ext* on page 196 has to be changed in the following way to accommodate compound identifiers: Given a signature symbol map $h \subseteq \mathit{SigSym} \times \mathit{SigSym}$,

$$\mathit{Ext}(h) = h \cup \mathit{IDAsSym}(\mathit{ExtID}(h'))^{10}$$

where

$$\begin{aligned} h' = \{(\mathit{Ident}, \mathit{Ident}') \mid\ & \mathit{Ident}, \mathit{Ident}' \in \mathtt{ID}, \\ & (\mathit{SSY}, \mathit{SSY}') \in h, \\ & \mathit{SSY} \text{ matches } \mathit{IDAsSym}(\mathit{Ident}), \\ & \mathit{SSY}' \text{ matches } \mathit{IDAsSym}(\mathit{Ident}')\}^{11}. \end{aligned}$$

[10] *IDAsSym* is applied to a binary relation by applying it component-wise. The union of h and $\mathit{IDAsSym}(\mathit{ExtID}(h'))$ may lead to a relation that is not a function (and, therefore, undefinedness of the instantiation of the generic specification, because the symbol map does not induce a signature morphism), if a compound identifier is mapped both explicitly by the fitting morphism and implicitly by the extension mechanism. Note that this can happen in CASL only for sort symbols, since fully qualified symbols are never identifiers.

[11] Notice that h' may fail to be a function even if h is one, destroying the definedness of the instantiation of a generic specification. In CASL, this may happen, for example, if two different profiles of a function are mapped to different names, and the function name occurs in a compound identifier.

The embedding, projection and membership symbols *em*, *pr* and $in(s)$ are not contained in `ID` and hence are not subject to the above extension mechanism for compound identifiers.

> Subsort embeddings between component sorts do *not* induce subsort embeddings between the compound sorts.
> Instantiation, however, does preserve subsorts. Compound identifiers must not be identified through the identification of components by the fitting morphism.

This behavior is automatically achieved by the use of a final signature morphism in the definition of signature morphism induced by a symbol map (which is used in the definition of the extension of a signature morphism along a signature extension, which is in turn used in the semantics of instantiations of generic specifications).

5

Architectural Specification Semantics

Architectural specifications are for imposing structure on models, expressing their *composition* from component units.

The component units may all be regarded as *unit functions*: functions without arguments give self-contained units; functions with arguments use such units in constructing further units.

The specification of a unit function indicates the properties to be assumed of the arguments, and the properties to be guaranteed of the result. In Casl, self-contained units are simply subsorted models as defined in Chap. 3, and their properties are expressed by ordinary (perhaps structured) specifications.

Thus a unit function maps models of argument specifications to models of a result specification. A specification of such functions can be simply a list of the argument specifications together with the result specification. Thinking of argument and result specifications as *types* of models, a specification of a unit function may be regarded as a *function type*. Such a specification describes the class of all *persistent* functions from *compatible* tuples of models of the argument specifications to models of the result specification.

An entire *architectural specification* is a collection of unit function specifications, together with a description of how the functions are to be composed to give a resulting unit. A model of an architectural specification is a collection of unit functions with the specified types or definitions, together with the result of composing them as described.

In Sect. 5.1 we define semantic domains to model the concepts mentioned above. At the risk of stretching informality and neglecting precision, the vocabulary of key semantic notions to be introduced may be summarized as follows:

concept	static semantics	model semantics (individual entities)	model semantics (classes of entities)
unit	unit signature	unit	unit specification
unit expression	unit signature	unit evaluator	—
unit declarations and definitions	static unit context	unit environment	unit context
architectural specification	architectural signature	architectural model	architectural specification

Sections 5.2–5.5 then give static and model semantics for architectural specifications, extending what was provided for basic and structured specifications. Finally, in Sect. 5.6, we refine the static analysis for architectural specifications by giving an *extended static semantics*, which gathers considerably more static information and helps to discharge some of the requirements in the model semantics.

5.1 Architectural Concepts

To formally explicate the meaning of architectural constructs we need semantic domains to model the new notions hinted at above.

First, some very preliminary definitions:

$$\begin{aligned} \mathit{UN} &\in \mathit{UnitName} = \texttt{SIMPLE-ID} \\ \mathit{ASN} &\in \mathit{ArchSpecName} = \texttt{SIMPLE-ID} \\ \mathit{USN} &\in \mathit{UnitSpecName} = \texttt{SIMPLE-ID} \end{aligned}$$

Unit signatures, static unit contexts and architectural signatures carry static information necessary for 'typing' unit terms and expressions, unit declarations and definitions, and architectural specifications.

An *architectural signature* consists of a unit signature (for the result unit) together with a *static unit context*: unit signatures for the component units named by unit names (see below). A *unit signature* gives a sequence of signatures[1] for the unit arguments and a signature that extends their union for the result of the unit applications. For non-generic units this reduces to their (result) signature. To take into account constraints on applications of generic units, the signatures of their imports are stored as well.

$$\begin{aligned} &\Sigma_1, \ldots, \Sigma_n \to \Sigma \\ &\qquad \text{or } \overline{\Sigma} \to \Sigma \in \mathit{ParUnitSig} = \mathit{FinSeq}(\mathbf{Sig}) \times \mathbf{Sig} \\ &\qquad\qquad U\Sigma \in \mathit{UnitSig} = \mathit{ParUnitSig} \cup \mathbf{Sig} \end{aligned}$$

[1] Unless explicitly stated otherwise, in this chapter *signature* means *subsorted signature*, as introduced in Chap. 3.

However, the semantics of Casl architectural constructs given here is independent of the underlying institution, provided the institution comes equipped with the structure introduced in Chap. 4 for the semantics of Casl structured specifications. The only extra assumptions we rely on here, which hold for the (subsorted) institution of Casl, are that the empty signature is included in all signatures, it has exactly one model, and that the sinks of signature morphisms given by signature unions are mapped by the model functor to sources that are jointly injective.

$$
\begin{aligned}
&(\Sigma^I, \Sigma_1, \ldots, \Sigma_n \to \Sigma)\\
&\quad\text{or } (\Sigma^I, \overline{\Sigma} \to \Sigma)\\
&\qquad\text{or } (\Sigma^I, U\Sigma) \in \mathit{ImpUnitSig} = \mathbf{Sig} \times \mathit{UnitSig}\\
&\qquad\qquad C_s \in \mathit{StUnitCtx} = \mathit{UnitName} \overset{\text{fin}}{\to} (\mathit{ImpUnitSig} \cup \mathbf{Sig})\\
&\quad(C_s, U\Sigma) \text{ or } A\Sigma \in \mathit{ArchSig} = \mathit{StUnitCtx} \times \mathit{UnitSig}
\end{aligned}
$$

For a unit signature $\Sigma_1, \ldots, \Sigma_n \to \Sigma$ in *ParUnitSig*, we require $n \geq 1$ and Σ to be an extension of $\Sigma_1 \cup \ldots \cup \Sigma_n$. For a unit with import signature $(\Sigma^I, U\Sigma)$ in *ImpUnitSig*, we require $U\Sigma \in \mathit{ParUnitSig}$ and then, for $U\Sigma = \Sigma_1, \ldots, \Sigma_n \to \Sigma$, that Σ is an extension of $\Sigma^I \cup \Sigma_1 \cup \ldots \cup \Sigma_n$. We write $C_s^\emptyset$ for the empty static context.

Units may be regarded as unit functions: functions without arguments give self-contained units, which are CASL models; functions with arguments use such units in constructing further units.

A *non-generic unit* is just a CASL model. A *generic unit* over a unit signature $\Sigma_1, \ldots, \Sigma_n \to \Sigma$ is a (partial) function mapping sequences of compatible models over $\Sigma_1, \ldots, \Sigma_n$ to models over Σ. Models over $\Sigma_1, \ldots, \Sigma_n$ are *compatible* if they can be amalgamated to a model over $\Sigma_1 \cup \ldots \cup \Sigma_n$.

$$
\begin{aligned}
&M \in \mathbf{Unit}(\Sigma) = \mathbf{Mod}(\Sigma)\\
&\langle M_1, \ldots, M_n \rangle\\
&\quad\text{or } \overline{M} \in \mathbf{CompMod}(\Sigma_1, \ldots, \Sigma_n) =\\
&\qquad\qquad \{\langle M|_{\Sigma_1}, \ldots, M|_{\Sigma_n} \rangle \mid M \in \mathbf{Mod}(\Sigma_1 \cup \ldots \cup \Sigma_n)\}\\
&F \in \mathbf{Unit}(\Sigma_1, \ldots, \Sigma_n \to \Sigma) =\\
&\qquad\qquad \mathbf{CompMod}(\Sigma_1, \ldots, \Sigma_n) \rightharpoonup \mathbf{Mod}(\Sigma)\\
&U \in \mathbf{Unit} = \textstyle\bigcup_{U\Sigma \in \mathit{UnitSig}} \mathbf{Unit}(U\Sigma)
\end{aligned}
$$

A generic unit $F \in \mathbf{Unit}(\Sigma_1, \ldots, \Sigma_n \to \Sigma)$ is required to *protect* its arguments, i.e., for each $\langle M_1, \ldots, M_n \rangle \in \mathit{Dom}(F)$, $(F\langle M_1, \ldots, M_n \rangle)|_{\Sigma_1} = M_1$, $\ldots$, $(F\langle M_1, \ldots, M_n \rangle)|_{\Sigma_n} = M_n$.

Given compatible models $(M_1, \ldots, M_n) \in \mathbf{CompMod}(\Sigma_1, \ldots, \Sigma_n)$, we write $M_1 \oplus \ldots \oplus M_n$ for their *amalgamation*, i.e., the unique model in $\mathbf{Mod}(\Sigma_1 \cup \ldots \cup \Sigma_n)$ such that $(M_1 \oplus \ldots \oplus M_n)|_{\Sigma_1} = M_1, \ldots, (M_1 \oplus \ldots \oplus M_n)|_{\Sigma_n} = M_n$ (uniqueness is ensured by the property of signature unions mentioned in footnote 1). For $\mathcal{M}_1 \subseteq \mathbf{Mod}(\Sigma_1), \ldots, \mathcal{M}_n \subseteq \mathbf{Mod}(\Sigma_n)$, $\mathcal{M}_1 \oplus \ldots \oplus \mathcal{M}_n = \{M \in \mathbf{Mod}(\Sigma_1 \cup \ldots \cup \Sigma_n) \mid M|_{\Sigma_1} \in \mathcal{M}_1, \ldots M|_{\Sigma_n} \in \mathcal{M}_n\}$.

We extend this to 'amalgamate' models with generic units.

Given a generic unit signature $\Sigma_1, \ldots, \Sigma_n \to \Sigma$ and a signature Σ', we write $(\Sigma_1, \ldots, \Sigma_n \to \Sigma) \cup \Sigma'$ for $\Sigma_1, \ldots, \Sigma_n \to (\Sigma \cup \Sigma')$. Then we say that a unit $F \in \mathbf{Unit}(\Sigma_1, \ldots, \Sigma_n \to \Sigma)$ is *compatible* with a model $M' \in \mathbf{Mod}(\Sigma')$ if for all $\overline{M} \in \mathit{Dom}(F)$ such that $\overline{M}$ and M' are compatible, $F(\overline{M})$ and M' are compatible as well. If this is the case, we define

$$F \oplus M' = \{\overline{M} \mapsto (F(\overline{M}) \oplus M') \mid \overline{M}, M' \text{are compatible}, \overline{M} \in \mathit{Dom}(F)\},$$

so that $F \oplus M' \in \mathbf{Unit}((\Sigma_1, \ldots, \Sigma_n \to \Sigma) \cup \Sigma')$.

An architectural specification denotes a class of architectural models over an architectural signature.

An *architectural model* over an architectural signature $(C_s, U\Sigma)$ consists of a (result) unit over the unit signature $U\Sigma$ and a *unit environment*: a collection of (component) units over the signatures given in C_s, named by the respective unit names. In Casl, all architectural specifications are deterministic in the sense that no environment determines more than one result unit.

$$\begin{aligned} E \in \mathbf{UnitEnv}(C_s) \subseteq \mathbf{UnitEnv} &= \mathit{UnitName} \overset{\text{fin}}{\to} \mathbf{Unit} \\ (E, U) \text{ or } AM \in \mathbf{ArchMod}(C_s, U\Sigma) &= \mathbf{UnitEnv}(C_s) \times \mathbf{Unit}(U\Sigma) \\ \mathbf{ArchSpec}(A\Sigma) &= \mathit{Set}(\mathbf{ArchMod}(A\Sigma)) \\ \mathcal{AM} \in \mathbf{ArchSpec} &= \textstyle\bigcup_{A\Sigma \in \mathit{ArchSig}} \mathbf{ArchSpec}(A\Sigma) \end{aligned}$$

A unit environment E is in $\mathbf{UnitEnv}(C_s)$ if $\mathit{Dom}(E) \supseteq \mathit{Dom}(C_s)$ and for each $UN \in \mathit{Dom}(C_s)$ with $C_s(UN) = (\Sigma^I, U\Sigma)$ or $C_s(UN) = U\Sigma$, $E(UN) \in \mathbf{Unit}(U\Sigma)$.

Any architectural specification $\mathcal{AM} \in \mathbf{ArchSpec}$ is required to be functional: if for some $E \in \mathbf{UnitEnv}$, $(E, UN_1) \in \mathcal{AM}$ and $(E, UN_2) \in \mathcal{AM}$ then UN_1 and UN_2 coincide.

Unit specifications denote sets of units:

$$\begin{aligned} \mathbf{UnitSpec}(U\Sigma) &= \mathit{Set}(\mathbf{Unit}(U\Sigma)) \\ \mathcal{U} \in \mathbf{UnitSpec} &= \textstyle\bigcup_{U\Sigma \in \mathit{UnitSig}} \mathbf{UnitSpec}(U\Sigma) \end{aligned}$$

Unit terms and unit expressions, used for instance to define the result units, denote *unit evaluators* which build a unit from a unit environment that records the units the term might involve.

$$\begin{aligned} \mathbf{UnitEval}(U\Sigma) &= \mathbf{UnitEnv} \rightharpoonup \mathbf{Unit}(U\Sigma) \\ \mathit{UEv} \in \mathbf{UnitEval} &= \textstyle\bigcup_{U\Sigma \in \mathit{UnitSig}} \mathbf{UnitEval}(U\Sigma) \\ \mathbf{ModEval}(\Sigma) &= \mathbf{UnitEnv} \rightharpoonup \mathbf{Mod}(\Sigma) \\ \mathit{MEv} \in \mathbf{ModEval} &= \textstyle\bigcup_{\Sigma \in \mathbf{Sig}} \mathbf{ModEval}(\Sigma) \end{aligned}$$

We require that unit evaluators are monotone in the following sense: for $\mathit{UEv} \in \mathbf{UnitEval}$, if $E \in \mathit{Dom}(\mathit{UEv})$ then for all $E' \supseteq E$, $\mathit{UEv}(E') = \mathit{UEv}(E)$ (in particular $E' \in \mathit{Dom}(\mathit{UEv})$ as well). Similar requirement is imposed on model evaluators in $\mathbf{ModEval}$; note also that since signatures are in fact unit signatures as well, we have $\mathbf{ModEval} \subseteq \mathbf{UnitEval}$.

Given any unit signature $U\Sigma$, unit specification $\mathcal{U} \subseteq \mathbf{Unit}(U\Sigma)$, signature Σ^I, and model evaluator $\mathit{MEv} \in \mathbf{ModEval}(\Sigma^I)$, the *import extension* of $\mathcal{U}$ by MEv,

$$\mathcal{U} \oplus \mathit{MEv} \subseteq \mathbf{UnitEval}(U\Sigma \cup \Sigma^I)$$

is the empty set if for some $E \in \mathit{Dom}(\mathit{MEv})$ there is no $F \in \mathcal{U}$ that is compatible with $\mathit{MEv}(E)$; otherwise $\mathcal{U} \oplus \mathit{MEv}$ is the set of all unit evaluators

$UEv \in \mathbf{UnitEval}(U\Sigma \cup \Sigma^I)$ such that $Dom(UEv) = Dom(MEv)$ and for $E \in Dom(UEv)$, $UEv(E) = F \oplus MEv(E)$ for some $F \in \mathcal{U}$ that is compatible with $MEv(E)$. (Notice that due to the definitions of compatibility and amalgamation above, this notion applies to both non-generic and generic unit specifications.)

Unit contexts are sets of unit environments, and thus record constraints on units as well as dependencies between them.

One might think that all we need to know about units declared within an architectural specification are their individual specifications. However, since declared units may rely on imported units, non-trivial dependencies between them occur and should be taken into account. Consequently, *unit contexts* are sets of unit environments, thus recording constraints on as well as dependencies between units introduced.

$$C \in \mathbf{UnitCtx} = Set(\mathbf{UnitEnv})$$

Unit contexts are required to be 'closed' in the sense that if $E \in C$ then for all $E' \supseteq E$, $E' \in C$ as well.

The 'empty' unit context $C^{\emptyset} = \mathbf{UnitEnv}(C_s^{\emptyset}) = \mathbf{UnitEnv}$ imposes no constraints on units, and so consists of all unit environments.

We use two auxiliary notations to manipulate unit contexts. Given any unit context C, unit name UN, class $\mathcal{U}$ of units, and unit evaluator UEv with $C \subseteq Dom(UEv)$, we define

$$C[UN/\mathcal{U}] = \{E + \{UN \mapsto U\} \mid E \in C, U \in \mathcal{U}\}$$

and

$$C[UN/UEv] = \{E + \{UN \mapsto UEv(E)\} \mid E \in C\}.$$

'+' denotes the 'overwriting' union of partial functions: given two unit environments E and E', $Dom(E + E') = Dom(E) \cup Dom(E')$ and then for $UN \in Dom(E + E')$, $(E + E')(UN) = E'(UN)$ if $UN \in Dom(E')$ while $(E + E')(UN) = E(UN)$ otherwise.

The entities used in the model semantics are required to be compatible with the corresponding entities of the static semantics.

A unit context $C \in \mathbf{UnitCtx}$ is *compatible* with a static unit context $C_s \in StUnitCtx$ if $C \subseteq \mathbf{UnitEnv}(C_s)$. Moreover, given a unit context C that is compatible with a static unit context C_s, and $C'_s \in StUnitCtx$ and $C' \in \mathbf{UnitCtx}$, C'_s and C' are *compatible extensions* of C_s and C if $Dom(C'_s)$ and $Dom(C_s)$ are disjoint, and $C \cap C'$ is compatible with $C_s \cup C'_s$. Furthermore, $U\Sigma \in UnitSig$ and $UEv \in \mathbf{UnitEval}$ are *compatible additions* to C_s and C if $UEv \in \mathbf{UnitEval}(U\Sigma)$ and $C \subseteq Dom(UEv)$.

As introduced elsewhere (cf. Sect. 6.1), model (resp., static) global environments Γ_m (resp., Γ_s) contain an architectural specification component

$\mathcal{A}_m\colon \mathit{ArchSpecName} \overset{\text{fin}}{\to} \mathbf{ArchSpec}$ (resp., $\mathcal{A}_s\colon \mathit{ArchSpecName} \overset{\text{fin}}{\to} \mathit{ArchSig}$). Moreover, the definition of compatibility of static and model global environments is extended here in the obvious way: $\Gamma_s = (\ldots, \mathcal{A}_s, \ldots)$ and $\Gamma_m = (\ldots, \mathcal{A}_m, \ldots)$ are *compatible* only if $\mathit{Dom}(\mathcal{A}_s) = \mathit{Dom}(\mathcal{A}_m)$ and for each $\mathit{ASN} \in \mathit{Dom}(\mathcal{A}_s)$, $\mathcal{A}_m(\mathit{ASN}) \in \mathbf{ArchSpec}(\mathcal{A}_s(\mathit{ASN}))$.

Similarly, model (resp., static) global environments Γ_m (resp., Γ_s) contain a unit specification component $\mathcal{T}_m\colon \mathit{UnitSpecName} \overset{\text{fin}}{\to} \mathbf{UnitSpec}$ (resp., $\mathcal{T}_s\colon \mathit{UnitSpecName} \overset{\text{fin}}{\to} \mathit{UnitSig}$). Moreover, the definition of compatibility of static and model global environments is further extended in the obvious way: $\Gamma_s = (\ldots, \mathcal{T}_s, \ldots)$ and $\Gamma_m = (\ldots, \mathcal{T}_m, \ldots)$ are *compatible* only if $\mathit{Dom}(\mathcal{T}_s) = \mathit{Dom}(\mathcal{T}_m)$ and for each $\mathit{USN} \in \mathit{Dom}(\mathcal{T}_s)$, $\mathcal{T}_m(\mathit{USN}) \in \mathbf{UnitSpec}(\mathcal{T}_s(\mathit{USN}))$.

The rest of this chapter indicates the semantics of the constructs of *architectural* specifications, extending what was provided for basic and structured specifications.

5.2 Architectural Specification Definitions

An architectural specification definition `ARCH-SPEC-DEFN` defines the name `ARCH-SPEC-NAME` to refer to the architectural specification `ARCH-SPEC`, extending the global environment (which must not already include a definition for `ARCH-SPEC-NAME`). The local environment given to `ARCH-SPEC` is empty.

```
ARCH-SPEC-DEFN ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC      ::= BASIC-ARCH-SPEC | ARCH-SPEC-NAME
ARCH-SPEC-NAME ::= SIMPLE-ID
```

$$\boxed{\Gamma_s \vdash \texttt{ARCH-SPEC-DEFN} \rhd \Gamma'_s \qquad\qquad \Gamma_s, \Gamma_m \vdash \texttt{ARCH-SPEC-DEFN} \Rightarrow \Gamma'_m}$$

Γ_s and Γ_m are compatible global environments; then Γ'_s and Γ'_m are compatible as well, and extend Γ_s and Γ_m, respectively.

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\\ \mathit{ASN} \notin \mathit{Dom}(\mathcal{G}_s) \cup \mathit{Dom}(\mathcal{V}_s) \cup \mathit{Dom}(\mathcal{A}_s) \cup \mathit{Dom}(\mathcal{T}_s)\\ \Gamma_s \vdash \texttt{ARCH-SPEC} \rhd A\Sigma\end{array}}{\Gamma_s \vdash \texttt{arch-spec-defn}\ \mathit{ASN}\ \ \texttt{ARCH-SPEC} \rhd (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s \cup \{\mathit{ASN} \mapsto A\Sigma\}, \mathcal{T}_s)}$$

$$\frac{\begin{array}{c}\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)\\ \Gamma_s, \Gamma_m \vdash \texttt{ARCH-SPEC} \Rightarrow \mathcal{AM}\end{array}}{\begin{array}{l}\Gamma_s, \Gamma_m \vdash \texttt{arch-spec-defn}\ \mathit{ASN}\ \ \texttt{ARCH-SPEC} \Rightarrow\\ \qquad\qquad\qquad\qquad (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m \cup \{\mathit{ASN} \mapsto \mathcal{AM}\}, \mathcal{T}_m)\end{array}}$$

$$\boxed{\varGamma_s \vdash \mathtt{ARCH\text{-}SPEC} \rhd A\varSigma \qquad\qquad \varGamma_s, \varGamma_m \vdash \mathtt{ARCH\text{-}SPEC} \Rightarrow \mathcal{AM}}$$

$\varGamma_s$ and $\varGamma_m$ are compatible global environments; then $\mathcal{AM} \in \mathbf{ArchSpec}(A\varSigma)$.

$$\frac{\varGamma_s \vdash \mathtt{BASIC\text{-}ARCH\text{-}SPEC} \rhd A\varSigma}{\varGamma_s \vdash \mathtt{BASIC\text{-}ARCH\text{-}SPEC}\,\text{qua}\;\mathtt{ARCH\text{-}SPEC} \rhd A\varSigma}$$

$$\frac{\varGamma_s, \varGamma_m \vdash \mathtt{BASIC\text{-}ARCH\text{-}SPEC} \Rightarrow \mathcal{AM}}{\varGamma_s, \varGamma_m \vdash \mathtt{BASIC\text{-}ARCH\text{-}SPEC}\,\text{qua}\;\mathtt{ARCH\text{-}SPEC} \Rightarrow \mathcal{AM}}$$

$$\frac{ASN \in Dom(\mathcal{A}_s)}{(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \vdash ASN \text{ qua } \mathtt{ARCH\text{-}SPEC} \rhd \mathcal{A}_s(ASN)}$$

$$\frac{}{\varGamma_s, (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m) \vdash ASN \text{ qua } \mathtt{ARCH\text{-}SPEC} \Rightarrow \mathcal{A}_m(ASN)}$$

A basic architectural specification BASIC-ARCH-SPEC consists of a list of unit declarations and definitions together with a unit expression describing how such units are to be composed. A model of such an architectural specification consists of a unit for each unit declaration and definition in the list and the composition of these units as described by the result unit expression.

```
BASIC-ARCH-SPEC ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT
UNIT-DECL-DEFN  ::= UNIT-DECL | UNIT-DEFN
RESULT-UNIT     ::= result-unit UNIT-EXPRESSION
```

$$\boxed{\varGamma_s \vdash \mathtt{BASIC\text{-}ARCH\text{-}SPEC} \rhd A\varSigma \qquad\qquad \varGamma_s, \varGamma_m \vdash \mathtt{BASIC\text{-}ARCH\text{-}SPEC} \Rightarrow \mathcal{AM}}$$

$\varGamma_s$ and $\varGamma_m$ are compatible global environments; then $\mathcal{AM} \in \mathbf{ArchSpec}(A\varSigma)$.

$$\frac{\begin{array}{c}\varGamma_s \vdash \mathtt{UNIT\text{-}DECL\text{-}DEFN}^{+} \rhd C_s \\ \varGamma_s, C_s \vdash \mathtt{RESULT\text{-}UNIT} \rhd U\varSigma\end{array}}{\varGamma_s \vdash \mathtt{basic\text{-}arch\text{-}spec}\ \mathtt{UNIT\text{-}DECL\text{-}DEFN}^{+}\ \mathtt{RESULT\text{-}UNIT} \rhd (C_s, U\varSigma)}$$

$$\frac{\begin{array}{c}\varGamma_s \vdash \mathtt{UNIT\text{-}DECL\text{-}DEFN}^{+} \rhd C_s \\ \varGamma_s, \varGamma_m \vdash \mathtt{UNIT\text{-}DECL\text{-}DEFN}^{+} \Rightarrow C \\ \varGamma_s, \varGamma_m, C_s, C \vdash \mathtt{RESULT\text{-}UNIT} \Rightarrow UEv\end{array}}{\begin{array}{r}\varGamma_s, \varGamma_m \vdash \mathtt{basic\text{-}arch\text{-}spec}\ \mathtt{UNIT\text{-}DECL\text{-}DEFN}^{+}\ \mathtt{RESULT\text{-}UNIT} \Rightarrow \\ \{(E, UEv(E)) \mid E \in C\}\end{array}}$$

$$\boxed{\varGamma_s \vdash \mathtt{UNIT\text{-}DECL\text{-}DEFN}^{+} \rhd C_s \qquad\qquad \varGamma_s, \varGamma_m \vdash \mathtt{UNIT\text{-}DECL\text{-}DEFN}^{+} \Rightarrow C}$$

$\varGamma_s$ and $\varGamma_m$ are compatible global environments; then C is compatible with C_s too.

$$\frac{\begin{array}{c}\mathit{\Gamma}_s, C_s^{\emptyset} \vdash \texttt{UNIT-DECL-DEFN}_1 \rhd (C_s)_1 \\ \ldots \\ \mathit{\Gamma}_s, (C_s)_{n-1} \vdash \texttt{UNIT-DECL-DEFN}_n \rhd (C_s)_n\end{array}}{\mathit{\Gamma}_s \vdash \texttt{UNIT-DECL-DEFN}_1 \ldots \texttt{UNIT-DECL-DEFN}_n \rhd (C_s)_n}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}_s, C_s^{\emptyset} \vdash \texttt{UNIT-DECL-DEFN}_1 \rhd (C_s)_1 \\ \mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s^{\emptyset}, C^{\emptyset} \vdash \texttt{UNIT-DECL-DEFN}_1 \Rightarrow C_1 \\ \ldots \\ \mathit{\Gamma}_s, (C_s)_{n-1} \vdash \texttt{UNIT-DECL-DEFN}_n \rhd (C_s)_n \\ \mathit{\Gamma}_s, \mathit{\Gamma}_m, (C_s)_{n-1}, C_{n-1} \vdash \texttt{UNIT-DECL-DEFN}_n \Rightarrow C_n\end{array}}{\mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{UNIT-DECL-DEFN}_1 \ldots \texttt{UNIT-DECL-DEFN}_n \Rightarrow C_n}$$

$$\boxed{\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-DECL-DEFN} \rhd C_s' \qquad \mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-DECL-DEFN} \Rightarrow C'}$$

$\mathit{\Gamma}_s$ and $\mathit{\Gamma}_m$ are compatible global environments, and C is compatible with C_s; then C' and C_s' are compatible, $C_s \subseteq C_s'$ and $C \supseteq C'$.

$$\frac{\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-DECL} \rhd C_s'}{\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-DECL}\,\text{qua}\,\texttt{UNIT-DECL-DEFN} \rhd C_s \cup C_s'}$$

$$\frac{\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-DECL} \Rightarrow C'}{\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-DECL}\,\text{qua}\,\texttt{UNIT-DECL-DEFN} \Rightarrow C \cap C'}$$

$$\frac{\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-DEFN} \rhd C_s'}{\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-DEFN}\,\text{qua}\,\texttt{UNIT-DECL-DEFN} \rhd C_s \cup C_s'}$$

$$\frac{\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-DEFN} \Rightarrow C'}{\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-DEFN}\,\text{qua}\,\texttt{UNIT-DECL-DEFN} \Rightarrow C \cap C'}$$

$$\boxed{\mathit{\Gamma}_s, C_s \vdash \texttt{RESULT-UNIT} \rhd U\mathit{\Sigma} \qquad \mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{RESULT-UNIT} \Rightarrow \mathit{UEv}}$$

$\mathit{\Gamma}_s$ and $\mathit{\Gamma}_m$ are compatible global environments, and C is a unit context that is compatible with static unit context C_s; then $U\mathit{\Sigma}$ and $\mathit{UEv} \in \mathbf{UnitEval}(U\mathit{\Sigma})$ are compatible additions to C_s and C.

$$\frac{\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-EXPRESSION} \rhd U\mathit{\Sigma}}{\mathit{\Gamma}_s, C_s \vdash \texttt{result-unit UNIT-EXPRESSION} \rhd U\mathit{\Sigma}}$$

$$\frac{\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-EXPRESSION} \Rightarrow \mathit{UEv}}{\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{result-unit UNIT-EXPRESSION} \Rightarrow \mathit{UEv}}$$

5.3 Unit Declarations and Definitions

The visibility of unit names in architectural specifications is linear, and no unit name may be introduced more than once in a particular architectural specification. Declarations and definitions of units do not affect the global environment: a unit may be referenced only within the architectural specification in which it occurs.

5.3.1 Unit Declarations

A unit declaration `UNIT-DECL` provides a unit specification `UNIT-SPEC` and a unit name `UNIT-NAME`, which is used for referring to the unit in subsequent unit expressions.
In addition, the `UNIT-IMPORTED` lists any units that are imported for the implementation of the declared unit (corresponding to implementing a generic unit function and applying it only once, to the imported units). The unit specification `UNIT-SPEC` is treated as an extension of the signatures of the imported units; see Sect. 5.4.

```
UNIT-DECL     ::= unit-decl UNIT-NAME UNIT-SPEC UNIT-IMPORTED
UNIT-IMPORTED ::= unit-imported UNIT-TERM*
UNIT-NAME     ::= SIMPLE-ID
```

$$\boxed{\Gamma_s, C_s \vdash \texttt{UNIT-DECL} \rhd C_s' \qquad\qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-DECL} \Rightarrow C'}$$

Γ_s and Γ_m are compatible global environments, and C is a unit context that is compatible with static unit context C_s; then C_s' and C' are compatible extensions of C_s and C.

$$\frac{\begin{array}{c} C_s \vdash \texttt{UNIT-IMPORTED} \rhd \Sigma^I \\ \Sigma^I, \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd \Sigma \\ UN \notin Dom(C_s) \end{array}}{\Gamma_s, C_s \vdash \texttt{unit-decl}\ \ UN\ \ \texttt{UNIT-SPEC UNIT-IMPORTED} \rhd \{UN \mapsto (\Sigma^I \cup \Sigma)\}}$$

$$\frac{\begin{array}{c} C_s \vdash \texttt{UNIT-IMPORTED} \rhd \Sigma^I \\ \Sigma^I, \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd \overline{\Sigma} \rightarrow \Sigma_0 \\ UN \notin Dom(C_s) \end{array}}{\begin{array}{l} \Gamma_s, C_s \vdash \texttt{unit-decl}\ \ UN\ \ \texttt{UNIT-SPEC UNIT-IMPORTED} \rhd \\ \qquad\qquad\qquad\qquad\qquad\qquad \{UN \mapsto (\Sigma^I, \overline{\Sigma} \rightarrow \Sigma_0 \cup \Sigma^I)\} \end{array}}$$

We treat non-generic and generic unit specifications separately, as slightly different signature information is stored in static contexts in each case. However, the uniform notation adopted for import extensions allows us to capture the model semantics for both cases in a single rule.

$$\frac{\begin{array}{c} C_s, C \vdash \texttt{UNIT-IMPORTED} \Rightarrow MEv^I \\ \Sigma^I, \{MEv^I(E) \mid E \in C\}, \Gamma_m, \Gamma_s \vdash \texttt{UNIT-SPEC} \Rightarrow \mathcal{U} \end{array}}{\begin{array}{l} \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{unit-decl}\ \ UN\ \ \texttt{UNIT-SPEC UNIT-IMPORTED} \Rightarrow \\ \qquad\qquad\qquad\qquad\qquad\qquad C^{\emptyset}[UN/(\mathcal{U} \oplus MEv^I)] \end{array}}$$

$$\boxed{C_s \vdash \texttt{UNIT-IMPORTED} \rhd \Sigma \qquad\qquad C_s, C \vdash \texttt{UNIT-IMPORTED} \Rightarrow MEv}$$

C is a unit context that is compatible with static unit context C_s; then Σ and MEv are compatible additions to C_s and C.

$$\frac{C_s \vdash \texttt{UNIT-TERM}_1 \rhd \Sigma_1 \quad \cdots \quad C_s \vdash \texttt{UNIT-TERM}_k \rhd \Sigma_k}{C_s \vdash \texttt{unit-imported UNIT-TERM}_1\texttt{,...,UNIT-TERM}_k \rhd \Sigma_1 \cup \ldots \cup \Sigma_k}$$

By definition, the union of the empty family of signatures (considered here if $k = 0$) is the empty signature.

$$\frac{\begin{array}{c} C_s, C \vdash \texttt{UNIT-TERM}_1 \Rightarrow MEv_1 \quad \cdots \quad C_s, C \vdash \texttt{UNIT-TERM}_k \Rightarrow MEv_k \\ \text{for each } E \in C, MEv_1(E), \ldots, MEv_k(E) \text{ are compatible} \end{array}}{\begin{array}{l} C_s, C \vdash \texttt{unit-imported UNIT-TERM}_1\texttt{,...,UNIT-TERM}_k \Rightarrow \\ \qquad\qquad\qquad\qquad \lambda E \in C \cdot MEv_1(E) \oplus \ldots \oplus MEv_k(E) \end{array}}$$

5.3.2 Unit Definitions

A unit definition `UNIT-DEFN` defines the name `UNIT-NAME` to refer to the unit resulting from the composition described by the unit expression `UNIT-EXPRESSION`.

```
UNIT-DEFN ::= unit-defn UNIT-NAME UNIT-EXPRESSION
```

$$\boxed{\Gamma_s, C_s \vdash \texttt{UNIT-DEFN} \rhd C_s' \qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-DEFN} \Rightarrow C'}$$

Γ_s and Γ_m are compatible global environments, and C is a unit context that is compatible with static unit context C_s; then C_s' and C' are compatible extensions of C_s and C, and moreover, if two unit environments in $C \cap C'$ coincide on the unit names outside the domain of C_s' then they are in fact equal.

$$\frac{\begin{array}{c} \Gamma_s, C_s \vdash \texttt{UNIT-EXPRESSION} \rhd \Sigma \\ UN \notin Dom(C_s) \end{array}}{\Gamma_s, C_s \vdash \texttt{unit-defn } UN \texttt{ UNIT-EXPRESSION} \rhd \{UN \mapsto \Sigma\}}$$

$$\frac{\begin{array}{c} \Gamma_s, C_s \vdash \texttt{UNIT-EXPRESSION} \rhd \overline{\Sigma} \rightarrow \Sigma \\ UN \notin Dom(C_s) \end{array}}{\Gamma_s, C_s \vdash \texttt{unit-defn } UN \texttt{ UNIT-EXPRESSION} \rhd \{UN \mapsto (\emptyset, \overline{\Sigma} \rightarrow \Sigma)\}}$$

$\emptyset$ is the empty signature here: no imports are given to defined units.

The above rules prevent unit names from being re-introduced in local unit definitions (in agreement with the requirement that each unit name is introduced only once in a given architectural specification). However, according to the usual visibility rules, the same unit name may be used for local unit definitions that are not in the scope of one another.

$$\frac{\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-EXPRESSION} \Rightarrow UEv}{\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{unit-defn } UN \texttt{ UNIT-EXPRESSION} \Rightarrow C^{\emptyset}[UN/UEv]}$$

5.4 Unit Specifications

A unit specification definition `UNIT-SPEC-DEFN` defines the name `SPEC-NAME` to refer to the unit specification `UNIT-SPEC`, extending the global environment (which must not already include a definition for `SPEC-NAME`). The local environment given to `UNIT-SPEC` is empty, i.e., the unit specification is implicitly closed.

```
UNIT-SPEC-DEFN   ::= unit-spec-defn SPEC-NAME UNIT-SPEC
```

$$\boxed{\Gamma_s \vdash \texttt{UNIT-SPEC-DEFN} \rhd \Gamma'_s \qquad\qquad \Gamma_s, \Gamma_m \vdash \texttt{UNIT-SPEC-DEFN} \rhd \Gamma'_m}$$

Γ_s and Γ_m are compatible global environments; then Γ'_s and Γ'_m are compatible as well, and extend Γ_s and Γ_m, respectively.

$$\frac{\begin{array}{c}\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\\ \mathit{USN} \notin \mathit{Dom}(\mathcal{G}_s) \cup \mathit{Dom}(\mathcal{V}_s) \cup \mathit{Dom}(\mathcal{A}_s) \cup \mathit{Dom}(\mathcal{T}_s)\\ \emptyset, \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd U\Sigma\end{array}}{\Gamma_s \vdash \texttt{unit-spec-defn}\ \ \mathit{USN}\ \ \texttt{UNIT-SPEC} \rhd (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s \cup \{\mathit{USN} \mapsto U\Sigma\})}$$

$$\frac{\begin{array}{c}\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)\\ \emptyset, \mathbf{Mod}(\emptyset), \Gamma_s, \Gamma_m \vdash \texttt{UNIT-SPEC} \Rightarrow \mathcal{U}\end{array}}{\begin{array}{l}\Gamma_s, \Gamma_m \vdash \texttt{unit-spec-defn}\ \ \mathit{USN}\ \ \texttt{UNIT-SPEC} \Rightarrow\\ \qquad\qquad\qquad\qquad\qquad (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m \cup \{\mathit{USN} \mapsto \mathcal{U}\})\end{array}}$$

$\emptyset$ is the empty signature here, and $\mathbf{Mod}(\emptyset)$ is the (singleton) class of models over this signature. This captures the fact that `UNIT-SPEC` is being closed, so no dependencies on imports and hence on the unit environment can occur.

A unit specification may be a unit type, the name of another unit specification, an architectural specification (either a reference to the defined name of an architectural specification, or an anonymous architectural specification), or an explicitly-closed unit specification. In unit declarations, unit specifications are used as extensions of imported units, see Sect. 5.3.1.

```
UNIT-SPEC        ::= UNIT-TYPE | SPEC-NAME | ARCH-UNIT-SPEC
                   | CLOSED-UNIT-SPEC
```

$$\boxed{\Sigma, \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd U\Sigma \qquad\qquad \Sigma^I, \mathcal{M}^I, \Gamma_s, \Gamma_m \vdash \texttt{UNIT-SPEC} \Rightarrow \mathcal{U}}$$

Γ_s and Γ_m are compatible global environments, Σ^I is a signature and $\mathcal{M}^I \subseteq \mathbf{Mod}(\Sigma^I)$; then $U\Sigma$ is a unit signature and $\mathcal{U} \in \mathbf{UnitSpec}(U\Sigma)$.

$$\frac{\Sigma^I, \Gamma_s \vdash \texttt{UNIT-TYPE} \rhd U\Sigma}{\Sigma^I, \Gamma_s \vdash \texttt{UNIT-TYPE}\,\text{qua}\,\texttt{UNIT-SPEC} \rhd U\Sigma}$$

$$\frac{\Sigma^I, \mathcal{M}^I, \Gamma_s, \Gamma_m \vdash \texttt{UNIT-TYPE} \Rightarrow \mathcal{U}}{\Sigma^I, \mathcal{M}^I, \Gamma_s, \Gamma_m \vdash \texttt{UNIT-TYPE}\,\text{qua}\,\texttt{UNIT-SPEC} \Rightarrow \mathcal{U}}$$

$$\frac{USN \in Dom(\mathcal{T}_s)}{\Sigma^I, (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \vdash USN\,\text{qua}\,\texttt{UNIT-SPEC} \rhd \mathcal{T}_s(USN)}$$

$$\frac{}{\Sigma^I, \mathcal{M}^I, \Gamma_s, (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m) \vdash USN\,\text{qua}\,\texttt{UNIT-SPEC} \Rightarrow \mathcal{T}_m(USN)}$$

$$\frac{\Gamma_s \vdash \texttt{ARCH-UNIT-SPEC} \rhd U\Sigma}{\Sigma^I, \Gamma_s \vdash \texttt{ARCH-UNIT-SPEC}\,\text{qua}\,\texttt{UNIT-SPEC} \rhd U\Sigma}$$

$$\frac{\Gamma_s, \Gamma_m \vdash \texttt{ARCH-UNIT-SPEC} \Rightarrow \mathcal{U}}{\Sigma^I, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{ARCH-UNIT-SPEC}\,\text{qua}\,\texttt{UNIT-SPEC} \Rightarrow \mathcal{U}}$$

$$\frac{\Gamma_s \vdash \texttt{CLOSED-UNIT-SPEC} \rhd U\Sigma}{\Sigma^I, \Gamma_s \vdash \texttt{CLOSED-UNIT-SPEC}\,\text{qua}\,\texttt{UNIT-SPEC} \rhd U\Sigma}$$

$$\frac{\Gamma_s, \Gamma_m \vdash \texttt{CLOSED-UNIT-SPEC} \Rightarrow \mathcal{U}}{\Sigma^I, \mathcal{M}^I, \Gamma_s, \Gamma_m \vdash \texttt{CLOSED-UNIT-SPEC}\,\text{qua}\,\texttt{UNIT-SPEC} \Rightarrow \mathcal{U}}$$

5.4.1 Unit Types

A unit type lists argument specifications and the result specification. A unit satisfies a unit type when it is a persistent function that maps compatible tuples of models of the argument specifications to models of their extension by the result specification.

```
UNIT-TYPE ::= unit-type SPEC* SPEC
```

$$\boxed{\Sigma, \Gamma_s \vdash \texttt{UNIT-TYPE} \rhd U\Sigma \qquad\qquad \Sigma^I, \mathcal{M}^I, \Gamma_s, \Gamma_m \vdash \texttt{UNIT-TYPE} \Rightarrow \mathcal{U}}$$

Γ_s and Γ_m are compatible global environments, Σ^I is a signature and $\mathcal{M}^I \subseteq$ **Mod**(Σ^I); then $U\Sigma$ is a unit signature and $\mathcal{U} \in$ **UnitSpec**$(U\Sigma)$.

$$\frac{\Sigma^I, \Gamma_s \vdash \texttt{SPEC} \rhd \Sigma}{\Sigma^I, \Gamma_s \vdash \texttt{unit-type SPEC} \rhd \Sigma}$$

$$\frac{\Sigma^I, \mathcal{M}^I, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}}{\Sigma^I, \mathcal{M}^I, \Gamma_s, \Gamma_m \vdash \texttt{unit-type SPEC} \Rightarrow \mathcal{M}}$$

$$\frac{\begin{array}{c}\emptyset, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma_1 \quad \cdots \quad \emptyset, \Gamma_s \vdash \mathtt{SPEC}_n \rhd \Sigma_n \\ \Sigma^I \cup \Sigma_1 \cup \ldots \cup \Sigma_n, \Gamma_s \vdash \mathtt{SPEC} \rhd \Sigma\end{array}}{\Sigma^I, \Gamma_s \vdash \texttt{unit-type}\ \mathtt{SPEC}_1\texttt{,...,}\mathtt{SPEC}_n\ \ \mathtt{SPEC} \rhd \Sigma_1, \ldots, \Sigma_n \to \Sigma}$$

$$\frac{\begin{array}{c}\emptyset, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma_1 \quad \cdots \quad \emptyset, \Gamma_s \vdash \mathtt{SPEC}_n \rhd \Sigma_n \\ \emptyset, \mathbf{Mod}(\emptyset), \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_1 \Rightarrow \mathcal{M}_1 \ \cdots \ \emptyset, \mathbf{Mod}(\emptyset), \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_n \Rightarrow \mathcal{M}_n \\ \Sigma^I \cup \Sigma_1 \cup \ldots \cup \Sigma_n, \Gamma_s \vdash \mathtt{SPEC} \rhd \Sigma \\ \mathcal{M}_0 = \mathcal{M}^I \oplus \mathcal{M}_1 \oplus \ldots \oplus \mathcal{M}_n \\ \Sigma^I \cup \Sigma_1 \cup \ldots \cup \Sigma_n, \mathcal{M}_0, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC} \Rightarrow \mathcal{M}\end{array}}{\begin{array}{l}\Sigma^I, \mathcal{M}^I, \Gamma_s, \Gamma_m \vdash \texttt{unit-type}\ \mathtt{SPEC}_1\texttt{,...,}\mathtt{SPEC}_n\ \ \mathtt{SPEC} \Rightarrow \\ \quad \{F \in \mathbf{Unit}(\Sigma_1, \ldots, \Sigma_n \to \Sigma) \mid \text{for some } M^I \in \mathcal{M}^I, \\ \qquad\quad \text{for all } M \in \mathcal{M}_0 \text{ with } M|_{\Sigma^I} = M^I, \langle M|_{\Sigma_1}, \ldots, M|_{\Sigma_n}\rangle \in Dom(F), \\ \qquad\qquad\quad F\langle M|_{\Sigma_1}, \ldots, M|_{\Sigma_n}\rangle \in \mathcal{M}, \text{ and } (F\langle M|_{\Sigma_1}, \ldots, M|_{\Sigma_n}\rangle)|_{\Sigma^I} = M^I\}\end{array}}$$

Imports are *not* taken as local environment for parameter specifications here, in accordance with the interpretation of imports as 'already instantiated parameters'.

The semantics of SPEC is given in Sect. 4.2.

5.4.2 Architectural Unit Specifications

A unit satisfies ARCH-UNIT-SPEC when it is the result unit of some model of the architectural specification ARCH-SPEC.

```
ARCH-UNIT-SPEC   ::= arch-unit-spec ARCH-SPEC
```

$$\boxed{\Gamma_s \vdash \mathtt{ARCH\text{-}UNIT\text{-}SPEC} \rhd U\Sigma \qquad\qquad \Gamma_s, \Gamma_m \vdash \mathtt{ARCH\text{-}UNIT\text{-}SPEC} \Rightarrow \mathcal{U}}$$

Γ_s and Γ_m are compatible global environments; then $U\Sigma$ is a unit signature and $\mathcal{U} \in \mathbf{UnitSpec}(U\Sigma)$.

$$\frac{\Gamma_s \vdash \mathtt{ARCH\text{-}SPEC} \rhd (C_s, U\Sigma)}{\Gamma_s \vdash \mathtt{ARCH\text{-}SPEC}\,\text{qua}\,\mathtt{ARCH\text{-}UNIT\text{-}SPEC} \rhd U\Sigma}$$

$$\frac{\Gamma_s, \Gamma_m \vdash \mathtt{ARCH\text{-}SPEC} \Rightarrow \mathcal{AM}}{\Gamma_s, \Gamma_m \vdash \mathtt{ARCH\text{-}SPEC}\,\text{qua}\,\mathtt{ARCH\text{-}UNIT\text{-}SPEC} \Rightarrow \{U \mid (E, U) \in \mathcal{AM} \text{ for some } E\}}$$

5.4.3 Closed Unit Specifications

A closed unit specification CLOSED-UNIT-SPEC determines the same type as UNIT-SPEC determines in the empty local environment.

```
CLOSED-UNIT-SPEC ::= closed-unit-spec UNIT-SPEC
```

$$\boxed{\Gamma_s \vdash \texttt{CLOSED-UNIT-SPEC} \rhd U\Sigma \qquad\qquad \Gamma_s, \Gamma_m \vdash \texttt{CLOSED-UNIT-SPEC} \Rightarrow \mathcal{U}}$$

Γ_s and Γ_m are compatible global environments; then $U\Sigma$ is a unit signature and $\mathcal{U} \in \mathbf{UnitSpec}(U\Sigma)$.

$$\frac{\emptyset, \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd U\Sigma}{\Gamma_s \vdash \texttt{closed UNIT-SPEC} \rhd U\Sigma}$$

$$\frac{\emptyset, \mathbf{Mod}(\emptyset), \Gamma_s, \Gamma_m \vdash \texttt{UNIT-SPEC} \Rightarrow \mathcal{U}}{\Gamma_s, \Gamma_m \vdash \texttt{closed UNIT-SPEC} \Rightarrow \mathcal{U}}$$

5.5 Unit Expressions

A unit expression (with some unit bindings) describes a composition of units declared (or defined) in the enclosing architectural specification. The result unit is a function, mapping the arguments specified by the unit bindings (if any) to the unit described by the unit term UNIT-TERM.

```
UNIT-EXPRESSION ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING    ::= unit-binding UNIT-NAME UNIT-SPEC
```

$$\boxed{\begin{array}{l}\Gamma_s, C_s \vdash \texttt{UNIT-EXPRESSION} \rhd U\Sigma \\ \qquad\qquad\qquad\qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-EXPRESSION} \Rightarrow UEv\end{array}}$$

Γ_s and Γ_m are compatible global environments, and C is a unit context that is compatible with static unit context C_s; then $U\Sigma$ and UEv are compatible additions to C_s and C.

$$\frac{\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \rhd \Sigma}{\Gamma_s, C_s \vdash \texttt{unit-expression UNIT-TERM} \rhd \Sigma}$$

$$\frac{\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow MEv}{\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{unit-expression UNIT-TERM} \Rightarrow MEv}$$

$$\frac{\begin{array}{c}\Gamma_s \vdash \texttt{UNIT-BINDING}_1 \rhd (UN_1, \Sigma_1) \quad \cdots \quad \Gamma_s \vdash \texttt{UNIT-BINDING}_n \rhd (UN_n, \Sigma_n) \\ UN_1, \ldots, UN_n \text{ are distinct} \qquad \{UN_1, \ldots, UN_n\} \cap Dom(C_s) = \emptyset \\ C'_s = \{UN_1 \mapsto \Sigma_1, \ldots, UN_n \mapsto \Sigma_n\} \\ \Gamma_s, C_s \cup C'_s \vdash \texttt{UNIT-TERM} \rhd \Sigma \qquad \Sigma_1 \cup \ldots \cup \Sigma_n \subseteq \Sigma\end{array}}{\begin{array}{r}\Gamma_s, C_s \vdash \texttt{unit-expression}\, \texttt{UNIT-BINDING}_1\texttt{,...,}\texttt{UNIT-BINDING}_n \ \texttt{UNIT-TERM} \\ \rhd \Sigma_1, \ldots, \Sigma_n {\rightarrow} \Sigma\end{array}}$$

$$\cfrac{\begin{array}{c}\Gamma_s \vdash \texttt{UNIT-BINDING}_1 \rhd (UN_1, \Sigma_1) \quad \cdots \quad \Gamma_s \vdash \texttt{UNIT-BINDING}_n \rhd (UN_n, \Sigma_n) \\ \Gamma_s, \Gamma_m \vdash \texttt{UNIT-BINDING}_1 \Rightarrow (UN_1, \mathcal{M}_1) \\ \cdots \\ \Gamma_s, \Gamma_m \vdash \texttt{UNIT-BINDING}_n \Rightarrow (UN_n, \mathcal{M}_n) \\ \Sigma^P = \Sigma_1 \cup \ldots \cup \Sigma_n \qquad \mathcal{M}^P = \mathcal{M}_1 \oplus \ldots \oplus \mathcal{M}_n \\ UN^P \notin Dom(C_s) \cup \{UN_1, \ldots, UN_n\} \\ C'_s = \{UN^P \mapsto \Sigma^P, UN_1 \mapsto \Sigma_1, \ldots, UN_n \mapsto \Sigma_n\} \\ C' = C^{\emptyset}[UN^P/\mathcal{M}^P][UN_1/(\lambda E \cdot E(UN^P)|_{\Sigma_1})] \ldots [UN_n/(\lambda E \cdot E(UN^P)|_{\Sigma_n})] \\ \Gamma_s, \Gamma_m, C_s \cup C'_s, C \cap C' \vdash \texttt{UNIT-TERM} \Rightarrow MEv \\ \text{for all } E \in C \cap C', MEv(E)|_{\Sigma^P} = E(UN^P) \\ \begin{array}{l} UEv \in \mathbf{UnitEval}(\Sigma_1, \ldots, \Sigma_n \rightarrow \Sigma) \text{ is such that } Dom(UEv) = C \text{ and} \\ \text{for } E \in C, \\ \quad \langle M_1, \ldots, M_n \rangle \in Dom(UEv(E)) \subseteq \mathbf{CompMod}(\Sigma_1, \ldots, \Sigma_n) \text{ iff} \\ \quad\; E + \{UN^P \mapsto (M_1 \oplus \ldots \oplus M_n), UN_1 \mapsto M_1, \ldots, UN_n \mapsto M_n\} \in C \cap C' \\ \quad \text{and then for } \langle M_1, \ldots, M_n \rangle \in Dom(UEv(E)), \\ \quad UEv(E)\langle M_1, \ldots, M_n \rangle = \\ \qquad MEv(E + \{UN^P \mapsto (M_1 \oplus \ldots \oplus M_n), UN_1 \mapsto M_1, \ldots, UN_n \mapsto M_n\}) \end{array}\end{array}}{\begin{array}{l}\Gamma_s, \Gamma_m, C_s, C \vdash \\ \quad \texttt{unit-expression UNIT-BINDING}_1\texttt{,...,UNIT-BINDING}_n\texttt{ UNIT-TERM} \\ \hfill \Rightarrow UEv\end{array}}$$

The trick with the use of a new unit name UN^P restricts attention to environments with formal parameter names denoting compatible models only.

$$\boxed{\Gamma_s \vdash \texttt{UNIT-BINDING} \rhd (UN, \Sigma) \qquad \Gamma_s, \Gamma_m \vdash \texttt{UNIT-BINDING} \Rightarrow (UN, \mathcal{M})}$$

Γ_s and Γ_m are compatible global environments; then UN is a unit name (the same for the static and model semantics) and $\mathcal{M} \subseteq \mathbf{Mod}(\Sigma)$.

$$\frac{\emptyset, \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd \Sigma}{\Gamma_s \vdash \texttt{unit-binding } UN \texttt{ UNIT-SPEC} \rhd (UN, \Sigma)}$$

The above rule imposes the restriction that only non-generic units can be bound in unit bindings.

$$\frac{\emptyset, \mathbf{Mod}(\emptyset), \Gamma_s, \Gamma_m \vdash \texttt{UNIT-SPEC} \Rightarrow \mathcal{M}}{\Gamma_s, \Gamma_m \vdash \texttt{unit-binding } UN \texttt{ UNIT-SPEC} \Rightarrow (UN, \mathcal{M})}$$

5.5.1 Unit Terms

Unit terms provide counterparts to most of the constructs of structured specifications: translations, reductions, amalgamations (corresponding to unions), local unit definitions, and applications (corresponding to instantiations) – but with a crucially different semantics. For units, enough sharing is required so that the constructs as applied to the given units will always produce results. Sharing between symbols is understood here semantically: two symbols share if they coincide semantically. See Sect. 5.6 for further elaboration on this issue.

Taking the unit type of each unit name from its declaration, the unit term must be well-typed. All the constructs involved must get argument units over the appropriate signatures.

```
UNIT-TERM ::= UNIT-REDUCTION | UNIT-TRANSLATION | AMALGAMATION
            | LOCAL-UNIT | UNIT-APPL
```

$\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \rhd \Sigma \qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow \mathit{MEv}$

Γ_s and Γ_m are compatible global environments; C is a unit context that is compatible with static unit context C_s; then Σ and MEv are compatible additions to C_s and C.

Rules elided.

Due to the semantic verification condition in the model semantics for applications, in general we may not be able to derive $C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow \mathit{MEv}$ even if the static semantics works ($C_s \vdash \texttt{UNIT-TERM} \rhd \Sigma$ can be derived).

The only reason to keep the global environments available for the semantic of unit terms is that they are needed for the semantics of unit expressions in local definitions (Sect. 5.5.1).

Unit Translations

A unit translation allows some of the unit symbols to be renamed. Any symbols that happen to be glued together by the renaming must share.

```
UNIT-TRANSLATION ::= unit-translation UNIT-TERM RENAMING
```

$\Gamma_s, C_s \vdash \texttt{UNIT-TRANSLATION} \rhd \Sigma$

$\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TRANSLATION} \Rightarrow \mathit{MEv}$

Γ_s and Γ_m are compatible global environments; C is a unit context that is compatible with static unit context C_s; then Σ and MEv are compatible additions to C_s and C.

$$\frac{\begin{array}{c}\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-TERM} \rhd \mathit{\Sigma} \\ \mathit{\Sigma} \vdash \texttt{RENAMING} \rhd \sigma{:}\mathit{\Sigma}{\to}\mathit{\Sigma}'\end{array}}{\mathit{\Gamma}_s, C_s \vdash \texttt{unit-translation UNIT-TERM RENAMING} \rhd \mathit{\Sigma}'}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-TERM} \rhd \mathit{\Sigma} \\ \mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow \mathit{MEv} \\ \mathit{\Sigma} \vdash \texttt{RENAMING} \rhd \sigma{:}\mathit{\Sigma}{\to}\mathit{\Sigma}' \\ \text{for all } E \in C,\ \text{there exists a unique } M' \in \mathbf{Mod}(\mathit{\Sigma}') \text{ with } M'|_\sigma = \mathit{MEv}(E)\end{array}}{\begin{array}{l}\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{unit-translation UNIT-TERM RENAMING} \Rightarrow \\ \qquad\qquad \{E \mapsto M' \mid E \in C, M' \in \mathbf{Mod}(\mathit{\Sigma}'), M'|_\sigma = \mathit{MEv}(E)\}\end{array}}$$

The semantics of RENAMING is given in Sect. 4.2.1.

Unit Reductions

A unit-reduction allows parts of the unit to be hidden and other parts to be simultaneously renamed.

```
UNIT-REDUCTION ::= unit-reduction UNIT-TERM RESTRICTION
```

$$\boxed{\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-REDUCTION} \rhd \mathit{\Sigma} \quad \mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-REDUCTION} \Rightarrow \mathit{MEv}}$$

$\mathit{\Gamma}_s$ and $\mathit{\Gamma}_m$ are compatible global environments; C is a unit context that is compatible with static unit context C_s; then $\mathit{\Sigma}$ and MEv are compatible additions to C_s and C.

$$\frac{\begin{array}{c}\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-TERM} \rhd \mathit{\Sigma} \\ (\emptyset, \mathit{\Sigma}) \vdash \texttt{RESTRICTION} \rhd \sigma{:}\mathit{\Sigma}'{\to}\mathit{\Sigma}''\end{array}}{\mathit{\Gamma}_s, C_s \vdash \texttt{unit-reduction UNIT-TERM RESTRICTION} \rhd \mathit{\Sigma}''}$$

$$\frac{\begin{array}{c}\mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-TERM} \rhd \mathit{\Sigma} \\ \mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow \mathit{MEv} \\ (\emptyset, \mathit{\Sigma}) \vdash \texttt{RESTRICTION} \rhd \sigma{:}\mathit{\Sigma}'{\to}\mathit{\Sigma}'' \\ \begin{array}{l}\text{for all } E \in C, \\ \qquad \text{there exists a unique } M'' \in \mathbf{Mod}(\mathit{\Sigma}'') \text{ with } M''|_\sigma = \mathit{MEv}(E)|_{\mathit{\Sigma}'}\end{array}\end{array}}{\begin{array}{l}\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{unit-reduction UNIT-TERM RESTRICTION} \Rightarrow \\ \qquad\qquad \{E \mapsto M'' \mid E \in C, M'' \in \mathbf{Mod}(\mathit{\Sigma}''), M''|_\sigma = \mathit{MEv}(E)|_{\mathit{\Sigma}'}\}\end{array}}$$

The semantics for RESTRICTION is given in Sect. 4.2.2.

Amalgamations

An amalgamation produces a unit that consists of the components of all the amalgamated units put together. Compatibility of the unit terms must be ensured.

```
AMALGAMATION ::= amalgamation UNIT-TERM+
```

$$\boxed{\Gamma_s, C_s \vdash \texttt{AMALGAMATION} \rhd \Sigma \qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{AMALGAMATION} \Rightarrow \mathit{MEv}}$$

Γ_s and Γ_m are compatible global environments; C is a unit context that is compatible with static unit context C_s; then Σ and MEv are compatible additions to C_s and C.

$$\frac{\Gamma_s, C_s \vdash \texttt{UNIT-TERM}_1 \rhd \Sigma_1 \quad \cdots \quad \Gamma_s, C_s \vdash \texttt{UNIT-TERM}_n \rhd \Sigma_n}{\Gamma_s, C_s \vdash \texttt{amalgamation UNIT-TERM}_1\texttt{,...,UNIT-TERM}_n \rhd \Sigma_1 \cup \ldots \cup \Sigma_n}$$

$$\frac{\begin{array}{c}\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM}_1 \Rightarrow \mathit{MEv}_1 \\ \ldots \\ \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM}_n \Rightarrow \mathit{MEv}_n \\ \text{for all } E \in C, \mathit{MEv}_1(E), \ldots, \mathit{MEv}_n(E) \text{ are compatible}\end{array}}{\begin{array}{l}\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{amalgamation UNIT-TERM}_1\texttt{,...,UNIT-TERM}_n \Rightarrow \\ \qquad\qquad\qquad\qquad \lambda E \in C \cdot \mathit{MEv}_1(E) \oplus \ldots \oplus \mathit{MEv}_k(E)\end{array}}$$

Local Units

This construct allows for naming units that are locally defined for use in a unit term, these units being intermediate results that are not to be visible in the models of the enclosing architectural specification.

```
LOCAL-UNIT ::= local-unit UNIT-DEFN+ UNIT-TERM
```

$$\boxed{\Gamma_s, C_s \vdash \texttt{LOCAL-UNIT} \rhd \Sigma \qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{LOCAL-UNIT} \Rightarrow \mathit{MEv}}$$

Γ_s and Γ_m are compatible global environments; C is a unit context that is compatible with static unit context C_s; then Σ and MEv are compatible additions to C_s and C.

$$\frac{\begin{array}{c}\Gamma_s, C_s \vdash \texttt{UNIT-DEFN}_1 \rhd (C_s)_1 \\ \ldots \\ \Gamma_s, C_s \cup (C_s)_1 \cup \ldots \cup (C_s)_{n-1} \vdash \texttt{UNIT-DEFN}_n \rhd (C_s)_n \\ \Gamma_s, C_s \cup (C_s)_1 \cup \ldots \cup (C_s)_n \vdash \texttt{UNIT-TERM} \rhd \Sigma\end{array}}{\Gamma_s, C_s \vdash \texttt{local-unit UNIT-DEFN}_1\texttt{,...,UNIT-DEFN}_n\ \texttt{UNIT-TERM} \rhd \Sigma}$$

$$\frac{\begin{array}{c}\Gamma_s, C_s \vdash \mathtt{UNIT\text{-}DEFN}_1 \rhd (C_s)_1\\ \Gamma_s, \Gamma_m, C_s, C \vdash \mathtt{UNIT\text{-}DEFN}_1 \Rightarrow C_1\\ \dots\\ \Gamma_s, C_s \cup (C_s)_1 \cup \ldots \cup (C_s)_{n-1} \vdash \mathtt{UNIT\text{-}DEFN}_n \rhd (C_s)_n\\ \Gamma_s, \Gamma_m, C_s \cup (C_s)_1 \cup \ldots \cup (C_s)_{n-1}, C \cap C_1 \cap \ldots \cap C_{n-1} \vdash\\ \qquad\qquad\qquad\qquad \mathtt{UNIT\text{-}TERM}_n \Rightarrow C_n\\ \Gamma_s, \Gamma_m, C_s \cup (C_s)_1 \cup \ldots \cup (C_s)_n, C \cap C_1 \cap \ldots \cap C_n \vdash \mathtt{UNIT\text{-}TERM} \Rightarrow \mathit{MEv}\\ \mathit{MEv}' = \{E \mapsto \mathit{MEv}(E + E^L) \mid\\ \qquad E \in C, \mathit{Dom}(E^L) = \mathit{Dom}(C_s^L), (E + E^L) \in C \cap C^L\}\end{array}}{\begin{array}{l}\Gamma_s, \Gamma_m, C_s, C \vdash\\ \qquad \mathtt{local\text{-}unit}\ \mathtt{UNIT\text{-}DEFN}_1\mathtt{,\ldots,}\mathtt{UNIT\text{-}DEFN}_n\ \mathtt{UNIT\text{-}TERM} \Rightarrow \mathit{MEv}'\end{array}}$$

Notice that by the semantic properties of unit definitions, for each E there is at most one E^L with $\mathit{Dom}(E^L) = \mathit{Dom}(C_s^L)$ such that $(E + E^L) \in C \cap C^L$.

Unit Applications

A unit application `UNIT-APPL` refers to a generic unit named `UNIT-NAME` that has already been declared or defined in the enclosing architectural specification, providing a fitting argument for each declared parameter. The fitting argument fits the argument unit given by the unit term to the corresponding formal argument for the generic unit via a signature morphism determined by the symbol mapping. The signature morphism is obtained in the same way as for generic specifications. Each fitting argument unit is required to be a model of the corresponding argument specification.

```
UNIT-APPL ::= unit-appl UNIT-NAME FIT-ARG-UNIT*
FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*
```

$$\boxed{\Gamma_s, C_s \vdash \mathtt{UNIT\text{-}APPL} \rhd \Sigma \qquad\qquad \Gamma_s, \Gamma_m, C_s, C \vdash \mathtt{UNIT\text{-}APPL} \Rightarrow \mathit{MEv}}$$

Γ_s and Γ_m are compatible global environments; C is a unit context that is compatible with static unit context C_s; then Σ and MEv are compatible additions to C_s and C.

$$\frac{C_s(\mathit{UN}) = \Sigma}{\Gamma_s, C_s \vdash \mathtt{unit\text{-}appl}\ \ \mathit{UN} \rhd \Sigma}$$

$$\frac{}{\Gamma_s, \Gamma_m, C_s, C \vdash \mathtt{unit\text{-}appl}\ \ \mathit{UN} \Rightarrow \{E \mapsto E(\mathit{UN}) \mid \mathit{UN} \in \mathit{Dom}(E)\}}$$

$$\begin{array}{c}
C_s(UN) = (\Sigma^I, (\Sigma_1, \ldots, \Sigma_n \to \Sigma)) \\
\Sigma^F = \Sigma^I \cup \Sigma_1 \cup \ldots \cup \Sigma_n \\
\Sigma_1, \Gamma_s, C_s \vdash \texttt{FIT-ARG-UNIT}_1 \rhd \sigma_1{:}\Sigma_1 \to \Sigma_1^A \\
\ldots \\
\Sigma_n, \Gamma_s, C_s \vdash \texttt{FIT-ARG-UNIT}_n \rhd \sigma_n{:}\Sigma_n \to \Sigma_n^A \\
\Sigma^A = \Sigma^I \cup \Sigma_1^A \cup \ldots \cup \Sigma_n^A \qquad \sigma^A = (id_{\Sigma^I} \cup \sigma_1 \cup \ldots \cup \sigma_n) \colon \Sigma^F \to \Sigma^A \\
\sigma^A(\Delta) \colon \Sigma \to (\Sigma^A \cup \Sigma^A(\Delta)), \text{ where } \Delta{:}\Sigma^F \to \Sigma \text{ is the signature extension} \\
\hline
\Gamma_s, C_s \vdash \texttt{unit-appl}\ UN\ \texttt{FIT-ARG-UNIT}_1\texttt{,...,FIT-ARG-UNIT}_n \rhd \\
\Sigma^A \cup \Sigma^A(\Delta)
\end{array}$$

See Sect. 4.1.3 for the definition of $\sigma^A(\Delta)$, the extension of a signature morphism along a signature extension.

$$\begin{array}{c}
C_s(UN) = (\Sigma^I, (\Sigma_1, \ldots, \Sigma_n \to \Sigma)) \\
\Sigma^F = \Sigma^I \cup \Sigma_1 \cup \ldots \cup \Sigma_n \\
\Sigma_1, \Gamma_s, C_s \vdash \texttt{FIT-ARG-UNIT}_1 \rhd \sigma_1{:}\Sigma_1 \to \Sigma_1^A \\
\Sigma_1, \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{FIT-ARG-UNIT}_1 \Rightarrow MEv_1 \\
\ldots \\
\Sigma_n, \Gamma_s, C_s \vdash \texttt{FIT-ARG-UNIT}_n \rhd \sigma_n{:}\Sigma_n \to \Sigma_n^A \\
\Sigma_n, \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{FIT-ARG-UNIT}_n \Rightarrow MEv_n \\
\text{for all } E \in C, \langle MEv_1(E)|_{\sigma_1}, \ldots, MEv_n(E)|_{\sigma_n} \rangle \in Dom(E(UN)) \\
\Sigma^A = \Sigma^I \cup \Sigma_1^A \cup \ldots \cup \Sigma_n^A \qquad \sigma^A = (id_{\Sigma^I} \cup \sigma_1 \cup \ldots \cup \sigma_n) \colon \Sigma^F \to \Sigma^A \\
\sigma^A(\Delta) \colon \Sigma \to (\Sigma^A \cup \Sigma^A(\Delta)), \text{ where } \Delta{:}\Sigma^F \to \Sigma \text{ is the signature extension} \\
\text{for all } E \in C, \text{ there exists a unique } M \in \mathbf{Mod}(\Sigma^A \cup \Sigma^A(\Delta)) \\
\text{such that } M|_{\Sigma_1^A} = MEv_1(E), \ldots, M|_{\Sigma_n^A} = MEv_n(E) \text{ and} \\
M|_{\sigma^A(\Delta)} = E(UN)\langle MEv_1(E)|_{\sigma_1}, \ldots, MEv_n(E)|_{\sigma_n} \rangle \\
MEv = \{E \mapsto M \mid E \in C, M \in \mathbf{Mod}(\Sigma^A \cup \Sigma^A(\Delta)), \\
M|_{\Sigma_1^A} = MEv_1(E), \ldots, M|_{\Sigma_n^A} = MEv_n(E), \\
M|_{\sigma^A(\Delta)} = E(UN)\langle MEv_1(E)|_{\sigma_1}, \ldots, MEv_n(E)|_{\sigma_n} \rangle\} \\
\hline
\Gamma_s, \Gamma_m, C_s, C \vdash \\
\texttt{unit-appl}\ UN\ \texttt{FIT-ARG-UNIT}_1\texttt{,..., FIT-ARG-UNIT}_n \Rightarrow MEv
\end{array}$$

$$\boxed{\begin{array}{l}
\Sigma, \Gamma_s, C_s \vdash \texttt{FIT-ARG-UNIT} \rhd \sigma{:}\Sigma \to \Sigma^A \\
\qquad\qquad \Sigma, \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{FIT-ARG-UNIT} \Rightarrow MEv^A
\end{array}}$$

Γ_s and Γ_m are compatible global environments; C is a unit context that is compatible with static unit context C_s; then $\sigma \colon \Sigma \to \Sigma^A$ is a signature morphism, and Σ^A and MEv^A are compatible additions to C_s and C.

$$\begin{array}{c}
\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \rhd \Sigma^A \\
\vdash \texttt{SYMB-MAP-ITEMS*} \rhd r \\
\hline
\Sigma, \Gamma_s, C_s \vdash \texttt{fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*} \rhd r|^{\Sigma}_{\Sigma^A}
\end{array}$$

See Sect. 4.1.3 for the definition of $r|^{\Sigma}_{\Sigma^A}{:}\Sigma \rightarrow \Sigma^A$, the signature morphism induced by the symbol map r.

$$\frac{\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow \mathit{MEv}}{\Sigma, \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*} \Rightarrow \mathit{MEv}}$$

The semantics of SYMB-MAP-ITEMS* is given in Sect. 4.5.2.

5.6 Extended Static Semantics

The static semantics of architectural specifications given above performs only a rough static analysis, collecting only very limited static information from what is potentially available about units being defined by unit terms. Consequently, many conditions (notably those concerning amalgamability of models built by unit terms) that one would expect to be discharged by a static semantics have to be checked in the rules for the model semantics. On one hand, this gives extra flexibility: the model semantics by definition stores all the information we have about units. On the other hand though, this is inconvenient in practice, because a static analysis tool that follows the static semantics would fail to detect some errors that could be caught using typechecking methods, without resorting to a theorem prover.

In this section we present an extended static semantics for architectural specifications, which gathers considerably more information than before, and therefore allows for simplification of the model semantics, with some conditions removed. To make the simplification visible below we will repeat the rules of the model semantics, explicitly crossing out the conditions that can be removed thanks to the extended static analysis. The extended static semantics, although presented here in a technically somewhat different form, is essentially equivalent to the one worked out in [62] for a simplified fragment of Casl architectural specifications.

In a way, the extended static semantics is more restrictive than necessary: there are cases where the extended static semantics fails while the semantics given in the previous sections produces a valid result (see Sect. 5.6.6 for the statement of some form of correctness of the extended static semantics). These cases are rare in practice and typically indicate that some conditions follow, often incidentally, from subtle constraints on the models imposed by specifications. We recommend therefore that Casl support tools realize the extended analysis as much as possible[2], and when it fails issue a warning to the user that (s)he has to rely on more powerful tools (e.g., proof mechanisms) to ensure correctness of the specification (or, perhaps more likely, modify the specification).

[2] The conditions used in extended static semantics are undecidable in general; however, there are efficient methods to check them in most cases of interest – see [31] for a more complete analysis.

5.6.1 Architectural Concepts

Signature diagrams are used to keep track of dependencies between units. Diagram nodes correspond to the units declared or built so far and record their signatures. Diagram edges are labelled by signature morphisms that indicate how the unit corresponding to the source of each edge is incorporated as a part of the unit corresponding to its target.

The extended static semantics keeps track of the sharing information on units stored in unit environments. Therefore, a more complicated form of static contexts is necessary: rather than just naming signatures of these units, they also store a diagram of their signatures, keeping track of the mutual dependencies between units.

A *signature diagram* $D\colon \mathit{Shape}(D) \to \mathbf{Sig}$ is a functor from its *shape* (small) category $\mathit{Shape}(D)$ to the category **Sig** of signatures. We will often identify the shape category with its graph. We write $\mathit{Nodes}(D)$ for the set of objects of $\mathit{Shape}(D)$ and $\mathit{Edges}(D)$ for the set of its edges, using the notation $e\colon p \to q$ for $e \in \mathit{Shape}(D)$ and $p, q \in \mathit{Nodes}(D)$ to indicate the source and target of an edge e. Consequently, for $p \in \mathit{Nodes}(D)$, $D(p)$ is a signature and for $e\colon p \to q$ in $\mathit{Shape}(D)$, $D(e)\colon D(p) \to D(q)$ is a signature morphism.

Although we do not rely on this here, it is worth noticing that the shapes of all diagrams we consider are in fact dags (directed acyclic graphs).

It is convenient to assume that both nodes and edges of the diagrams considered come from a given infinite set *Item*; to make the choice of 'new' nodes and edges deterministic, we may assume that *Item* comes equipped with a fixed enumeration – then 'new' always means 'first not used as yet'.

Diag denotes the class of all signature diagrams.

We say that a diagram $D \in \mathit{Diag}$ *extends* $D' \in \mathit{Diag}$ (or that D' is a *subdiagram* of D) if $\mathit{Shape}(D')$ is a subcategory of $\mathit{Shape}(D)$ and D' coincides with D on nodes and edges in $\mathit{Shape}(D)$. Somewhat informal statements that one diagram extends another by a (new) node or by an edge will be used with the obvious meaning.

Diagrams D_i, $i \in \mathcal{I}$ (for an arbitrary set of indices $\mathcal{I}$) *disjointly extends* D' if each D_i, $i \in \mathcal{I}$, extends D' and moreover, for all distinct $i, j \in \mathcal{I}$, $\mathit{Shape}(D_i) \cap \mathit{Shape}(D_j) = \mathit{Shape}(D')$. If this is the case then the union $\bigcup_{i \in \mathcal{I}} D_i$ is well-defined, its shape $\mathit{Shape}(\bigcup_{i \in \mathcal{I}} D_i)$ is the obvious free closure of $\bigcup_{i \in \mathcal{I}} \mathit{Shape}(D_i)$, and the union $\bigcup_{i \in \mathcal{I}} D_i$ extends D'.

Whenever such a 'disjointness' requirement occurs in the rules below, it may be eliminated by the appropriate choices of new nodes and edges in the diagrams involved. Spelling this out in the semantics would require carrying around the set of nodes and edges already used – rather than cluttering the rules and judgements with this extra parameter, we state the disjointness requirement explicitly when necessary.

The extended static semantics takes signatures from signature diagrams, indicating for each non-generic unit the corresponding node in the current signature diagram, in which the unit signature is recorded.

In the extended static semantics, unit contexts carry the diagram of dependencies between unit signatures. The signatures of non-generic units (both those stored in environments as well as those imported by generic units) are 'based' on this diagram, and are given by reference to its nodes.

$$\begin{aligned}
B_s &\in \mathit{StBasedUnitCtx} = \mathit{UnitName} \overset{\text{fin}}{\to} \mathit{Item} \\
(p, \Sigma_1, \ldots, \Sigma_n \to \Sigma) & \\
\text{or } (p, \overline{\Sigma} \to \Sigma) & \\
\text{or } (p, U\Sigma) &\in \mathit{BasedParUnitSig} = \mathit{Item} \times \mathit{ParUnitSig} \\
P_s &\in \mathit{StParUnitCtx} = \mathit{UnitName} \overset{\text{fin}}{\to} \mathit{BasedParUnitSig} \\
(P_s, B_s, D) \text{ or } \mathcal{C}_s &\in \mathit{ExtStUnitCtx} = \\
&\qquad \mathit{StParUnitCtx} \times \mathit{StBasedUnitCtx} \times \mathit{Diag}
\end{aligned}$$

The following requirements are imposed on any extended static unit context (P_s, B_s, D) in *ExtStUnitCtx*:

- the domains of P_s and B_s are disjoint;
- for each $\mathit{UN} \in \mathit{Dom}(B_s)$, $B_s(\mathit{UN})$ is a node in D; and
- for each $\mathit{UN} \in \mathit{Dom}(P_s)$ with $(p, U\Sigma) = P_s(\mathit{UN})$, p is a node in D and $(D(p), U\Sigma) \in \mathit{ImpUnitSig}$.

There is an obvious map $\mathit{ctx}\colon \mathit{ExtStUnitCtx} \to \mathit{StUnitCtx}$ given by

$$\begin{aligned}
\mathit{ctx}(P_s, B_s, D) = {} & \{\mathit{UN} \mapsto D(B_s(\mathit{UN})) \mid \mathit{UN} \in \mathit{Dom}(B_s)\} \cup \\
& \{\mathit{UN} \mapsto (D(p), U\Sigma) \mid \mathit{UN} \in \mathit{Dom}(P_s), P_s(\mathit{UN}) = (p, U\Sigma)\}.
\end{aligned}$$

We define the projection $\mathit{dgm}\colon \mathit{ExtStUnitCtx} \to \mathit{Diag}$ by $\mathit{dgm}(P_s, B_s, D) = D$. We write $\mathit{Dom}(\mathcal{C}_s)$ for $\mathit{Dom}(\mathit{ctx}(\mathcal{C}_s))$ (spelling this out: $\mathit{Dom}(P_s, B_s, D) = \mathit{Dom}(P_s) \cup \mathit{Dom}(B_s)$), and $\mathcal{C}_s^{\emptyset}$ for the empty extended static context (where both maps and the diagram are empty: $\mathcal{C}_s^{\emptyset} = (\{\}, \{\}, \{\})$).

We say that (P'_s, B'_s, D') *extends* (P_s, B_s, D) if D' extends D, $P_s \subseteq P'_s$, and $B_s \subseteq B'_s$.

(P'_s, B'_s, D') is an *admissible addition* to (P_s, B_s, D) if they have disjoint domains (i.e., $\mathit{Dom}(P'_s, B'_s, D') \cap \mathit{Dom}(P_s, B_s, D) = \emptyset$) and D' extends D. We then write $(P_s, B_s, D) + (P'_s, B'_s, D')$ for $(P_s \cup P'_s, B_s \cup B'_s, D')$.

The entities used in the model semantics are required to be compatible with the corresponding entities of the extended static semantics.

Given a diagram $D \in \mathit{Diag}$, we write $\mathbf{Mod}(D)$ for the class of all $\mathit{Nodes}(D)$-indexed model families consistent with D: $\langle M_p \rangle_{p \in \mathit{Nodes}(D)}$ is *consistent* with D if for each $p \in \mathit{Nodes}(D)$, $M_p \in \mathbf{Mod}(D(p))$ and for each $e\colon p \to q$ in D, $M_p = M_q|_{D(e)}$.

As for static contexts in the previous sections, for any extended static context $\mathcal{C}_s$, we define a class $\mathbf{UnitEnv}(\mathcal{C}_s) \subseteq \mathbf{UnitEnv}$ of unit environments. Let $\mathcal{C}_s = (P_s, B_s, D)$. Then $E \in \mathbf{UnitEnv}(\mathcal{C}_s)$ if $\mathit{Dom}(E) \supseteq \mathit{Dom}(\mathcal{C}_s)$ and there exists a model family $\langle M_p \rangle_{p \in \mathit{Nodes}(D)} \in \mathbf{Mod}(D)$ such that:

- for all $\mathit{UN} \in \mathit{Dom}(B_s)$, $E(\mathit{UN}) = M_{B_s(\mathit{UN})}$; and
- for all $\mathit{UN} \in \mathit{Dom}(P_s)$ with $(p, U\Sigma) = P_s(\mathit{UN})$, $E(\mathit{UN}) \in \mathbf{Unit}(U\Sigma)$ and $E(\mathit{UN})$ is compatible with M_p.

The definition of compatibility of a unit context with a static context carries over to the extended case: a unit context $C \in \mathbf{UnitCtx}$ is *compatible* with an extended static context $\mathcal{C}_s \in \mathit{ExtStUnitCtx}$ if $C \subseteq \mathbf{UnitEnv}(\mathcal{C}_s)$.

Moreover, given a unit context C compatible with an extended static context $\mathcal{C}_s$, and $\mathcal{C}'_s \in \mathit{ExtStUnitCtx}$ and $C' \in \mathbf{UnitCtx}$, $\mathcal{C}'_s$ and C' are *compatible extensions* of $\mathcal{C}_s$ and C if $\mathcal{C}'_s$ is an admissible addition to $\mathcal{C}_s$ and $C \cap C'$ is compatible with $\mathcal{C}_s + \mathcal{C}'_s$. Furthermore, given a diagram D' that extends $\mathit{dgm}(\mathcal{C}_s)$, a node $p \in \mathit{Nodes}(D')$, and a model evaluator MEv, (p, Σ, D') and MEv are *compatible additions* to $\mathcal{C}_s$ and C if $D'(p) = \Sigma$ and for some unit name $\mathit{UN} \notin \mathit{Dom}(\mathcal{C}_s)$, $C[\mathit{UN}/\mathit{MEv}]$ is compatible with $\mathcal{C}_s + (\{\}, \{\mathit{UN} \mapsto p\}, D')$. Similarly, given a diagram D' that extends $\mathit{dgm}(\mathcal{C}_s)$, a node $p \in \mathit{Nodes}(D')$, a generic unit signature $U\Sigma$, and a unit evaluator UEv, $(p, U\Sigma, D')$ and UEv are *compatible additions* to $\mathcal{C}_s$ and C if $D'(p)$ is a subsignature of the result signature in $U\Sigma$ and for some unit name $\mathit{UN} \notin \mathit{Dom}(\mathcal{C}_s)$, $C[\mathit{UN}/\mathit{UEv}]$ is compatible with $\mathcal{C}_s + (\{\mathit{UN} \mapsto (p, U\Sigma)\}, \{\}, D')$.

Amalgamability requirements are formulated statically, with reference to the signature diagram.

Given a diagram $D \in \mathit{Diag}$, a *sink* α on a set of nodes $K \subseteq \mathit{Nodes}(D)$ is a signature Σ together with a family of signature morphisms $\alpha_p \colon D(p) \to \Sigma$, $p \in K$. We say that D *ensures amalgamability* along $\alpha = (\Sigma, \langle \alpha_p \colon D(p) \to \Sigma \rangle_{p \in K})$ if for every model family $\langle M_q \rangle_{q \in \mathit{Nodes}(D)}$ consistent with D there exists a unique model $M \in \mathbf{Mod}(\Sigma)$ such that for all $p \in K$, $M|_{\alpha_p} = M_p$. We use slightly different but hopefully self-explanatory variants of this terminology and notation for special cases when K consists of one or two elements, is given by enumeration, etc.

Although we have formulated this amalgamability condition in terms of model families, it is an essentially static property: the class of model families considered is not restricted by any axioms, but only by signatures and morphisms between them. This static nature of the condition may be made explicit by embedding the underlying institution into an institution that admits amalgamation. For Casl this can be given by considering so-called enriched Casl signatures, where, roughly, one admits an arbitrary category, rather than just a pre-order, of subsort embeddings. See [62] for details.

Given the above definitions, we define below the extended static semantics, enriching the static semantics above by the analysis of sharing between units. The judgements of this extended static semantics will be written using the symbols $\_\_ \vdash \_\_ \mathrel{\rhd\!\!\rhd} \_\_$. As with the semantics above, we give the rules for extended static semantics for all constructs of the abstract syntax. The headings corresponding to each syntactic category will be given, quoting also the corresponding judgements of the semantics above. We recall the conditions linking the static and model semantics from the previous sections, and add to these formal properties of the extended static semantics as well as conditions linking the extended static semantics with the static and model semantics above. Many of the rules of the extended static semantics differ only in uninteresting details from the rules of the static semantics given above. The essential changes are mainly in the rules for the static analysis of unit terms.

For the sake of readability, we recall the abstract syntax, although we skip the explanation included above.

Finally, in Sect. 5.6.6 we discuss correctness and completeness of the extended static semantics w.r.t. the simple static and model semantics given in the previous section.

5.6.2 Architectural Specification Definitions

```
ARCH-SPEC-DEFN ::= arch-spec-defn ARCH-SPEC-NAME ARCH-SPEC
ARCH-SPEC      ::= BASIC-ARCH-SPEC | ARCH-SPEC-NAME
```

$$\boxed{\begin{array}{c} \Gamma_s \vdash \texttt{ARCH-SPEC-DEFN} \rhd \Gamma'_s \qquad\qquad \Gamma_s, \Gamma_m \vdash \texttt{ARCH-SPEC-DEFN} \Rightarrow \Gamma'_m \\ \Gamma_s \vdash \texttt{ARCH-SPEC-DEFN} \mathrel{\rhd\!\!\rhd} \Gamma'_s \end{array}}$$

Γ_s and Γ_m are compatible global environments; then Γ'_s and Γ'_m are compatible as well, and extend Γ_s and Γ_m, respectively.

If $\Gamma_s \vdash \texttt{ARCH-SPEC-DEFN} \mathrel{\rhd\!\!\rhd} \Gamma'_s$ then $\Gamma_s \vdash \texttt{ARCH-SPEC-DEFN} \rhd \Gamma'_s$.

$$\frac{\begin{array}{c} \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \\ ASN \notin Dom(\mathcal{G}_s) \cup Dom(\mathcal{V}_s) \cup Dom(\mathcal{A}_s) \cup Dom(\mathcal{T}_s) \\ \Gamma_s \vdash \texttt{ARCH-SPEC} \mathrel{\rhd\!\!\rhd} A\Sigma \end{array}}{\begin{array}{l} \Gamma_s \vdash \texttt{arch-spec-defn}\ ASN\ \ \texttt{ARCH-SPEC} \mathrel{\rhd\!\!\rhd} \\ \qquad\qquad\qquad\qquad (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s \cup \{ASN \mapsto A\Sigma\}, \mathcal{T}_s) \end{array}}$$

$$\boxed{\begin{array}{c} \Gamma_s \vdash \texttt{ARCH-SPEC} \rhd A\Sigma \qquad\qquad \Gamma_s, \Gamma_m \vdash \texttt{ARCH-SPEC} \Rightarrow \mathcal{AM} \\ \Gamma_s \vdash \texttt{ARCH-SPEC} \mathrel{\rhd\!\!\rhd} A\Sigma \end{array}}$$

Γ_s and Γ_m are compatible global environments; then $\mathcal{AM} \in \mathbf{ArchSpec}(A\Sigma)$.

If $\Gamma_s \vdash \texttt{ARCH-SPEC} \mathrel{\rhd\!\!\rhd} A\Sigma$ then $\Gamma_s \vdash \texttt{ARCH-SPEC} \rhd A\Sigma$.

$$\frac{\Gamma_s \vdash \texttt{BASIC-ARCH-SPEC} \mathrel{\rhd\!\!\rhd} A\Sigma}{\Gamma_s \vdash \texttt{BASIC-ARCH-SPEC}\,\text{qua}\,\texttt{ARCH-SPEC} \mathrel{\rhd\!\!\rhd} A\Sigma}$$

$$\frac{ASN \in Dom(\mathcal{A}_s)}{(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \vdash ASN \text{ qua } \texttt{ARCH-SPEC} \rhd\!\!\rhd \mathcal{A}_s(ASN)}$$

```
BASIC-ARCH-SPEC ::= basic-arch-spec UNIT-DECL-DEFN+ RESULT-UNIT
UNIT-DECL-DEFN  ::= UNIT-DECL | UNIT-DEFN
RESULT-UNIT     ::= result-unit UNIT-EXPRESSION
```

$$\boxed{\begin{array}{c} \Gamma_s \vdash \texttt{BASIC-ARCH-SPEC} \rhd A\Sigma \qquad\qquad \Gamma_s, \Gamma_m \vdash \texttt{BASIC-ARCH-SPEC} \Rightarrow \mathcal{AM} \\ \Gamma_s \vdash \texttt{BASIC-ARCH-SPEC} \rhd\!\!\rhd A\Sigma \end{array}}$$

Γ_s and Γ_m are compatible global environments; then $\mathcal{AM} \in \mathbf{ArchSpec}(A\Sigma)$.

If $\Gamma_s \vdash \texttt{BASIC-ARCH-SPEC} \rhd\!\!\rhd A\Sigma$ then $\Gamma_s \vdash \texttt{BASIC-ARCH-SPEC} \rhd A\Sigma$.

$$\frac{\begin{array}{c} \Gamma_s \vdash \texttt{UNIT-DECL-DEFN}^+ \rhd\!\!\rhd \mathcal{C}_s \\ \Gamma_s, \mathcal{C}_s \vdash \texttt{RESULT-UNIT} \rhd\!\!\rhd U\Sigma \end{array}}{\Gamma_s \vdash \texttt{basic-arch-spec UNIT-DECL-DEFN}^+ \ \texttt{RESULT-UNIT} \rhd\!\!\rhd (ctx(\mathcal{C}_s), U\Sigma)}$$

$$\boxed{\begin{array}{c} \Gamma_s \vdash \texttt{UNIT-DECL-DEFN}^+ \rhd C_s \qquad\qquad \Gamma_s, \Gamma_m \vdash \texttt{UNIT-DECL-DEFN}^+ \Rightarrow C \\ \Gamma_s \vdash \texttt{UNIT-DECL-DEFN}^+ \rhd\!\!\rhd \mathcal{C}_s \end{array}}$$

Γ_s and Γ_m are compatible global environments; then C is compatible with C_s too.

If $\Gamma_s \vdash \texttt{UNIT-DECL-DEFN}^+ \rhd\!\!\rhd \mathcal{C}_s$ then $\Gamma_s \vdash \texttt{UNIT-DECL-DEFN}^+ \rhd C_s$, with $C_s = ctx(\mathcal{C}_s)$.

$$\frac{\begin{array}{c} \Gamma_s, \mathcal{C}_s^{\emptyset} \vdash \texttt{UNIT-DECL-DEFN}_1 \rhd\!\!\rhd (\mathcal{C}_s)_1 \\ \cdots \\ \Gamma_s, (\mathcal{C}_s)_{n-1} \vdash \texttt{UNIT-DECL-DEFN}_n \rhd\!\!\rhd (\mathcal{C}_s)_n \end{array}}{\Gamma_s \vdash \texttt{UNIT-DECL-DEFN}_1 \ldots \texttt{UNIT-DECL-DEFN}_n \rhd\!\!\rhd (\mathcal{C}_s)_n}$$

$$\boxed{\begin{array}{c} \Gamma_s, C_s \vdash \texttt{UNIT-DECL-DEFN} \rhd C_s' \qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-DECL-DEFN} \Rightarrow C' \\ \Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-DECL-DEFN} \rhd\!\!\rhd \mathcal{C}_s' \end{array}}$$

Γ_s and Γ_m are compatible global environments, and C is compatible with C_s; then C' and C_s' are compatible, $C_s \subseteq C_s'$ and $C \supseteq C'$.

$\mathcal{C}_s'$ extends $\mathcal{C}_s$.

If $\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-DECL-DEFN} \rhd\!\!\rhd \mathcal{C}_s'$ then $\Gamma_s, ctx(\mathcal{C}_s) \vdash \texttt{UNIT-DECL-DEFN} \rhd C_s'$ and $C_s' = ctx(\mathcal{C}_s')$. Moreover, if $\Gamma_s, \Gamma_m, ctx(\mathcal{C}_s), C \vdash \texttt{UNIT-DECL-DEFN} \Rightarrow C'$ and C is compatible with $\mathcal{C}_s$ then C' and $\mathcal{C}_s'$ are compatible.

$$\frac{\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-DECL} \rhd\!\!\rhd \mathcal{C}_s'}{\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-DECL} \text{ qua } \texttt{UNIT-DECL-DEFN} \rhd\!\!\rhd \mathcal{C}_s + \mathcal{C}_s'}$$

$$\frac{\mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{UNIT-DEFN} \mathrel{\rhd\!\!\rhd} \mathcal{C}'_s}{\mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{UNIT-DEFN}\,\text{qua}\,\texttt{UNIT-DECL-DEFN} \mathrel{\rhd\!\!\rhd} \mathcal{C}_s + \mathcal{C}'_s}$$

$$\boxed{\begin{array}{c} \mathit{\Gamma}_s, C_s \vdash \texttt{RESULT-UNIT} \rhd \mathit{U\Sigma} \qquad\qquad \mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{RESULT-UNIT} \Rightarrow \mathit{UEv} \\ \mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{RESULT-UNIT} \mathrel{\rhd\!\!\rhd} \mathit{U\Sigma} \end{array}}$$

$\mathit{\Gamma}_s$ and $\mathit{\Gamma}_m$ are compatible global environments, and C is a unit context that is compatible with static unit context C_s; then $\mathit{U\Sigma}$ and $\mathit{UEv} \in \mathbf{UnitEval}(\mathit{U\Sigma})$ are compatible additions to C_s and C.

If $\mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{RESULT-UNIT} \mathrel{\rhd\!\!\rhd} \mathit{U\Sigma}$ then $\mathit{\Gamma}_s, \mathit{ctx}(\mathcal{C}_s) \vdash \texttt{RESULT-UNIT} \rhd \mathit{U\Sigma}$.

$$\frac{\mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{UNIT-EXPRESSION} \mathrel{\rhd\!\!\rhd} (p, \mathit{U\Sigma}, D)}{\mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{result-unit UNIT-EXPRESSION} \mathrel{\rhd\!\!\rhd} \mathit{U\Sigma}}$$

5.6.3 Unit Declarations and Definitions

Unit Declarations

```
UNIT-DECL     ::= unit-decl UNIT-NAME UNIT-SPEC UNIT-IMPORTED
UNIT-IMPORTED ::= unit-imported UNIT-TERM*
UNIT-NAME     ::= SIMPLE-ID
```

$$\boxed{\begin{array}{c} \mathit{\Gamma}_s, C_s \vdash \texttt{UNIT-DECL} \rhd C'_s \qquad\qquad \mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \texttt{UNIT-DECL} \Rightarrow C' \\ \mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{UNIT-DECL} \mathrel{\rhd\!\!\rhd} \mathcal{C}'_s \end{array}}$$

$\mathit{\Gamma}_s$ and $\mathit{\Gamma}_m$ are compatible global environments, and C is a unit context that is compatible with static unit context C_s; then C'_s and C' are compatible extensions of C_s and C.

$\mathcal{C}'_s$ is an admissible addition to $\mathcal{C}_s$.

If $\mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{UNIT-DECL} \mathrel{\rhd\!\!\rhd} \mathcal{C}'_s$ then $\mathit{\Gamma}_s, \mathit{ctx}(\mathcal{C}_s) \vdash \texttt{UNIT-DECL} \rhd C'_s$, with $C'_s = \mathit{ctx}(\mathcal{C}'_s)$. Moreover, if $\mathit{\Gamma}_s, \mathit{\Gamma}_m, \mathit{ctx}(\mathcal{C}_s), C \vdash \texttt{UNIT-DECL} \Rightarrow C'$ and C is compatible with $\mathcal{C}_s$ then C' and $\mathcal{C}'_s$ are compatible extensions of C and $\mathcal{C}_s$.

$$\frac{\begin{array}{c} \mathcal{C}_s \vdash \texttt{UNIT-IMPORTED} \mathrel{\rhd\!\!\rhd} (p, D) \\ D(p), \mathit{\Gamma}_s \vdash \texttt{UNIT-SPEC} \rhd \mathit{\Sigma} \\ \mathit{UN} \notin \mathit{Dom}(\mathcal{C}_s) \\ D' \text{ extends } D \text{ by new node } q \text{ with } D'(q) = D(p) \cup \mathit{\Sigma} \\ \qquad \text{and edge } e\colon p \rightarrow q \text{ with } D'(e) = \iota_{D(p) \subseteq D'(q)} \text{ being the inclusion} \end{array}}{\begin{array}{l} \mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{unit-decl}\ \mathit{UN}\ \texttt{UNIT-SPEC UNIT-IMPORTED} \mathrel{\rhd\!\!\rhd} \\ \qquad\qquad\qquad\qquad\qquad\qquad (\{\}, \{\mathit{UN} \mapsto q\}, D') \end{array}}$$

$$\frac{\begin{array}{c} \mathcal{C}_s \vdash \texttt{UNIT-IMPORTED} \rhd\!\!\rhd (p, D) \\ D(p), \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd \overline{\Sigma} \to \Sigma_0 \\ UN \notin Dom(\mathcal{C}_s) \end{array}}{\begin{array}{l} \Gamma_s, \mathcal{C}_s \vdash \texttt{unit-decl}\ \ UN\ \ \texttt{UNIT-SPEC UNIT-IMPORTED} \rhd\!\!\rhd \\ \qquad\qquad\qquad\qquad (\{UN \mapsto (p, \overline{\Sigma} \to \Sigma_0 \cup \Sigma^I)\}, \{\}, D) \end{array}}$$

$$\boxed{\begin{array}{c} C_s \vdash \texttt{UNIT-IMPORTED} \rhd \Sigma \qquad\qquad C_s, C \vdash \texttt{UNIT-IMPORTED} \Rightarrow MEv \\ \mathcal{C}_s \vdash \texttt{UNIT-IMPORTED} \rhd\!\!\rhd (p, D) \end{array}}$$

C is a unit context that is compatible with static unit context C_s; then Σ and MEv are compatible additions to C_s and C.

D extends $dgm(\mathcal{C}_s)$ and $p \in Nodes(D)$.

If $\mathcal{C}_s \vdash \texttt{UNIT-IMPORTED} \rhd\!\!\rhd (p, D)$ then $ctx(\mathcal{C}_s) \vdash \texttt{UNIT-IMPORTED} \rhd \Sigma$, with $\Sigma = D(p)$. Moreover, if $ctx(\mathcal{C}_s), C \vdash \texttt{UNIT-IMPORTED} \Rightarrow MEv$ and C is compatible with $\mathcal{C}_s$ then (p, D) and MEv are compatible additions to C and $\mathcal{C}_s$.

$$\frac{\begin{array}{c} \mathcal{C}_s \vdash \texttt{UNIT-TERM}_1 \rhd\!\!\rhd (p_1, D_1) \quad \cdots \quad \mathcal{C}_s \vdash \texttt{UNIT-TERM}_k \rhd\!\!\rhd (p_k, D_k) \\ \Sigma = D_1(p_1) \cup \ldots \cup D_k(p_k) \\ D_1, \ldots, D_k \text{ disjointly extend } dgm(\mathcal{C}_s) \qquad D' = D_1 \cup \ldots \cup D_k \\ D' \text{ ensures amalgamability along } (\Sigma, \langle \iota_{D_i(p_i) \subseteq \Sigma} \colon D'(p_i) \to \Sigma \rangle_{i=1,\ldots,k}) \\ \begin{array}{l} D'' \text{ extends } D' \text{ by new node } q \text{ with } D''(q) = \Sigma \\ \qquad \text{and edges } e_i \colon p_i \to q \text{ with } D''(e_i) = \iota_{D_i(p_i) \subseteq \Sigma}, \text{ for } i = 1, \ldots, k \end{array} \end{array}}{\mathcal{C}_s \vdash \texttt{unit-imported UNIT-TERM}_1\texttt{,...,UNIT-TERM}_k \rhd\!\!\rhd (q, D'')}$$

Assuming that the extended static analysis is successful for the phrase `unit-imported UNIT-TERM₁,...,UNIT-TERMₖ`, we can simplify the corresponding rule in the model semantics as follows:

$$\frac{\begin{array}{c} C_s, C \vdash \texttt{UNIT-TERM}_1 \Rightarrow MEv_1 \quad \cdots \quad C_s, C \vdash \texttt{UNIT-TERM}_k \Rightarrow MEv_k \\ \cancel{\text{for each } E \in C, MEv_1(E), \ldots, MEv_k(E) \text{ are compatible}} \end{array}}{\begin{array}{l} C_s, C \vdash \texttt{unit-imported UNIT-TERM}_1\texttt{,...,UNIT-TERM}_k \Rightarrow \\ \qquad\qquad\qquad\qquad \lambda E \in C \cdot MEv_1(E) \oplus \ldots \oplus MEv_k(E) \end{array}}$$

Unit Definitions

```
UNIT-DEFN ::= unit-defn UNIT-NAME UNIT-EXPRESSION
```

$$\boxed{\begin{array}{c} \Gamma_s, C_s \vdash \texttt{UNIT-DEFN} \rhd C'_s \qquad\qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-DEFN} \Rightarrow C' \\ \Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-DEFN} \rhd\!\!\rhd \mathcal{C}'_s \end{array}}$$

Γ_s and Γ_m are compatible global environments, and C is a unit context that is compatible with static unit context C_s; then C'_s and C' are compatible extensions of C_s and C, and moreover, C' maps any unit in its domain to a one-element set of unit evaluators.

$\mathcal{C}'_s$ is an admissible addition to $\mathcal{C}_s$.

If $\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-DEFN} \mathrel{\rhd\!\!\rhd} \mathcal{C}'_s$ then $\Gamma_s, ctx(\mathcal{C}_s) \vdash \texttt{UNIT-DEFN} \rhd C'_s$, with $C'_s = ctx(\mathcal{C}'_s)$. Moreover, if $\Gamma_s, \Gamma_m, ctx(\mathcal{C}_s), C \vdash \texttt{UNIT-DEFN} \Rightarrow C'$ and C is compatible with $\mathcal{C}_s$ then $\mathcal{C}'_s$ and C' are compatible extensions of C and $\mathcal{C}_s$.

$$\dfrac{\begin{array}{c}\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-EXPRESSION} \mathrel{\rhd\!\!\rhd} (p, \Sigma, D) \\ UN \notin Dom(\mathcal{C}_s)\end{array}}{\Gamma_s, \mathcal{C}_s \vdash \texttt{unit-defn}\ \ UN\ \ \texttt{UNIT-EXPRESSION} \mathrel{\rhd\!\!\rhd} (\{\}, \{UN \mapsto p\}, D)}$$

$$\dfrac{\begin{array}{c}\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-EXPRESSION} \mathrel{\rhd\!\!\rhd} (p, \overline{\Sigma} \to \Sigma, D) \\ UN \notin Dom(C_s)\end{array}}{\Gamma_s, C_s \vdash \texttt{unit-defn}\ \ UN\ \ \texttt{UNIT-EXPRESSION} \mathrel{\rhd\!\!\rhd} (\{UN \mapsto (p, \overline{\Sigma} \to \Sigma)\}, \{\}, D)}$$

5.6.4 Unit Specifications

For unit specifications, extended static semantics coincides with the static semantics above, so no new rules are needed.

5.6.5 Unit Expressions

```
UNIT-EXPRESSION ::= unit-expression UNIT-BINDING* UNIT-TERM
UNIT-BINDING    ::= unit-binding UNIT-NAME UNIT-SPEC
```

$$\boxed{\begin{array}{lcr}\Gamma_s, C_s \vdash \texttt{UNIT-EXPRESSION} \rhd U\Sigma & & \\ & & \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-EXPRESSION} \Rightarrow UEv \\ & \Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-EXPRESSION} \mathrel{\rhd\!\!\rhd} (p, U\Sigma, D) & \end{array}}$$

Γ_s and Γ_m are compatible global environments, and C is a unit context that is compatible with static unit context C_s; then $U\Sigma$ and UEv are compatible additions to C_s and C.

D extends $dgm(\mathcal{C}_s)$, $p \in Nodes(D)$. For $U\Sigma = \Sigma \in \mathbf{Sig}$, $D(p) = \Sigma$. For $U\Sigma = \overline{\Sigma} \to \Sigma \in ParUnitSig$, $D(p)$ is the empty signature.

If we have $\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-EXPRESSION} \mathrel{\rhd\!\!\rhd} (p, U\Sigma, D)$ then $\Gamma_s, ctx(\mathcal{C}_s) \vdash \texttt{UNIT-EXPRESSION} \rhd U\Sigma$. Moreover, if we also have $\Gamma_s, \Gamma_m, ctx(\mathcal{C}_s), C \vdash \texttt{UNIT-EXPRESSION} \Rightarrow UEv$ and C is compatible with $\mathcal{C}_s$ then UEv and $(p, U\Sigma, D)$ are compatible additions to C and $\mathcal{C}_s$.

$$\dfrac{\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-TERM} \mathrel{\rhd\!\!\rhd} (p, D)}{\Gamma_s, C_s \vdash \texttt{unit-expression UNIT-TERM} \mathrel{\rhd\!\!\rhd} (p, D(p), D)}$$

$$
\begin{array}{c}
\mathit{\Gamma}_s \vdash \texttt{UNIT-BINDING}_1 \mathrel{\rhd\!\!\!\rhd} (UN_1, \mathit{\Sigma}_1) \;\cdots\; \mathit{\Gamma}_s \vdash \texttt{UNIT-BINDING}_n \mathrel{\rhd\!\!\!\rhd} (UN_n, \mathit{\Sigma}_n) \\
\overline{\mathit{\Sigma}} = \langle \mathit{\Sigma}_1, \ldots, \mathit{\Sigma}_n \rangle \qquad \mathit{\Sigma} = \mathit{\Sigma}_1 \cup \ldots \cup \mathit{\Sigma}_n \\
UN_1, \ldots, UN_n \text{ are distinct} \qquad \{UN_1, \ldots, UN_n\} \cap Dom(\mathcal{C}_s) = \emptyset \\
D' \text{ extends } dgm(\mathcal{C}_s) \text{ by new node } q \text{ with } D'(q) = \mathit{\Sigma}, \\
\qquad \text{nodes } p_i \text{ and edges } e_i \colon p_i \to q \text{ with } D'(e_i) = \iota_{\mathit{\Sigma}_i \subseteq \mathit{\Sigma}}, \text{ for } i = 1, \ldots, n \\
\mathcal{C}'_s = (\{\}, \{UN_1 \mapsto p_1, \ldots, UN_n \mapsto p_n\}, D') \\
\mathit{\Gamma}_s, \mathcal{C}_s + \mathcal{C}'_s \vdash \texttt{UNIT-TERM} \mathrel{\rhd\!\!\!\rhd} (p, D'') \\
D'' \text{ ensures amalgamability along } (D''(p), \langle id_{D''(p)}, \iota_{\mathit{\Sigma}_i \subseteq D''(p)} \rangle_{i=1,\ldots,n}) \\
D''' \text{ extends } D'' \text{ by new node } z \text{ with } D'''(z) = \emptyset \\
\hline
\mathit{\Gamma}_s, \mathcal{C}_s \vdash \\
\quad \texttt{unit-expression UNIT-BINDING}_1\texttt{,...,UNIT-BINDING}_n \texttt{ UNIT-TERM} \\
\hfill \mathrel{\rhd\!\!\!\rhd} (z, \overline{\mathit{\Sigma}} \to D''(p), D''')
\end{array}
$$

Assuming that the extended static analysis is successful for the phrase `unit-expression UNIT-BINDING`$_1$`,...,UNIT-BINDING`$_n$ `UNIT-TERM`, we can simplify the corresponding rule in the model semantics as follows:

$$
\begin{array}{c}
\mathit{\Gamma}_s \vdash \texttt{UNIT-BINDING}_1 \rhd (UN_1, \mathit{\Sigma}_1) \;\cdots\; \mathit{\Gamma}_s \vdash \texttt{UNIT-BINDING}_n \rhd (UN_n, \mathit{\Sigma}_n) \\
\mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{UNIT-BINDING}_1 \Rightarrow (UN_1, \mathcal{M}_1) \\
\cdots \\
\mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{UNIT-BINDING}_n \Rightarrow (UN_n, \mathcal{M}_n) \\
\mathit{\Sigma}^P = \mathit{\Sigma}_1 \cup \ldots \cup \mathit{\Sigma}_n \qquad \mathcal{M}^P = \mathcal{M}_1 \oplus \ldots \oplus \mathcal{M}_n \\
UN^P \notin Dom(C_s) \cup \{UN_1, \ldots, UN_n\} \\
C'_s = \{UN^P \mapsto \mathit{\Sigma}^P, UN_1 \mapsto \mathit{\Sigma}_1, \ldots, UN_n \mapsto \mathit{\Sigma}_n\} \\
C' = C^{\emptyset}[UN^P/\mathcal{M}^P][UN_1/(\lambda E \cdot E(UN^P)|_{\mathit{\Sigma}_1})] \ldots [UN_n/(\lambda E \cdot E(UN^P)|_{\mathit{\Sigma}_n})] \\
\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s \cup C'_s, C \cap C' \vdash \texttt{UNIT-TERM} \Rightarrow MEv
\end{array}
$$

~~for all $E \in C \cap C'$, $MEv(E)|_{\mathit{\Sigma}^P} = E(UN^P)$~~

$$
\begin{array}{l}
UEv \in \mathbf{UnitEval}(\mathit{\Sigma}_1, \ldots, \mathit{\Sigma}_n \to \mathit{\Sigma}) \text{ is such that } Dom(UEv) = C \text{ and} \\
\text{for } E \in C, \\
\quad \langle M_1, \ldots, M_n \rangle \in Dom(UEv(E)) \subseteq \mathbf{CompMod}(\mathit{\Sigma}_1, \ldots, \mathit{\Sigma}_n) \text{ iff} \\
\quad\; E + \{UN^P \mapsto (M_1 \oplus \ldots \oplus M_n), UN_1 \mapsto M_1, \ldots, UN_n \mapsto M_n\} \in C \cap C' \\
\quad \text{and then for } \langle M_1, \ldots, M_n \rangle \in Dom(UEv(E)), \\
\qquad UEv(E)\langle M_1, \ldots, M_n \rangle = \\
\qquad\quad MEv(E + \{UN^P \mapsto (M_1 \oplus \ldots \oplus M_n), UN_1 \mapsto M_1, \ldots, UN_n \mapsto M_n\}) \\
\hline
\mathit{\Gamma}_s, \mathit{\Gamma}_m, C_s, C \vdash \\
\quad \texttt{unit-expression UNIT-BINDING}_1\texttt{,...,UNIT-BINDING}_n \texttt{ UNIT-TERM} \\
\hfill \Rightarrow UEv
\end{array}
$$

$$
\boxed{
\begin{array}{c}
\mathit{\Gamma}_s \vdash \texttt{UNIT-BINDING} \rhd (UN, \mathit{\Sigma}) \qquad \mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{UNIT-BINDING} \Rightarrow (UN, \mathcal{M}) \\
\mathit{\Gamma}_s \vdash \texttt{UNIT-BINDING} \mathrel{\rhd\!\!\!\rhd} (UN, \mathit{\Sigma})
\end{array}
}
$$

Γ_s and Γ_m are compatible global environments; then UN is a unit name (the same for the static and model semantics) and $\mathcal{M} \subseteq \mathbf{Mod}(\Sigma)$.

If $\Gamma_s \vdash \texttt{UNIT-BINDING} \rhd\!\!\!> (UN, \Sigma)$ then $\Gamma_s \vdash \texttt{UNIT-BINDING} \rhd (UN, \Sigma)$.

$$\frac{\emptyset, \Gamma_s \vdash \texttt{UNIT-SPEC} \rhd\!\!\!> \Sigma}{\Gamma_s \vdash \texttt{unit-binding}\ \ UN\ \ \texttt{UNIT-SPEC} \rhd\!\!\!> (UN, \Sigma)}$$

Unit Terms

```
UNIT-TERM ::= UNIT-REDUCTION | UNIT-TRANSLATION | AMALGAMATION
            | LOCAL-UNIT | UNIT-APPL
```

$$\boxed{\begin{array}{c} \Gamma_s, C_s \vdash \texttt{UNIT-TERM} \rhd \Sigma \qquad\qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow MEv \\ \Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-TERM} \rhd\!\!\!> (p, D) \end{array}}$$

Γ_s and Γ_m are compatible global environments; C is a unit context that is compatible with static unit context C_s; then Σ and MEv are compatible additions to C_s and C.

D extends $dgm(\mathcal{C}_s)$, $p \in Nodes(D)$.

If $\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-TERM} \rhd\!\!\!> (p, D)$ then $\Gamma_s, ctx(\mathcal{C}_s) \vdash \texttt{UNIT-TERM} \rhd \Sigma$, with $\Sigma = D(p)$. Moreover, if $\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow MEv$ and C is compatible with $\mathcal{C}_s$ then MEv and (p, Σ, D) are compatible additions to C and $\mathcal{C}_s$.

Rules elided.

Unit Translations

```
UNIT-TRANSLATION ::= unit-translation UNIT-TERM RENAMING
```

$$\boxed{\begin{array}{l} \Gamma_s, C_s \vdash \texttt{UNIT-TRANSLATION} \rhd \Sigma \\ \hfill \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TRANSLATION} \Rightarrow MEv \\ \hfill \Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-TRANSLATION} \rhd\!\!\!> (p, D) \hfill \end{array}}$$

See the requirements for the semantics of UNIT-TERM.

$$\frac{\begin{array}{c} \Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-TERM} \rhd\!\!\!> (p, D) \\ D(p) \vdash \texttt{RENAMING} \rhd \sigma{:}D(p) \rightarrow \Sigma' \\ D \text{ ensures amalgamability along } (\Sigma', \langle \sigma{:}D(p) \rightarrow \Sigma' \rangle) \\ D' \text{ extends } D \text{ by new node } q \text{ and edge } e\colon p \rightarrow q \text{ with } D'(e) = \sigma \end{array}}{\Gamma_s, \mathcal{C}_s \vdash \texttt{unit-translation UNIT-TERM RENAMING} \rhd\!\!\!> (q, D')}$$

Assuming that the extended static analysis is successful for the phrase `unit-translation UNIT-TERM RENAMING`, we can simplify the corresponding rule in the model semantics as follows:

$$\begin{array}{c}
\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \rhd \Sigma \\
\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow \mathit{MEv} \\
\Sigma \vdash \texttt{RENAMING} \rhd \sigma{:}\Sigma \to \Sigma' \\
\enclose{horizontalstrike}{\text{for all } E \in C\text{, there exists a unique } M' \in \mathbf{Mod}(\Sigma') \text{ with } M'|_\sigma = \mathit{MEv}(E)} \\
\hline
\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{unit-translation UNIT-TERM RENAMING} \Rightarrow \\
\qquad\qquad \{E \mapsto M' \mid E \in C, M' \in \mathbf{Mod}(\Sigma'), M'|_\sigma = \mathit{MEv}(E)\}
\end{array}$$

The semantics of `RENAMING` is given in Sect. 4.2.1.

Unit Reductions

```
UNIT-REDUCTION ::= unit-reduction UNIT-TERM RESTRICTION
```

$$\boxed{\begin{array}{c}
\Gamma_s, C_s \vdash \texttt{UNIT-REDUCTION} \rhd \Sigma \quad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-REDUCTION} \Rightarrow \mathit{MEv} \\
\Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-REDUCTION} \mathrel{\rhd\!\!\!\rhd} (p, D)
\end{array}}$$

See the requirements for the semantics of `UNIT-TERM`.

$$\begin{array}{c}
\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \mathrel{\rhd\!\!\!\rhd} (p, D) \\
(\emptyset, D(p)) \vdash \texttt{RESTRICTION} \rhd \sigma{:}\Sigma' \to \Sigma'' \\
D' \text{ extends } D \text{ by new node } q \text{ and edge } e\colon q \to p \text{ with } D'(e) = \iota_{\Sigma' \subseteq D(p)} \\
D' \text{ ensures amalgamability along } (\Sigma'', \langle \sigma\colon D'(q) \to \Sigma'' \rangle) \\
D'' \text{ extends } D' \text{ by new node } q' \text{ and edge } e\colon q \to q' \text{ with } D''(e) = \sigma \\
\hline
\Gamma_s, C_s \vdash \texttt{unit-reduction UNIT-TERM RESTRICTION} \mathrel{\rhd\!\!\!\rhd} (q', D'')
\end{array}$$

Assuming that the extended static analysis is successful for the phrase `unit-reduction UNIT-TERM REDUCTION`, we can simplify the corresponding rule in the model semantics as follows:

$$\begin{array}{c}
\Gamma_s, C_s \vdash \texttt{UNIT-TERM} \rhd \Sigma \\
\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM} \Rightarrow \mathit{MEv} \\
(\emptyset, \Sigma) \vdash \texttt{REDUCTION} \rhd \sigma{:}\Sigma' \to \Sigma'' \\
\enclose{horizontalstrike}{\text{for all } E \in C,} \\
\qquad \enclose{horizontalstrike}{\text{there exists a unique } M'' \in \mathbf{Mod}(\Sigma'') \text{ with } M''|_\sigma = \mathit{MEv}(E)|_{\Sigma'}} \\
\hline
\Gamma_s, \Gamma_m, C_s, C \vdash \texttt{unit-reduction UNIT-TERM RESTRICTION} \Rightarrow \\
\qquad\qquad \{E \mapsto M'' \mid E \in C, M'' \in \mathbf{Mod}(\Sigma''), M''|_\sigma = \mathit{MEv}(E)|_{\Sigma'}\}
\end{array}$$

The semantics for `RESTRICTION` is given in Sect. 4.2.2.

Amalgamations

```
AMALGAMATION ::= amalgamation UNIT-TERM+
```

$$\boxed{\begin{array}{c} \Gamma_s, C_s \vdash \texttt{AMALGAMATION} \rhd \Sigma \qquad\qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{AMALGAMATION} \Rightarrow \mathit{MEv} \\ \Gamma_s, \mathcal{C}_s \vdash \texttt{AMALGAMATION} \rhd\!\!\!\rhd (p, D) \end{array}}$$

See the requirements for the semantics of UNIT-TERM.

$$\frac{\begin{array}{c} \Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-TERM}_1 \rhd\!\!\!\rhd (p_1, D_1) \quad \cdots \quad \Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-TERM}_n \rhd\!\!\!\rhd (p_n, D_n) \\ \Sigma = D_1(p_1) \cup \ldots \cup D_n(p_n) \\ D_1, \ldots, D_n \text{ disjointly extend } \mathit{dgm}(\mathcal{C}_s) \qquad D' = D_1 \cup \ldots \cup D_n \\ D' \text{ ensures amalgamability along } (\Sigma, \langle \iota_{D_i(p_i) \subseteq \Sigma} \colon D'(p_i) \to \Sigma \rangle_{i=1,\ldots,n}) \\ D'' \text{ extends } D' \text{ by new node } q \text{ with } D''(q) = \Sigma \\ \text{and edges } e_i \colon p_i \to q \text{ with } D''(e_i) = \iota_{D_i(p_i) \subseteq \Sigma}, i = 1, \ldots, n \end{array}}{\Gamma_s, C_s \vdash \texttt{amalgamation UNIT-TERM}_1\texttt{,...,UNIT-TERM}_n \rhd\!\!\!\rhd (q, D'')}$$

Assuming that the extended static analysis is successful for the phrase `amalgamation (UNIT-TERM`$_1$`,...,UNIT-TERM`$_n$`)`, we can simplify the corresponding rule in the model semantics as follows:

$$\frac{\begin{array}{c} \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM}_1 \Rightarrow \mathit{MEv}_1 \\ \cdots \\ \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-TERM}_n \Rightarrow \mathit{MEv}_n \\ \cancel{\text{for all } E \in C, \mathit{MEv}_1(E), \ldots, \mathit{MEv}_n(E) \text{ are compatible}} \end{array}}{\begin{array}{r} \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{amalgamation UNIT-TERM}_1\texttt{,...,UNIT-TERM}_n \Rightarrow \\ \lambda E \in C \cdot \mathit{MEv}_1(E) \oplus \ldots \oplus \mathit{MEv}_k(E) \end{array}}$$

Local Units

```
LOCAL-UNIT ::= local-unit UNIT-DEFN+ UNIT-TERM
```

$$\boxed{\begin{array}{c} \Gamma_s, C_s \vdash \texttt{LOCAL-UNIT} \rhd \Sigma \qquad\qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{LOCAL-UNIT} \Rightarrow \mathit{MEv} \\ \Gamma_s, \mathcal{C}_s \vdash \texttt{LOCAL-UNIT} \rhd\!\!\!\rhd (p, D) \end{array}}$$

See the requirements for the semantics of UNIT-TERM.

$$\frac{\begin{array}{c} \Gamma_s, \mathcal{C}_s \vdash \texttt{UNIT-DEFN}_1 \rhd\!\!\!\rhd (\mathcal{C}_s)_1 \\ \cdots \\ \Gamma_s, \mathcal{C}_s + (\mathcal{C}_s)_1 + \ldots + (\mathcal{C}_s)_{n-1} \vdash \texttt{UNIT-DEFN}_n \rhd\!\!\!\rhd (\mathcal{C}_s)_n \\ \Gamma_s, \mathcal{C}_s + (\mathcal{C}_s)_1 + \ldots + (\mathcal{C}_s)_n \vdash \texttt{UNIT-TERM} \rhd\!\!\!\rhd (p, D) \end{array}}{\Gamma_s, \mathcal{C}_s \vdash \texttt{local-unit UNIT-DEFN}_1\texttt{,...,UNIT-DEFN}_n\ \texttt{UNIT-TERM} \rhd\!\!\!\rhd (p, D)}$$

Unit Applications

```
UNIT-APPL ::= unit-appl UNIT-NAME FIT-ARG-UNIT*
FIT-ARG-UNIT ::= fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*
```

$$\boxed{\begin{array}{c} \Gamma_s, C_s \vdash \texttt{UNIT-APPL} \rhd \Sigma \qquad\qquad \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{UNIT-APPL} \Rightarrow \mathit{MEv} \\ \Gamma_s, C_s \vdash \texttt{UNIT-APPL} \mathrel{\rhd\!\!>} (p, D) \end{array}}$$

See the requirements for the semantics of `UNIT-TERM`.

$$\frac{B_s(\mathit{UN}) = p}{\Gamma_s, (P_s, B_s, D) \vdash \texttt{unit-appl}\ \ \mathit{UN} \mathrel{\rhd\!\!>} (p, D)}$$

$$\frac{\begin{array}{c} \mathcal{C}_s = (P_s, B_s, D) \\ P_s(\mathit{UN}) = (p^I, (\Sigma_1, \ldots, \Sigma_n \to \Sigma)) \\ \Sigma^F = D(p^I) \cup \Sigma_1 \cup \ldots \cup \Sigma_n \\ \Sigma_1, \Gamma_s, \mathcal{C}_s \vdash \texttt{FIT-ARG-UNIT}_1 \mathrel{\rhd\!\!>} (\sigma_1{:}\Sigma_1 \to \Sigma_1^A, p_1^A, D_1) \\ \ldots \\ \Sigma_n, \Gamma_s, \mathcal{C}_s \vdash \texttt{FIT-ARG-UNIT}_n \mathrel{\rhd\!\!>} (\sigma_n{:}\Sigma_n \to \Sigma_n^A, p_n^A, D_n) \\ D_1, \ldots, D_k \text{ disjointly extend } D \qquad D^A = D_1 \cup \ldots \cup D_k \\ \Sigma^A = D(p^I) \cup \Sigma_1^A \cup \ldots \cup \Sigma_n^A \qquad \sigma^A = (id_{D(p^I)} \cup \sigma_1 \cup \ldots \cup \sigma_n) \colon \Sigma^F \to \Sigma^A \\ \sigma^A(\Delta) \colon \Sigma \to (\Sigma^A \cup \Sigma^A(\Delta)), \text{ where } \Delta{:}\Sigma^F \to \Sigma \text{ is the signature extension} \\ \Sigma^R = \Sigma^A \cup \Sigma^A(\Delta) \\ D^A \text{ ensures amalgamability along} \\ (\Sigma^A, \langle \iota_{D(p^I) \subseteq \Sigma^A} \colon D(p^I) \to \Sigma^A, \iota_{\Sigma_i^A \subseteq \Sigma^A} \colon \Sigma_i^A \to \Sigma^A \rangle_{i=1,\ldots,n}) \\ D' \text{ extends } D^A \text{ by new node } q^B, \text{ edge } e^I \colon p^I \to q^B \text{ with } D'(e^I) = \iota_{D(p^I) \subseteq \Sigma}, \\ \text{nodes } p_i^F \text{ and edges } e_i^F \colon p_i^F \to q^B \text{ with } D'(e_i^F) = \iota_{\Sigma_i \subseteq \Sigma} \\ \text{and } e_i \colon p_i^F \to p_i^A \text{ with } D'(e_i) = \sigma_i, \text{ for } i = 1, \ldots, n, \\ D' \text{ ensures amalgamability along} \\ (\Sigma^R, \langle \sigma^A(\Delta) \colon \Sigma \to \Sigma^R, \iota_{\Sigma_i^A \subseteq \Sigma^R} \colon \Sigma_i^A \to \Sigma^R \rangle_{i=1,\ldots,n}) \\ D'' \text{ extends } D' \text{ by new node } q, \text{ edge } e' \colon q^B \to q \text{ with } D''(e') = \sigma^A(\Delta) \\ \text{and edges } e'_i \colon p_i^A \to q \text{ with } D''(e'_i) = \iota_{\Sigma_i^A \subseteq \Sigma^R}, \text{ for } i = 1, \ldots, n \end{array}}{\Gamma_s, \mathcal{C}_s \vdash \texttt{unit-appl}\ \ \mathit{UN}\ \ \texttt{FIT-ARG-UNIT}_1\texttt{,...,FIT-ARG-UNIT}_n \mathrel{\rhd\!\!>} (q, D'')}$$

See Sect. 4.1.3 for the definition of $\sigma^A(\Delta)$, the extension of a signature morphism along a signature extension.

Assuming that the extended static analysis is successful for the phrase `unit-appl` UN $\texttt{FIT-ARG-UNIT}_1\texttt{,...,FIT-ARG-UNIT}_n$, we can simplify the corresponding rule in the model semantics as in Fig. 5.1.

$$\begin{array}{c}
C_s(UN) = (\Sigma^I, (\Sigma_1, \ldots, \Sigma_n \to \Sigma)) \\
\Sigma^F = \Sigma^I \cup \Sigma_1 \cup \ldots \cup \Sigma_n \\
\Sigma_1, \Gamma_s, C_s \vdash \texttt{FIT-ARG-UNIT}_1 \rhd \sigma_1{:}\Sigma_1 \to \Sigma_1^A \\
\Sigma_1, \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{FIT-ARG-UNIT}_1 \Rightarrow MEv_1 \\
\ldots \\
\Sigma_n, \Gamma_s, C_s \vdash \texttt{FIT-ARG-UNIT}_n \rhd \sigma_n{:}\Sigma_n \to \Sigma_n^A \\
\Sigma_n, \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{FIT-ARG-UNIT}_n \Rightarrow MEv_n \\
\text{for all } E \in C, \langle MEv_1(E)|_{\sigma_1}, \ldots, MEv_n(E)|_{\sigma_n} \rangle \in Dom(E(UN)) \\
\Sigma^A = \Sigma^I \cup \Sigma_1^A \cup \ldots \cup \Sigma_n^A \qquad \sigma^A = (id_{\Sigma^I} \cup \sigma_1 \cup \ldots \cup \sigma_n) \colon \Sigma^F \to \Sigma^A \\
\sigma^A(\Delta) \colon \Sigma \to (\Sigma^A \cup \Sigma^A(\Delta)), \text{ where } \Delta{:}\Sigma^F \to \Sigma \text{ is the signature extension}
\end{array}$$

~~for all $E \in C$, there exists a unique $M \in \mathbf{Mod}(\Sigma^A \cup \Sigma^A(\Delta))$~~
~~such that $M|_{\Sigma_1^A} = MEv_1(E), \ldots, M|_{\Sigma_n^A} = MEv_n(E)$ and~~
~~$M|_{\sigma^A(\Delta)} = E(UN)\langle MEv_1(E)|_{\sigma_1}, \ldots, MEv_n(E)|_{\sigma_n} \rangle$~~

$$\begin{array}{l}
MEv = \{E \mapsto M \mid E \in C, M \in \mathbf{Mod}(\Sigma^A \cup \Sigma^A(\Delta)), \\
\qquad\qquad M|_{\sigma^A(\Delta)} = E(UN)\langle MEv_1(E)|_{\sigma_1}, \ldots, MEv_n(E)|_{\sigma_n} \rangle, \\
\qquad\qquad M|_{\Sigma_1^A} = MEv_1(E), \ldots, M|_{\Sigma_n^A} = MEv_n(E)\} \\
\hline
\Gamma_s, \Gamma_m, C_s, C \vdash \\
\qquad \texttt{unit-appl } UN \ \texttt{FIT-ARG-UNIT}_1\texttt{,..., FIT-ARG-UNIT}_n \Rightarrow MEv
\end{array}$$

Fig. 5.1. Simplified model semantics rule for unit applications

Note also that the verification of the condition

$$\text{for all } E \in C, \langle MEv_1(E)|_{\sigma_1}, \ldots, MEv_n(E)|_{\sigma_n} \rangle \in Dom(E(UN))$$

may be somewhat simplified here – the extended static analysis ensures the compatibility of the actual parameters with the imports (implicitly required here) used to define the generic unit.

$$\boxed{\begin{array}{l}
\Sigma, \Gamma_s, C_s \vdash \texttt{FIT-ARG-UNIT} \rhd \sigma{:}\Sigma \to \Sigma^A \\
\qquad\qquad\qquad\qquad\qquad\qquad \Sigma, \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{FIT-ARG-UNIT} \Rightarrow MEv^A \\
\qquad\qquad\qquad \Sigma, \Gamma_s, \mathcal{C}_s \vdash \texttt{FIT-ARG-UNIT} \mathrel{\rhd\!\!\rhd} (\sigma{:}\Sigma \to \Sigma^A, p, D)
\end{array}}$$

Γ_s and Γ_m are compatible global environments; C is a unit context that is compatible with static unit context C_s; then $\sigma \colon \Sigma \to \Sigma^A$ is a signature morphism, and Σ^A and MEv^A are compatible additions to C_s and C.

D extends $dgm(\mathcal{C}_s)$, $p \in Nodes(D)$, and $D(p) = \Sigma^A$.

If $\Sigma, \Gamma_s, \mathcal{C}_s \vdash \texttt{FIT-ARG-UNIT} \mathrel{\rhd\!\!\rhd} (\sigma{:}\Sigma \to \Sigma^A, p, D)$ then $\Sigma, \Gamma_s, ctx(\mathcal{C}_s) \vdash \texttt{FIT-ARG-UNIT} \rhd \sigma{:}\Sigma \to \Sigma^A$. Moreover, if $\Sigma, \Gamma_s, \Gamma_m, C_s, C \vdash \texttt{FIT-ARG-UNIT} \Rightarrow MEv^A$ and C is compatible with $\mathcal{C}_s$ then MEv^A and (p, Σ^A, D) are compatible additions to C and $\mathcal{C}_s$.

$$\frac{\begin{array}{c}\mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{UNIT-TERM} \rhd\!\!\!\rhd (p, D)\\ \vdash \texttt{SYMB-MAP-ITEMS*} \rhd r\end{array}}{\mathit{\Sigma}, \mathit{\Gamma}_s, \mathcal{C}_s \vdash \texttt{fit-arg-unit UNIT-TERM SYMB-MAP-ITEMS*} \rhd\!\!\!\rhd (r|^{\mathit{\Sigma}}_{D(p)}, p, D)}$$

See Sect. 4.1.3 for the definition of $r|^{\mathit{\Sigma}}_{D(p)}{:}\mathit{\Sigma}{\rightarrow}D(p)$, the signature morphism induced by the symbol map r.

The semantics of SYMB-MAP-ITEMS* is given in Sect. 4.5.2.

5.6.6 Discussion

The requirements stated above on the semantic judgements for each syntactic category provide the kernel of an inductive proof of the correctness of the extended static semantics: if the extended static semantics is successful then the simple static semantics of the previous sections is successful as well and yields corresponding results. Moreover, the requirements stated there justify the simplifications of the model semantics indicated above, given that the extended static semantics has been successful.

Of course, no completeness result can be expected: even when the simple static semantics and the model semantics are successful for a given phrase, the extended static semantics may fail: it additionally requires that the amalgamability conditions must be discharged on the basis of the information on sharing between symbols as stored in the signature diagram in the extended static context. This is stricter than the simple static and model semantics in two respects. First, the requirements imposed on units by their specifications are disregarded, and so symbols may be viewed as distinct even if these specifications ensure that they always have the same interpretation. Second, in order for two symbols to share here they must be traced to the same symbol in some (non-generic) unit declaration or definition. This implies that the symbols in the result of application of a generic unit to an argument share with the environment only via the argument (and the imports, if any). In particular, new symbols in the results of two applications of a generic unit do not share, even if the arguments coincide. This gives a so-called *generative* semantics of generic modules, and the corresponding generative type discipline is often adopted for module systems of programming languages (e.g., Standard ML, cf. [26]).

The generative and *applicative* (non-generative) semantics coincide if every generic unit is used at most once. For any architectural specification ARCH-SPEC, one can build its *generative version* $\widehat{\texttt{ARCH-SPEC}}$ as follows. For each generic unit F declared in ARCH-SPEC, let n_F be the number of applications of F used in unit terms in ARCH-SPEC. If $n_F > 1$ then we replace the declaration of F by declarations of n_F new units with distinct names and the same specification as F, and replace each application of F by a single application of one of the new units. Now, the (applicative) semantics of $\widehat{\texttt{ARCH-SPEC}}$ gives a *generative* semantics for ARCH-SPEC.

Then, let $|\widehat{\texttt{ARCH-SPEC}}|$ result from $\widehat{\texttt{ARCH-SPEC}}$ by removing all the axioms in all the specifications involved, so that all the specifications used in ARCH-SPEC are reduced to signatures in $|\widehat{\texttt{ARCH-SPEC}}|$. (This is very informal, but hopefully intuitively quite clear: a precise definition would have to go recursively through the specification as well as through the global environments it relies on.)

If the simple static semantics and model semantics are successful for both $\widehat{\texttt{ARCH-SPEC}}$ and $|\widehat{\texttt{ARCH-SPEC}}|$, and neither $\widehat{\texttt{ARCH-SPEC}}$ nor $|\widehat{\texttt{ARCH-SPEC}}|$ involves generic units with inconsistent specifications, then the extended static semantics and modified model semantics are successful on ARCH-SPEC as well and the results of the simple and extended semantics for ARCH-SPEC match. The key to this property is the fact that if axioms are removed from specifications then for any syntactic phrase, given the signature diagram and the unit context constructed for it, any model family that is compatible with this signature diagram may be obtained for a unit environment that fits the unit context. Then, by definition, the extra amalgamability requirements imposed by the extended static semantics coincide with the requirements eliminated from the original model semantics. See Chap. IV:5 for more details.

6

Specification Library Semantics

A *local library* in CASL is a named sequence of definitions, each of which names a (structured, generic, view, architectural, or unit) specification. The (static and model) semantics of such a definition compute extensions of given global environments, as defined in Chaps. 4 and 5. The semantics of a library consists of the name of the library, together with the global environment obtained by composing the extensions given by the semantics of its definitions, starting from the empty global environment.

CASL supports the installation of *distributed libraries* on the Internet, also allowing different versions of the same library. A library can either be identified by a URL that gives direct access to (all existing versions of) the library, or it can be registered with a hierarchical path name in a global directory, giving indirect access. In both cases, there may be more than one URL giving access to the same library (due to mirrors, caches, and redirections).

A so-called *downloading item* in a library refers to particular definitions that are supposed to be provided by a different library. The semantics of the downloading item extends the global environment (of the enclosing library) with part of the global environment given by the semantics of the referenced library (possibly providing different names for the downloaded items, e.g. to avoid clashes with names already in use in the enclosing library). A reference to another library can either specify a particular version, or leave the version open (in which case the version that has the largest version number is obtained). Downloading must not lead to cyclic chains of references.

The rest of this chapter gives a formal semantics for specification libraries, extending what was provided for basic, structured, and architectural specifications in Chaps. 2–5. Section 6.1 defines some semantic domains: global environments and directories, and universal environments. Section 6.2 defines the semantics of *local* libraries, and Sect. 6.3 extends the semantics to cover the downloadings used in *distributed* libraries. Finally, Sect. 6.4 deals with library names and versions.

6.1 Library Concepts

Specifications may be *named* by *definitions* and collected in *libraries*. In the context of a library, the (re)use of a specification may be replaced by a *reference* to it through its name. The current association between names and the specifications that they reference is called the *global environment*; the global environment for a named specification is determined exclusively by the definitions that precede it.

A *static global environment* $\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)$ consists of finite functions from names to the static denotations of generic specifications, views, architectural specifications and unit specifications:

- $\mathcal{G}_s : \mathit{SpecName} \overset{\text{fin}}{\to} \mathit{GenSig}$
- $\mathcal{V}_s : \mathit{ViewName} \overset{\text{fin}}{\to} \mathit{ViewSig}$
- $\mathcal{A}_s : \mathit{ArchSpecName} \overset{\text{fin}}{\to} \mathit{ArchSig}$
- $\mathcal{T}_s : \mathit{UnitSpecName} \overset{\text{fin}}{\to} \mathit{UnitSig}$

Similarly, a *model global environment* $\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)$ consists of finite functions from names to the model denotations of generic specifications, views, architectural specifications and unit specifications:

- $\mathcal{G}_m : \mathit{SpecName} \overset{\text{fin}}{\to} \mathbf{GenSpec}$
- $\mathcal{V}_m : \mathit{ViewName} \overset{\text{fin}}{\to} \mathbf{ViewSpec}$
- $\mathcal{A}_m : \mathit{ArchSpecName} \overset{\text{fin}}{\to} \mathbf{ArchSpec}$
- $\mathcal{T}_m : \mathit{UnitSpecName} \overset{\text{fin}}{\to} \mathbf{UnitSpec}$

The domains of the various components of static and model global environments are disjoint.

The semantic domains *GenSig*, **GenSpec**, *ViewSig*, **ViewSpec** are defined in Chap. 4, while *ArchSpecSig*, **ArchSpec**, *UnitSpecSig* and **UnitSpec** are defined in Chap. 5.

A static global environment $\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)$ and a model global environment $\Gamma_m = (\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)$ are *compatible* if for each component $\mathcal{F}$ of the global environment ($\mathcal{F} \in \{\mathcal{G}, \mathcal{V}, \mathcal{A}, \mathcal{T}\}$) and for each name $\mathit{Name} \in$ `SIMPLE-ID` either its static and model denotations are compatible, that is, $\mathcal{F}_s(\mathit{Name})$ and $\mathcal{F}_m(\mathit{Name})$ are compatible, or both $\mathcal{F}_s(\mathit{Name})$ and $\mathcal{F}_m(\mathit{Name})$ are undefined. Compatibility of static and model semantics is defined in the respective sections of Chaps. 4 and 5.

$$\begin{aligned} \mathit{IN} \in \mathit{ItemName} = \mathit{SpecName} = \mathit{ViewName} = {} & \\ & \mathit{ArchSpecName} = \mathit{UnitSpecName} = \texttt{SIMPLE-ID} \end{aligned}$$

A library may be located at a particular *site* on the Internet. The library is referenced from other sites by a name which determines the location and perhaps identifies a particular version of the library. To allow libraries to be relocated without this invalidating existing references to them, library names may be interpreted relative to a *global directory* that maps names to URLs. Libraries may also be referenced directly by their (relative or absolute) URLs, independently of their registration in the global directory. A library may incorporate the *downloading* of (the semantics of) named specifications from (perhaps particular versions of) other libraries, whenever the library is used.

The semantics of libraries involves *URLs*, *paths*, and *version numbers*:

$$\begin{aligned} &u \in \mathit{Url} \\ &p \in \mathit{Path} \\ &v \in \mathit{Version} = \mathit{FinSeq}(\mathit{Nat}) \end{aligned}$$

The internal structure of *Url* and *Path* is irrelevant in the semantics. Version numbers are ordered lexicographically: $\langle n_1, \ldots, n_j \rangle < \langle n'_1, \ldots, n'_k \rangle$ iff either

- there exists $i \leq j, k$ such that $n_i < n'_i$ and for all $l < i$, $n_l = n'_l$, or
- $j < k$ and for all $l \leq j$, $n_l = n'_l$.

A (canonical) *library name* *LN* is a library identifier *LI* (i.e. a URL or a path) together with a (possibly empty) version:

$$\begin{aligned} \mathit{LI} &\in \mathit{LibId} = \mathit{Url} \uplus \mathit{Path} \\ \mathit{LN} &\in \mathit{LibName} = \mathit{LibId} \times \mathit{Version} \end{aligned}$$

The empty version $\langle\rangle$ may only be used to name a library that exists in just one version: if a second version of the same library is installed, the two versions must both be distinguished by non-empty version numbers. (This is the only case where an already-installed version of a library can be given a new version number.) However, $\langle\rangle$ may always be used to refer to the version of a library that has the *largest* version number.

A *static universal environment* U_s is a finite function from URLs and versions to static global environments:

$$U_s \in \mathit{UnivEnv}_s = \mathit{Url} \times \mathit{Version} \overset{\text{fin}}{\to} \mathit{GlobalEnv}_s$$

such that for all $u \in \mathit{Url}$, when v is the largest version number with $(u, v) \in \mathit{Dom}(U_s)$, we have $U_s(u, \langle\rangle) = U_s(u, v)$.

Similarly, a *model universal environment* U_m is a finite function from URLs and versions to model global environments:

$$U_m \in \mathit{UnivEnv}_m = \mathit{Url} \times \mathit{Version} \overset{\text{fin}}{\to} \mathit{GlobalEnv}_m$$

such that for all $u \in \mathit{Url}$, when v is the largest version number with $(u, v) \in \mathit{Dom}(U_m)$, we have $U_m(u, \langle\rangle) = U_m(u, v)$.

A *global directory GD* is a finite function from paths and versions to URLs:

$$\mathit{GD} \in \mathit{GlobalDir} = \mathit{Path} \times \mathit{Version} \overset{\text{fin}}{\to} \mathit{Url}$$

such that for all $p \in \mathit{Path}$, when v is the largest version number with $(p, v) \in \mathit{Dom}(\mathit{GD})$, we have $\mathit{GD}(p, \langle\rangle) = \mathit{GD}(p, v)$.

U_s, U_m, and GD are *compatible* iff:

- $\mathit{Dom}(U_s) = \mathit{Dom}(U_m)$;
- for all $(u, v) \in \mathit{Dom}(U_s)$, the global environments $U_s(u, v)$ and $U_m(u, v)$ are compatible, as defined above; and
- for all $(p, v) \in \mathit{Dom}(\mathit{GD})$, $(\mathit{GD}(p, v), v) \in \mathit{Dom}(U_s)$.

6.2 Local Libraries

```
LIB-DEFN ::= lib-defn LIB-NAME LIB-ITEM*
LIB-ITEM ::= SPEC-DEFN | VIEW-DEFN | ARCH-SPEC-DEFN | UNIT-SPEC-DEFN
```

A library definition `LIB-DEFN` provides a collection of specification (and perhaps also view) definitions. It is well-formed only when the defined names are distinct, and not referenced until (strictly) after their definitions. The global environment for each definition is that determined by the preceding definitions. Thus a library in Casl provides linear visibility, and mutual or cyclic chains of references are not allowed.

The semantics of distributed libraries in the next section involves universal environments U_s, U_m that map URLs and versions to global environments, as well as a global directory GD that maps paths and versions to URLs, as defined in Sect. 6.1. These components are incorporated but ignored by the semantics of local library items.

The effect of processing a library definition may depend not only on U_s, U_m, and GD, but also on the URL at which the library is to be located (and possibly registered). The details of library installation and registration are out of the scope of this semantics.

$$\boxed{U_s, \mathit{GD} \vdash \texttt{LIB-DEFN} \rhd (\mathit{LN}, \Gamma_s) \qquad U_s, U_m, \mathit{GD} \vdash \texttt{LIB-DEFN} \Rightarrow (\mathit{LN}, \Gamma_m)}$$

U_s, U_m, and GD are required to be compatible. Γ_s and Γ_m are then compatible too.

$$\frac{\begin{array}{c}\vdash \texttt{LIB-NAME} \rhd \mathit{LN} \qquad \mathit{versionOK}(U_s, \mathit{GD}, \mathit{LN}) \\ U_s, \mathit{GD}, \emptyset \vdash \texttt{LIB-ITEM*} \rhd \Gamma'_s\end{array}}{U_s, \mathit{GD} \vdash \texttt{lib-defn LIB-NAME LIB-ITEM*} \rhd (\mathit{LN}, \Gamma'_s)}$$

where for all $u \in \mathit{Url}, v \in \mathit{Version}$, $\mathit{versionOK}(U_s, \mathit{GD}, (u,v))$ means

$$v = \langle\rangle \text{ implies } \{v' \mid (u,v') \in \mathit{Dom}(U_s)\} = \{\langle\rangle\}$$

and for all $p \in \mathit{Path}, v \in \mathit{Version}$, $\mathit{versionOK}(U_s, \mathit{GD}, (p,v))$ means

$$v = \langle\rangle \text{ implies } \{v' \mid (p,v') \in \mathit{Dom}(\mathit{GD})\} = \{\langle\rangle\}$$

$$\frac{\begin{array}{c}\vdash \texttt{LIB-NAME} \rhd \mathit{LN} \qquad \mathit{versionOK}(U_m, \mathit{GD}, \mathit{LN}) \\ U_s, U_m, \mathit{GD}, \emptyset, \emptyset \vdash \texttt{LIB-ITEM*} \Rightarrow \mathit{\Gamma}'_m\end{array}}{U_s, U_m, \mathit{GD} \vdash \texttt{lib-defn LIB-NAME LIB-ITEM*} \Rightarrow (\mathit{LN}, \mathit{\Gamma}'_m)}$$

where for all $u \in \mathit{Url}, v \in \mathit{Version}$, $\mathit{versionOK}(U_m, \mathit{GD}, (u,v))$ means

$$v = \langle\rangle \text{ implies } \{v' \mid (u,v') \in \mathit{Dom}(U_m)\} = \{\langle\rangle\}$$

and for all $p \in \mathit{Path}, v \in \mathit{Version}$, $\mathit{versionOK}(U_m, \mathit{GD}, (p,v))$ means

$$v = \langle\rangle \text{ implies } \{v' \mid (p,v') \in \mathit{Dom}(\mathit{GD})\} = \{\langle\rangle\}$$

$$\boxed{U_s, \mathit{GD}, \mathit{\Gamma}_s \vdash \texttt{LIB-ITEM} \rhd \mathit{\Gamma}'_s \qquad U_s, U_m, \mathit{GD}, \mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{LIB-ITEM} \Rightarrow \mathit{\Gamma}'_m}$$

U_s, U_m, and GD are required to be compatible; so are $\mathit{\Gamma}_s$ and $\mathit{\Gamma}_m$. Then $\mathit{\Gamma}'_s$ and $\mathit{\Gamma}'_m$ are compatible, and extend $\mathit{\Gamma}_s$, resp. $\mathit{\Gamma}_m$.

$$\frac{\mathit{\Gamma}_s \vdash \texttt{SPEC-DEFN} \rhd \mathit{\Gamma}'_s}{U_s, \mathit{GD}, \mathit{\Gamma}_s \vdash \texttt{SPEC-DEFN}\,\text{qua}\,\texttt{LIB-ITEM} \rhd \mathit{\Gamma}'_s}$$

$$\frac{\mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{SPEC-DEFN} \Rightarrow \mathit{\Gamma}'_m}{U_s, U_m, \mathit{GD}, \mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{SPEC-DEFN}\,\text{qua}\,\texttt{LIB-ITEM} \Rightarrow \mathit{\Gamma}'_m}$$

Similar rules for VIEW-DEFN, ARCH-SPEC-DEFN, and UNIT-SPEC-DEFN are elided. The semantics of SPEC-DEFN and VIEW-DEFN is defined in Chap, 4, and the semantics of ARCH-SPEC-DEFN and UNIT-SPEC-DEFN is defined in Chap, 5.

$$\boxed{U_s, \mathit{GD}, \mathit{\Gamma}_s \vdash \texttt{LIB-ITEM*} \rhd \mathit{\Gamma}'_s \qquad U_s, U_m, \mathit{GD}, \mathit{\Gamma}_s, \mathit{\Gamma}_m \vdash \texttt{LIB-ITEM*} \Rightarrow \mathit{\Gamma}'_m}$$

U_s, U_m, and GD are required to be compatible; so are $\mathit{\Gamma}_s$ and $\mathit{\Gamma}_m$. Then $\mathit{\Gamma}'_s$ and $\mathit{\Gamma}'_m$ are compatible, and extend $\mathit{\Gamma}_s$, resp. $\mathit{\Gamma}_m$.

$$\frac{\begin{array}{c}U_s, \mathit{GD}, (\mathit{\Gamma}_s)_0 \vdash \texttt{LIB-ITEM}_1 \rhd (\mathit{\Gamma}_s)_1 \\ \dots \\ U_s, \mathit{GD}, (\mathit{\Gamma}_s)_{n-1} \vdash \texttt{LIB-ITEM}_n \rhd (\mathit{\Gamma}_s)_n\end{array}}{U_s, \mathit{GD}, (\mathit{\Gamma}_s)_0 \vdash \texttt{LIB-ITEM}_1\ \dots\ \texttt{LIB-ITEM}_n \rhd (\mathit{\Gamma}_s)_n}$$

$$\frac{\begin{array}{c} U_s, GD, (\Gamma_s)_0 \vdash \texttt{LIB-ITEM}_1 \rhd (\Gamma_s)_1 \\ U_s, U_m, GD, (\Gamma_s)_0, (\Gamma_m)_0 \vdash \texttt{LIB-ITEM}_1 \Rightarrow (\Gamma_m)_1 \\ \dots \\ U_s, GD, (\Gamma_s)_{n-1} \vdash \texttt{LIB-ITEM}_n \rhd (\Gamma_s)_n \\ U_s, U_m, GD, (\Gamma_s)_{n-1}, (\Gamma_m)_{n-1} \vdash \texttt{LIB-ITEM}_n \Rightarrow (\Gamma_m)_n \end{array}}{U_s, U_m, GD, (\Gamma_s)_0, (\Gamma_m)_0 \vdash \texttt{LIB-ITEM}_1 \ \dots \ \texttt{LIB-ITEM}_n \Rightarrow (\Gamma_m)_n}$$

6.3 Distributed Libraries

```
LIB-ITEM          ::= ... | DOWNLOAD-ITEMS
DOWNLOAD-ITEMS    ::= download-items LIB-NAME ITEM-NAME-OR-MAP+
ITEM-NAME-OR-MAP  ::= ITEM-NAME | ITEM-NAME-MAP
ITEM-NAME-MAP     ::= item-name-map ITEM-NAME ITEM-NAME
ITEM-NAME         ::= SIMPLE-ID
```

The ITEM-NAME-OR-MAP in a DOWNLOAD-ITEMS determines a selection and possible renaming of definitions from the named library, resulting in a global environment to be added to the current global environment. The following rules complete the definition of the semantics of library items, initiated in Sect. 6.2.

$$\frac{\begin{array}{c} \vdash \texttt{LIB-NAME} \rhd (u, v) \qquad\qquad (u, v) \in \mathit{Dom}(U_s) \\ U_s(u, v) \vdash \texttt{ITEM-NAME-OR-MAP+} \rhd \Gamma'_s \end{array}}{U_s, GD, \Gamma_s \vdash \texttt{download-items LIB-NAME ITEM-NAME-OR-MAP+} \rhd \Gamma_s \cup \Gamma'_s}$$

$$\frac{\begin{array}{c} \vdash \texttt{LIB-NAME} \rhd (p, v) \qquad (p, v) \in \mathit{Dom}(GD) \qquad (GD(p, v), v) \in \mathit{Dom}(U_s) \\ U_s(GD(p, v), v) \vdash \texttt{ITEM-NAME-OR-MAP+} \rhd \Gamma'_s \end{array}}{U_s, GD, \Gamma_s \vdash \texttt{download-items LIB-NAME ITEM-NAME-OR-MAP+} \rhd \Gamma_s \cup \Gamma'_s}$$

The rules for the model semantics are elided.

$$\boxed{\Gamma_s \vdash \texttt{ITEM-NAME-OR-MAP} \rhd \Gamma'_s \qquad\qquad \Gamma_m \vdash \texttt{ITEM-NAME-OR-MAP} \Rightarrow \Gamma'_m}$$

$$\frac{\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \qquad\qquad \mathit{IN} \in \mathit{Dom}(\mathcal{G}_s)}{\Gamma_s \vdash \mathit{IN} \text{ qua } \texttt{ITEM-NAME-OR-MAP} \rhd (\{\mathit{IN} \mapsto \mathcal{G}_s(\mathit{IN})\}, \emptyset, \emptyset, \emptyset)}$$

$$\frac{\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \qquad\qquad \mathit{IN}_1 \in \mathit{Dom}(\mathcal{G}_s)}{\Gamma_s \vdash \texttt{item-name-map } \mathit{IN}_1 \ \mathit{IN}_2 \rhd (\{\mathit{IN}_2 \mapsto \mathcal{G}_s(\mathit{IN}_1)\}, \emptyset, \emptyset, \emptyset)}$$

The rules for the model semantics of ITEM-NAME-OR-MAP are elided, as are those for item names which refer to views, architectural specifications, and unit specifications.

$$\boxed{\Gamma_s \vdash \texttt{ITEM-NAME-OR-MAP+} \rhd \Gamma'_s \qquad \Gamma_m \vdash \texttt{ITEM-NAME-OR-MAP+} \Rightarrow \Gamma'_m}$$

Γ'_s and Γ'_m correspond to subsets of Γ_s, resp. Γ_m, except that some of the item names may have been replaced.

$$\frac{\begin{array}{c}\Gamma_s \vdash \texttt{ITEM-NAME-OR-MAP}_1 \rhd (\Gamma'_s)_1 \\ \dots \\ \Gamma_s \vdash \texttt{ITEM-NAME-OR-MAP}_n \rhd (\Gamma'_s)_n\end{array}}{\Gamma_s \vdash \texttt{ITEM-NAME-OR-MAP}_1 \ \dots \ \texttt{ITEM-NAME-OR-MAP}_n \rhd (\Gamma'_s)_1 \cup \dots \cup (\Gamma'_s)_n}$$

The rules for the model semantics of `ITEM-NAME-OR-MAP+` are elided.

6.4 Library Names

```
LIB-NAME       ::= LIB-ID | LIB-VERSION
LIB-VERSION    ::= lib-version LIB-ID VERSION-NUMBER

VERSION-NUMBER ::= version-number NUMBER+

LIB-ID        ::= DIRECT-LINK | INDIRECT-LINK
DIRECT-LINK   ::= direct-link URL
INDIRECT-LINK ::= indirect-link PATH
```

The following judgements provide canonical library names.

$$\boxed{\vdash \texttt{LIB-NAME} \rhd LN}$$

$$\frac{\vdash \texttt{LIB-ID} \rhd LI}{\vdash \texttt{LIB-ID}\,\text{qua}\,\texttt{LIB-NAME} \rhd (LI, \langle\rangle)}$$

$$\frac{\vdash \texttt{LIB-ID} \rhd LI \qquad \vdash \texttt{VERSION-NUMBER} \rhd v}{\vdash \texttt{lib-version LIB-ID VERSION-NUMBER} \rhd (LI, v)}$$

$$\boxed{\vdash \texttt{VERSION-NUMBER} \rhd v}$$

v is a non-empty sequence of natural numbers.

$$\frac{\texttt{NUMBER}_i \text{ is decimal notation for } n_i,\ i = 1, \dots, m}{\vdash \texttt{version-number NUMBER}_1 \ \dots \ \texttt{NUMBER}_m \rhd \langle n_1, \dots, n_m \rangle}$$

$$\boxed{\vdash \texttt{LIB-ID} \rhd LI}$$

$$\frac{}{\vdash \texttt{direct-link}\ u \rhd u} \qquad \frac{}{\vdash \texttt{indirect-link}\ p \rhd p}$$

Part IV

Casl Logic

Till Mossakowski
Piotr Hoffman
Serge Autexier
Dieter Hutter

Editor: Till Mossakowski

P.D. Mosses (Ed.): CASL Reference Manual, LNCS 2960, pp. 273–359, 2004.

1

Introduction

This part of the CASL Reference Manual provides proof calculi for the various levels of CASL specifications. It should be read together with the the CASL semantics (Part III).

The aim of the CASL proof calculus is to support the users of CASL in the proof activities necessary in the process of software specification and development. Essentially, the goals are threefold. First, in a number of situations the model semantics for a CASL specification may fail even if the static semantics succeeds. This is the case for instance when a generic specification is instantiated with an actual parameter (which must then entail the formal parameter specification), when a view between two specifications is formed, or when a generic unit is applied to an argument in an architectural specification. One aim of the calculi developed here is to (make explicit and help the user to) discharge proof obligations such situations imply. Once this is done, we can be sure that the specification in question denotes a class of models. Then, the second aim of the calculi developed here is to prove consequences of such specifications – formulas that hold in all the models. Finally, this can be used to prove various relationships between CASL specifications.

This program is carried through the various layers of CASL. For basic specifications, no proof obligations arise, and proving their consequences amounts to proving consequences of a set of logical formulas. The corresponding calculus for the logic of CASL is given in Chap. 2. Actually, the calculus is for the many-sorted sublanguage of CASL, but via the reduction of subsorted CASL to many-sorted CASL given in the semantics (Chap. III:3), it can also be used for subsorted CASL, see Chap. 3.

Structured specifications are treated in Chap. 4. Since CASL structured specifications are quite complex, a simpler core formalism, called *development graphs*, is introduced. A development graph consists of a set of nodes (corresponding to whole structured specifications or parts thereof), and a set of arrows called definition links, indicating the dependency of each involved structured specification on its subparts. The proof calculus is given for development graphs, which makes the calculus much simpler than a calculus that would directly act on CASL structured specifications. The proof calculus makes use of the calculus for basic specifications. The link between CASL structured specifications and development graphs is given by a *verification semantics*. It is similar to the static semantics of structured specifications, but simultaneously extracts a development graph and a set of proof obligations (the latter are called theorem links). The proof obligations arise e.g. from instantiations of generic specifications. Once they have been extracted, they can be tackled with the proof calculus. Indeed, the proof obligations can be discharged if and only if the model semantics of the structured specification under consideration succeeds. The proof calculus can also be used to prove intended consequences of structured specifications or check their consistency. In fact, for this purpose, there are CASL annotations such as **%implies** and **%cons**, and the verification semantics leads to appropriate further proof obligations (i.e. going beyond those capturing the model semantics).

For the sake of simplicity, only a sublanguage of architectural specifications is covered in Chap. 5. The sublanguage is introduced with syntax, (extended) static and model semantics. The relation between extended static and model semantics is studied, continuing the discussion in Sect. III:5.6.6. Based on the calculus for structured specifications, a calculus is developed that can be used for proving that a given architectural specification has a denotation w.r.t. its model semantics (i.e., is correct) and that the units produced using it satisfy a given unit specification.

Finally, Chap. 6 points out how the various calculi may be integrated in order to obtain a calculus for proving the correctness of whole libraries.

1.1 Institution Independence

The proof calculi for structured and architectural specifications and libraries are independent of the framework that is used for basic specifications (this is similar to the institution independence discussed in Sect. III:4.1). The semantics of basic specification defines an *institution* [20], while the proof calculus for basic specifications extends this to a *logic* [34] (which is an institution equipped with a proof theoretic entailment relation). Hence, the proof calculi for structured and architectural specifications are parametrized over an arbitrary logic. The logic is required to fulfill some mild technical conditions.

The actual subdivision between institution dependent and institution independent calculi is not quite as clean as stated above. Namely, despite the fact that the calculus for structured specifications is institution independent in general, at some points, institution dependent proof rules are needed. This is due to the need to deal with free specifications and with checks for conservativity of extensions[1]. Up to now, these two problems have not been treated

[1] Note that checks for conservativity of extensions not only arise from explicit conservativity annotations; they may also arise during a proof of a proof obligation generated by a view whose source involves hiding.

in a logic independent way, and such a treatment seems to be rather difficult. Hence, one needs logic-specific rules at these places (see Sect. 4.6).

1.2 Style of the Proof Calculi

The proof calculi follow a natural deduction style. This makes the rules compact and easy to read. However, in the calculus for basic specifications, side conditions such as Eigenvariable conditions are not so easy to understand, compared to a Gentzen style presentation. We therefore indicate at some places how a Gentzen style presentation would look.

1.3 Soundness and Completeness

We prove the soundness of all the calculi for the different layers of Casl, which shows that only logical consequences can be proved. The converse property, that all logical consequences can be proved, is known as completeness. Unfortunately, the presented calculi are not complete, and they cannot be so in principle. There are different sources of incompleteness:

- sort generation constraints in Casl basic specifications,
- Casl structured specifications involving hiding and freeness (the former need some form of check for conservativity of extensions), and
- consistency checks in Casl architectural specifications.

For these constructs of Casl, there cannot be a recursively axiomatized complete calculus. However, we prove relative completeness results in the sense that if these constructs are omitted or if there is an oracle that deals with them, we obtain a complete calculus. Even then, completeness of the logic independent proof calculus for structured and architectural specifications of course requires completeness of the proof calculus of the logic for basic specifications.

Acknowledgement. The authors of the various chapters are:

Basic specifications: Till Mossakowski
Subsorting specifications: Till Mossakowski
Structured specifications: Till Mossakowski, Serge Autexier and Dieter Hutter
Architectural specifications: Piotr Hoffman
Libraries: Till Mossakowski

This document was assembled by Till Mossakowski. Proofreading was done by Andrzej Tarlecki.

This research was partly supported by CoFI-WG (ESPRIT Working Group 29432). Till Mossakowski's work was partly supported by the *Deutsche Forschungsgemeinschaft* under grant KR 1191/5-1. Piotr Hoffman's work was partly supported by the Polish State Committee for Research (KBN) under grant no. 7 T11C 002 21.

2

Basic Specification Calculus

Here, we define a proof calculus for many-sorted basic specifications as introduced in Chap. I:2. In Chap. 3 below, also subsorted specifications are treated.

The semantics of a basic specification is a signature together with a set of axioms; see Chap. III:2. The proof calculus for basic specifications allows for deriving consequences from such a set of sentences. More specifically, the proof calculus is given by a set of rules, which together generate a proof-theoretic entailment relation. The proof calculus is shown to be sound, i.e. only logical consequences can be derived in the calculus. The converse, completeness, cannot be achieved due to the presence of sort generation constraints; however, the calculus is complete if the latter are omitted.

Before we present the proof calculus, a few remarks on the syntax of formulas as defined in Sect. III:2.1.3 are in order. As stated there, we restrict ourselves to a kernel language consisting of predicate applications, existential equations, false, implication, and universal quantification – all the other types of equations, connectives and quantifiers can be expressed in terms of these. Moreover, we follow the remark stated there that it is possible to omit conditional terms of the form $\varphi \leftarrow t'|t''$; these can be eliminated as described in Sect. I:2.5.4 of the CASL Summary. Finally, for the sake of readability, we will deviate from the notation for formulas that has been introduced in the semantics of many-sorted basic specifications: within quantifications, instead of $\forall x_s.\varphi$ we write $\forall x : s \,.\, \varphi$, and we omit the sorts of variables and profiles of function and predicate symbols, if these are clear from the context.

We now introduce some auxiliary notions about substitutions that are needed in the calculus rules. In the sequel, fix a many-sorted signature $\Sigma = (S, TF, PF, P)$.

Given an S-sorted variable systems X, the S-sorted set of Σ-terms over X is denoted by $T_\Sigma(X)$.

Definition 2.1. *Given an S-sorted variable systems X and Y, an S-sorted function $\nu\colon X \to T_\Sigma(Y)$ is called a* substitution. *Given a substitution $\nu\colon X \to$*

$T_\Sigma(Y)$ and an S-sorted variable system Z, we denote by $\nu \setminus Z \colon X \cup Z \to T_\Sigma(Y \cup Z)$ the substitution being the identity on Z and being ν on $X \setminus Z$.

We now come to the definition of what it means to apply a substitution to a term or a formula. The application of a substitution to a formula is not defined in all cases because of the variable capture problem: if a term that is substituted for a variable contains free variables, the latter must not newly get into the scope of a quantifier in the term or formula resulting from the substitution.

Definition 2.2. *The term $t[\nu] \in T_\Sigma(Y)$ resulting from applying the substitution ν to a term $t \in T_\Sigma(X)$ is defined by*

- $x[\nu] = \nu_s(x)$ *for* $x \in X_s$
- $f_{ws}(t_1, \ldots, t_n)[\nu] = f_{ws}(t_1[\nu], \ldots, t_n[\nu])$

Given a Σ-formula φ over X, the formula $\varphi[\nu]$, which is either undefined or a Σ-formula over Y resulting from applying the substitution ν to φ, is defined inductively over φ:

- $(t_1 \overset{e}{=} t_2)[\nu] \;=\; t_1[\nu] \overset{e}{=} t_2[\nu]$
- $p_w(t_1, \ldots, t_n)[\nu] \;=\; p_w(t_1[\nu], \ldots, t_n[\nu])$
- $\mathit{false}[\nu] \;=\; \mathit{false}$
- $(\varphi \Rightarrow \psi)[\nu] \;=\; (\varphi[\nu]) \Rightarrow (\psi[\nu])$
- $(\forall z : s' \,.\, \varphi)[\nu] \;=\; \begin{cases} \forall z : s' \,.\, (\varphi[\nu \setminus \{z_{s'}\}]), & \text{if for all } x \in X_s, s \in S, \\ & \quad x[\nu] \neq x \text{ and } x \in FV(\forall z : s' \,.\, \varphi) \\ & \quad \text{imply } z \notin FV(x[\nu]) \\ \text{undefined}, & \text{otherwise} \end{cases}$

The last case causes $(\forall z : s' \,.\, \varphi)[\nu]$ to be undefined if a name clash occurs (where a name clash means that a free variable in $x[\nu]$ becomes bound by the quantification over $z_{s'}$). This restriction is important to keep the intended semantics of substitutions.

The rules of derivation are given in Fig. 2.1 (the first-order rules) and Fig. 2.2 (the induction rules). They are given in a natural deduction style. Some rules such as **(⇒-intro)** need local assumptions, which means that their premises are of the form 'if ψ can be derived from φ' (here, φ is the local assumption). In the calculus, this is written $\begin{array}{c}[\varphi]\\ \vdots \\ \psi\end{array}$.

The rule **(Congruence)** captures the usual congruence (take $\nu(x_s)$ to be a variable) as well as symmetry and transitivity of existential equality.

Recall from Sect. III:2.1.3 that $D(t)$ abbreviates $t \overset{e}{=} t$, and $\varphi \wedge \psi$ is defined as a complicated term using $\Rightarrow$ and *false*. For simplicity, we here consider $(\bigwedge_{i=1,\ldots,n} \varphi_i) \Rightarrow \psi$ as an abbreviation for $\varphi_1 \Rightarrow \ldots \Rightarrow \varphi_n \Rightarrow \psi$, with $\Rightarrow$ grouping to the right.

For the induction rules (Fig. 2.2), without loss of generality we assume that for a sort generation constraint

$$\textbf{(Absurdity)}\ \frac{\mathit{false}}{\varphi} \qquad \textbf{(Tertium non datur)}\ \frac{\begin{array}{cc}[\varphi] & [\varphi \Rightarrow \mathit{false}]\\ \vdots & \vdots\\ \psi & \psi\end{array}}{\psi}$$

$$\textbf{(}\Rightarrow\textbf{-intro)}\ \frac{\begin{array}{c}[\varphi]\\ \vdots\\ \psi\end{array}}{\varphi \Rightarrow \psi} \qquad \textbf{(}\Rightarrow\textbf{-elim)}\ \frac{\begin{array}{c}\varphi\\ \varphi \Rightarrow \psi\end{array}}{\psi} \qquad \textbf{(}\forall\textbf{-elim)}\ \frac{\forall x\colon s\,.\,\varphi}{\varphi}$$

$$\textbf{(}\forall\textbf{-intro)}\ \frac{\varphi}{\forall x\colon s\,.\,\varphi}\ \text{where } x_s \text{ occurs freely only in local assumptions}$$

$$\textbf{(Reflexivity)}\ \frac{}{x_s \overset{e}{=} x_s}\ \text{if } x_s \text{ is a variable}$$

$$\textbf{(Congruence)}\ \frac{\varphi}{(\bigwedge_{x_s \in FV(\varphi)} x_s \overset{e}{=} \nu(x_s)) \Rightarrow \varphi[\nu]}\ \text{if } \varphi[\nu] \text{ defined}$$

$$\textbf{(Totality)}\ \frac{}{D(f_{w,s}(x_{s_1}, \ldots, x_{s_n}))}\ \text{if } w = s_1 \ldots s_n, f \in TF_{w,s}$$

$$\textbf{(Substitution)}\ \frac{\varphi}{(\bigwedge_{x_s \in FV(\varphi)} D(\nu(x_s))) \Rightarrow \varphi[\nu]}$$

if $\varphi[\nu]$ defined and $FV(\varphi)$ occur freely only in local assumptions

$$\textbf{(Function Strictness)}\ \frac{t_1 \overset{e}{=} t_2}{D(t)}\ t \text{ some subterm of } t_1 \text{ or } t_2$$

$$\textbf{(Predicate Strictness)}\ \frac{p_w(t_1, \ldots, t_n)}{D(t_i)}\ i \in \{1, \ldots, n\}$$

Fig. 2.1. First-order deduction rules for CASL basic specifications

$$(S', F', \theta\colon \bar{\Sigma} \to \Sigma),$$

all the result sorts of function symbols in F' occur in S'. If not, we can just leave out from F' those function symbols not satisfying this requirement. The satisfaction of the sort generation constraint in any model will not be affected by this: in the $\bar{\Sigma}$-term t that (jointly with an appropriate assignment of its variables) witnesses the satisfaction of the constraint, any application of a function symbol with result sort outside S' can just be replaced by a variable of that sort, which then gets the value of the function application as assigned value.

A *derivation* of $\Phi \vdash \varphi$ is a tree (called *derivation tree*) such that

- the root of the tree is φ,
- all the leaves of the tree are either in Φ or marked as local assumption,

$$\textbf{(Induction)}\quad \frac{\begin{array}{c}(S',F',\theta\colon \bar{\Sigma}\to\Sigma)\\ \varphi_1\wedge\cdots\wedge\varphi_k\end{array}}{\bigwedge_{s\in S'}\forall x:\theta(s)\,.\,\varPsi_s(x)}$$

$F'=\{f_1\colon s^1_1\dots s^1_{m_1}\to s^1;\ \dots;\ f_k\colon s^k_1\dots s^k_{m_k}\to s^k\}$,
$\varPsi_s$ is a formula with one free variable of sort $\theta(s)$, for $s\in S'$,

$$\begin{aligned}\varphi_j = \forall x_1:\theta(s^j_1),\dots,x_{m_j}:\theta(s^j_{m_j})\,.\\ &\Big(D(\theta(f_j)(x_1,\dots,x_{m_j}))\wedge\bigwedge\nolimits_{i\in\{1,\dots,m_j\};\ s^j_i\in S'}\varPsi_{s^j_i}(x_i)\Big)\\ &\quad\Rightarrow\varPsi_{s_j}\big(\theta(f_j)(x_1,\dots,x_{m_j})\big)\end{aligned}$$

$$\textbf{(Sortgen-intro)}\quad \frac{\varphi_1\wedge\cdots\wedge\varphi_k\Rightarrow\bigwedge_{s\in S'}\forall x:\theta(s)\,.\,p_s(x)}{(S',F',\theta\colon \bar{\Sigma}\to\Sigma)}$$

$F'=\{f_1\colon s^1_1\dots s^1_{m_1}\to s^1;\ \dots;\ f_k\colon s^k_1\dots s^k_{m_k}\to s^k\}$,
the predicates $p_s:\theta(s)$ $(s\in S')$ occur only in local assumptions,

$$\begin{aligned}\varphi_j = \forall x_1:\theta(s^j_1),\dots,x_{m_j}:\theta(s^j_{m_j})\,.\\ &\Big(D(\theta(f_j)(x_1,\dots,x_{m_j}))\wedge\bigwedge\nolimits_{i\in\{1,\dots,m_j\};\ s^j_i\in S'}p_{s^j_i}(x_i)\Big)\\ &\quad\Rightarrow p_{s_j}\big(\theta(f_j)(x_1,\dots,x_{m_j})\big)\end{aligned}$$

Fig. 2.2. Induction rules for CASL basic specifications

- each non-leaf node is the (instance of the) conclusion of some rule, with its children being the (instances of the) premises,
- all assumptions marked with [...] in the proof rules are marked as local assumptions.

If $\varPhi$ and φ consist of Σ-formulas, we also write $\varPhi\vdash_\Sigma\varphi$. In practice, one will work with acyclic graphs instead of trees, since this allows the re-use of lemmas.

Some rules contain a condition that some variables occur freely only in local assumptions. These conditions are the usual Eigenvariable conditions of natural deduction style calculi. More precisely, they mean that if the mentioned variables occur freely in an assumption in a proof tree, the assumption must be marked as local and have been used in the proof of the premise of the respective rule (that is, it must not be an undischarged assumption that gets discharged only by a different rule).

Some readers might prefer a Gentzen-style presentation of the calculus, since this makes the role of the local assumptions and the Eigenvariable conditions more explicit. In order to help clarifications, we reformulate two of the rules in Gentzen style here:

$$\textbf{(}\Rightarrow\textbf{-intro)}\quad \frac{\varPhi\cup\{\varphi\}\vdash\psi}{\varPhi\vdash\varphi\Rightarrow\psi}$$

$$\textbf{(Substitution)}\ \frac{\Phi \vdash \varphi}{\Phi \vdash (\bigwedge_{x_s \in FV(\varphi)} D(\nu(x_s))) \Rightarrow \varphi[\nu]}$$

if $\varphi[\nu]$ is defined and the free variables of φ do not occur freely in Φ.

We now come to soundness and completeness of the calculus, largely following the presentation in [9]. We present only the (easier) soundness proof, in order to allow the reader to get familiar with the rules. The completeness proof (for the first-order fragment) is more complex, but follows the standard techniques for first-order completeness proofs; hence, we do not repeat it here.

We say that a Σ-sentence φ *semantically follows* from a set of Σ-sentences Φ, written $\Phi \models_\Sigma \varphi$, if each Σ-model satisfying all sentences in Φ also satisfies φ.

Theorem 2.3. *The calculus is sound i.e.*

$$\Phi \models_\Sigma \varphi \quad \textit{if} \quad \Phi \vdash_\Sigma \varphi$$

Moreover, the calculus is complete if sort generation constraints are not used, i.e.

$$\Phi \models_\Sigma \varphi \quad \textit{only if} \quad \Phi \vdash_\Sigma \varphi$$

if Φ, φ do not contain sort generation constraints.

Proof. (Soundness) By induction over the proof tree construction, we show: for all signatures Σ, all Σ-models M, all S-sorted set of variables X (where S is the sort set of Σ) and all valuations $\rho\colon X \to M$, all Σ-formulas Φ, φ over X [1] with $\Phi \vdash_\Sigma \varphi$:

$$\text{if } M \models_\rho \Phi, \text{ then } M \models_\rho \varphi$$

(Absurdity), **(Tertium non datur)** and **(⇒-elim)** are easily seen to be sound by inspecting truth tables. Concerning **(⇒-intro)**, note that the subproof of ψ has assumptions $\Phi \cup \{\varphi\}$, hence by induction hypothesis, if $M \models_\rho \Phi \cup \{\varphi\}$, then $M \models_\rho \psi$. From this, we get if $M \models_\rho \Phi$, then $M \models_\rho \varphi \Rightarrow \psi$.

(∀-elim): $M \models_\rho \forall x{:}s\,.\,\varphi$ means that $M \models_{\rho[x_s \mapsto a]} \varphi$ for all $a \in s^M$, hence in particular $M \models_\rho \varphi$, since $\rho = \rho[x_s \mapsto \rho(x_s)]$.

(∀-intro): By the side condition, x_s does not occur freely in Φ. Hence, assuming that $M \models_\rho \Phi$, we get $M \models_{\rho[x_s \mapsto a]} \Phi$ for all $a \in s^M$. By the induction hypothesis, for these then $M \models_\xi \varphi$. Hence, $M \models_\rho \forall x{:}s\,.\,\varphi$.

Soundness of **(Reflexivity)** and **(Totality)** are obvious.

Soundness of **(Congruence)** and **(Substitution)** follows from the following Lemma from [9].

Lemma 2.4. ***Substitution Lemma***
Let M be a Σ-model, $\rho\colon Y \to M$ be a valuation, $\nu\colon X \to T_\Sigma(Y)$ be a substitution and φ a Σ-formula over X. Under the conditions that

[1] It may be necessary to extend or shrink the variable set X when applying the induction hypothesis. This is no problem, since the induction is over all such variable sets in parallel, and since carrier sets of models are non-empty and hence satisfaction is not affected by adding or removing extra variables.

- $\rho^{\#} \circ \nu \colon X \to M$ *is well-defined (i.e. for all* $x \in X_s, s \in S$ *we have* $M \models_\rho D(\nu(x_s)))$ *and*
- $\varphi[\nu]$ *is defined,*

we have

$$M \models_{\rho^{\#} \circ \nu} \varphi \;\; \textit{if and only if} \;\; M \models_\rho \varphi[\nu]$$

where $\rho^{\#} \colon T_\Sigma(Y) \to M$ *is the obvious extension of* ρ *to terms.* □

Soundness of **(Function Strictness)** and **(Predicate Strictness)** follow from the semantics of function and predicate application, which ensures that terms are defined only if all subterms are defined, and predicates hold only for defined terms.

Soundness of **(Induction)**: Let $\bar{\Sigma} = (\bar{S}, \bar{TF}, \bar{PF}, \bar{P})$. Given a Σ-model M, an $\bar{S}$-sorted set $\mathcal{P} \subseteq M|_\theta$ is (S', F', θ)-*closed* iff it is closed under the application of functions $\theta(f_{ws})^M$ with $f \in F'_{ws}$ and $\mathcal{P}_s = \theta(s)^M$ for $s \in \bar{S} \setminus S'$.

Lemma 2.5. *In the notation of rule* **Induction***, given a valuation* $\rho \colon X \to M$*, consider the* $\bar{S}$*-sorted set* $\mathcal{P}(\rho)$ *formed by taking* $\{a \mid M \models_{\rho[x_s \mapsto a]} \Psi_s\}$ *for* $s \in S'$ *and of* $\theta(s)^M$ *for* $s \in \bar{S} \setminus S'$*.*

Then we have

$$M \models_\rho \varphi_1 \wedge \cdots \wedge \varphi_k \;\; \textit{iff} \; \mathcal{P}(\rho) \; \textit{is} \; (S', F', \theta)\textit{-closed.}$$

Proof. This directly follows from the form of the φ_j: φ_j states that for any argument tuple such that each argument of sort $s \in S'$ is in $\mathcal{P}(\rho)$, the application of f_j to the argument tuple is in $\mathcal{P}(\rho)$. Since any argument of sort $s \in \bar{S} \setminus S'$ trivially is in $\mathcal{P}(\rho)$, this amounts to closure of $\mathcal{P}(\rho)$ under f_j. □

Now assume that a model M under a valuation $\rho \colon X \to M$ satisfies the premises of the rule. Satisfaction of the sort generation constraint $(S', F', \theta \colon \bar{\Sigma} \to \Sigma)$ means that the smallest (S', F', θ)-closed set is $M|_\theta$. Satisfaction of the second premise by Lemma 2.5 means that $\mathcal{P}(\rho)$ is (S', F', θ)-closed. Hence, $\mathcal{P}(\rho)$ is already $M|_\theta$. But this just means that ρ satisfies the conclusion.

Soundness of **(Sortgen-intro)**: since the predicates $p_s : \theta(s)$ do neither occur in non-local assumptions nor in the conclusion of the rule, it is possible to assume that they are interpreted as "term generatedness by the operations in $\theta(F')$". With this, the premise of the rule reads "if the operations in $\theta(F')$ preserve term generatedness by the operations in $\theta(F')$, then all elements of carriers for sorts in $\theta(S')$ are term generated by the operations in $\theta(F')$". Since the condition before the 'then' is trivially true, this means that all elements of carriers for sorts in $\theta(S')$ are term generated by the operations in $\theta(F')$. Hence, the sort generation constraint is true.

The proof of completeness follows the lines of [9]. □

One may have doubts whether the rule **(Sortgen-intro)** really can be sound. After all, it seems to introduce a second-order principle based on first-order reasoning. (Note that an induction rule for all first-order formulas is different from an induction rule with a second-order predicate variable.) However, due to the Eigenvariable condition for the predicate symbols mentioned in the rule, we actually have some form of universally quantified predicate variables. In particular, in order to derive a sort generation constraint, one usually needs a (possibly different) sort generation constraint among the premises.

Theorem 2.6. *If sort generation constraints are used, the calculus is not complete. Moreover, there cannot be a recursively axiomatized sound and complete calculus for many-sorted* CASL *basic specifications.*

Proof. With sort generation constraints, the second-order Peano axioms can be expressed, specifying the natural numbers up to isomorphism. The stated results then follow from Gödel's incompleteness theorem (see e.g. [64]). □

Since completeness implies compactness (proofs are finite), incompleteness is also a corollary of the following theorem:

Theorem 2.7. *The semantic consequence relation $\models_\Sigma$ for many-sorted* CASL *basic specifications is not compact. (Compactness means that every formula that follows from an arbitrary set of formulas already follows from a finite subset of that set.)*

Proof. From the Peano axioms and $\{p(suc^n(0)) \mid n \in \mathbf{N}\}$, we can semantically derive $\forall x : nat.p(x)$. However, this does not follow from a finite subset. □

Theorem 2.8. *The above proof calculus satisfies the properties of an entailment system, i.e.*

1. reflexivity: $\{\varphi\} \vdash_\Sigma \varphi$,
2. monotonicity: *if* $\Gamma \vdash_\Sigma \varphi$ *and* $\Gamma' \supseteq \Gamma$ *then* $\Gamma' \vdash_\Sigma \varphi$,
3. transitivity: *if* $\Gamma \vdash_\Sigma \varphi_i$, *for* $i \in I$, *and* $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, *then* $\Gamma \vdash_\Sigma \psi$,
4. translation: *if* $\sigma\colon \Sigma_1 \to \Sigma_2$ *and* $\Gamma \vdash_{\Sigma_1} \varphi$, *then* $\sigma(\Gamma) \vdash_{\Sigma_2} \sigma(\varphi)$.

Proof. Reflexivity and monotonicity directly follow from the notion of derivation tree. Transitivity follows by noting that derivation trees can be composed. Translation follows by noting that the translation of a proof rule by a signature morphism yields an instance of a proof rule. □

Instead of using the above calculus, it is also possible to use an encoding of the CASL logic into second-order logic. This means that not only subsorting, but also partiality are coded out using standard many-sorted first-order logic, while sort generation constraints are translated to second-order induction axioms. Using this encoding, any entailment relation between sets of sentences in the CASL logic can be translated to an equivalent entailment in second-order logic. Hence, any calculus for second-order logic (or first-order logic with induction) can be re-used for CASL. The details can be found in [41].

3

Subsorting Specification Calculus

The logic of subsorted basic specifications is defined via a reduction to many-sorted specifications (see Chap. III:3). In particular, subsorted Σ-sentences are many-sorted $\Sigma^{\#}$-sentences for some many-sorted signature $\Sigma^{\#}$ constructed out of a subsorted signature Σ. Hence, we can just use the proof calculus for many-sorted basic specifications, while adding the following *logical axioms* (taken from the semantics of subsorted specifications):

Identity: $\forall x_s . em_{\langle s \rangle, s} \langle x_s \rangle \overset{\mathrm{e}}{=} x_s$

Transitivity: $\forall x_s . em_{\langle s' \rangle, s''} \langle em_{\langle s \rangle, s'} \langle x_s \rangle \rangle \overset{\mathrm{e}}{=} em_{\langle s \rangle, s''} \langle x_s \rangle$ for $s \leq s' \leq s''$

Projection: $\forall x_s . pr_{\langle s' \rangle, s} \langle em_{\langle s \rangle, s'} \langle x_s \rangle \rangle \overset{\mathrm{e}}{=} x_s$ for $s \leq s'$

Projection-injectivity: $\forall \{x_{s'}, y_{s'}\} . pr_{\langle s' \rangle, s} \langle x_{s'} \rangle \overset{\mathrm{e}}{=} pr_{\langle s' \rangle, s} \langle y_{s'} \rangle \Rightarrow x_{s'} \overset{\mathrm{e}}{=} y_{s'}$ for $s \leq s'$

Membership: $\forall x_{s'} . in(s)_{\langle s' \rangle} \langle x_{s'} \rangle \Leftrightarrow D(pr_{\langle s' \rangle, s} \langle x_{s'} \rangle)$ for $s \leq s'$

Function-monotonicity:

$$\forall \{x^1_{\overline{s}_1}, \ldots, x^n_{\overline{s}_n}\} . em_{\langle s \rangle, s''} \langle f_{w,s} \langle em_{\langle \overline{s}_1 \rangle, s_1} \langle x^1_{\overline{s}_1} \rangle, \ldots, em_{\langle \overline{s}_n \rangle, s_n} \langle x^n_{\overline{s}_n} \rangle \rangle \rangle$$
$$\overset{\mathrm{s}}{=} em_{\langle s' \rangle, s''} \langle f_{w',s'} \langle em_{\langle \overline{s}_1 \rangle, s'_1} \langle x^1_{\overline{s}_1} \rangle, \ldots, em_{\langle \overline{s}_n \rangle, s'_n} \langle x^n_{\overline{s}_n} \rangle \rangle \rangle$$

for $f_{w,s} \sim_F f'_{w',s'}$, where $w' = \langle s'_1, \ldots, s'_n \rangle$ and $w = \langle s_1, \ldots, s_n \rangle$, with $\overline{w} \leq w, w'$ for some $\overline{w} = \langle \overline{s}_1, \ldots, \overline{s}_n \rangle$, and $s, s' \leq s''$

Predicate-monotonicity:

$$\forall \{x^1_{\overline{s}_1}, \ldots, x^n_{\overline{s}_n}\} .\ p_w \langle em_{\langle \overline{s}_1 \rangle, s_1} \langle x^1_{\overline{s}_1} \rangle, \ldots, em_{\langle \overline{s}_n \rangle, s_n} \langle x^n_{\overline{s}_n} \rangle \rangle$$
$$\Leftrightarrow p'_{w'} \langle em_{\langle \overline{s}_1 \rangle, s'_1} \langle x^1_{\overline{s}_1} \rangle, \ldots, em_{\langle \overline{s}_n \rangle, s'_n} \langle x^n_{\overline{s}_n} \rangle \rangle$$

for $p_w \sim_P p'_{w'}$, where $w' = \langle s'_1, \ldots, s'_n \rangle$ and $w = \langle s_1, \ldots, s_n \rangle$, with $\overline{w} \leq w, w'$ for some $\overline{w} = \langle \overline{s}_1, \ldots, \overline{s}_n \rangle$

Soundness and completeness results directly carry over from the many-sorted case.

It seems hard (if not impossible, without further assumptions on the signatures) to build a subsorted calculus directly working on the CASL input syntax (i.e. without fully qualified symbols). This has been done for OBJ3 [25]. However, the calculus of OBJ3 imposes special requirements on signatures such as regularity, which are not present in CASL.

4

Structured Specification Calculus

For structured theorem proving, there are several possible ways to go. One possibility is to directly work on the language of Casl structured specifications, as e.g. in [58]. However, the corresponding calculus becomes inevitably rather complex, and soundness (let alone completeness) is hard to see.

The other possibility, which we will follow here, is to use a kernel language. We use so-called *development graphs* as a simple kernel formalism for structured theorem proving and proof management. A development graph consists of a set of nodes (corresponding to whole structured specifications or parts thereof), and a set of arrows called definition links, indicating the dependency of each involved structured specification on its subparts.

The link between Casl structured specifications and development graphs is given by a *verification semantics*. It is similar to the static semantics of structured specifications, but simultaneously extracts a development graph and a set of proof obligations (the latter are called theorem links). The proof obligations arise e.g. from instantiations of generic specifications. There is an adequacy theorem stating that the proof obligations can be discharged if and only if the model semantics of the structured specification under consideration succeeds. Further proof obligations are generated by semantic annotations such as **%implies** and **%cons**.

The *proof calculus* for development graphs is given by rules that allow for decomposing *global* theorem links into simpler ones, until eventually *local implications* are reached. The latter can be discharged using a logic-specific calculus as introduced in Chap. 2. We also address soundness and completeness of the proof calculus.

Let us now add some remarks about the choice of the kernel language. Development graphs are not the only possible choice. Another choice would be the simple kernel language of structured specifications given in [7]. The drawback of this approach is that its calculus is based on the rather strong assumption that the institution has the Craig interpolation property. By contrast, the calculus for development graphs is based on a different assumption, namely the existence of weakly amalgamable cocones (cf. Def. 4.1). Note that

Craig interpolation (although technically incomparable in strength with the property of existence of weakly amalgamable cocones) for practical purposes is the stronger property (cf. the results of [17]). The deeper reason of these differences is that the calculus for development graphs contains *global* rules for graphs involving hiding that introduce additional nodes corresponding to normal forms of structured specifications (note however that the structure of the specification is still kept). By contrast, the rules for structured specifications developed by Borzyszkowski [7] are entirely *local*, which exactly is the reason why the Craig interpolation property is needed to achieve completeness.

An important practical difference of the two approaches is the following. In contrast to the kernel language of structured specifications, development graphs allow for expressing the *sharing* among specifications due to multiple references to named specifications. Moreover, the proof management tools for Casl work directly on development graphs; hence, the material presented here can serve as a formal background for the use of these tools and for the understanding of how they work. Last but not least, development graphs also support management of change.

Development graphs are a device for dealing with *structured* specifications. They should not be confused with the (formally quite similar) diagrams arising in the extended semantics of *architectural* specifications (see Sect. III:5.6). The crucial difference is in the role of hiding when a specification is used twice, with parts of it hidden in both cases. If a specification SPEC occurs within a structured specification, then the semantics reflected by the development graphs requires that the overall model can be extended with the hidden parts for each occurrence of SPEC separately. In contrast, if a unit UN : SPEC is used within an architectural unit term, then the semantics reflected by the architectural diagrams requires that the overall model can be uniformly extended with the hidden parts to a single model of SPEC, common for all the occurrences of UN. This amounts to saying that a global model of the colimit of the architectural diagram can be constructed. This is not possible for development graphs. Although there is a colimit construction for them as well, it is different in that the diagram is not given directly by the development graph, but by the diagram of all paths in the development graph (cf. the rule **(Theorem-Hide-Shift)** in Sect. 4.4 below).

4.1 Institution Independence

In order to achieve independence from the specific framework of basic specifications, we assume that we have an entailment relation for basic specifications (either given by some proof calculus, or by some logical encoding, or in some other way). Based on this, the proof calculus in this chapter is largely logic-independent. A framework for basic specification formally consists of an institution $(\mathbf{Sig}, \mathbf{Sen}, \mathbf{Mod}, \models)$ [20] together with a sound entailment system $\vdash$ for the institution (this is also called a *logic*). Here, an entailment system is a family of relations $(\vdash_\Sigma \subseteq \mathcal{P}(\mathbf{Sen}(\Sigma)) \times \mathbf{Sen}(\Sigma))_{\Sigma \in |\mathbf{Sig}|}$ between sets of sentences and sentences that is required to satisfy the properties listed in Theorem 2.8.

Additionally, we need a technical assumption about the logic, namely that every finite diagram of signatures has a weakly amalgamable cocone. We now explain what this means.

Definition 4.1. *Given a diagram $D\colon J \to \mathbf{Sig}$* [1]*, a family of models $(M_j)_{j\in|J|}$ is called* D-consistent *if $M_k|_{D(\delta)} = M_j$ for each $\delta\colon j \to k \in J$. A cocone $(\Sigma, (\mu_j\colon D(j) \to \Sigma)_{j\in|J|})$ over the diagram $D\colon J \to \mathbf{Sig}$ is called* weakly amalgamable *if for each D-consistent family of models $(M_j)_{j\in|J|}$, there is a Σ-model M with $M|_{\mu_j} = M_j$ $(j \in |J|)$. If this model is unique, the cocone is called* amalgamable*. If additionally also model morphisms can be uniquely amalgamated in this way, the cocone is called* morphism-amalgamable*. A logic is said to* admit weak amalgamation*, if each finite diagram of signatures has a weakly amalgamable cocone.*

Proposition 4.2. *The Casl logic for many-sorted basic specifications admits weak amalgamation.*

Proof. According to Theorem III:2.17, it suffices to take the colimit of the diagram – this is even morphism-amalgamable. □

Note that this proposition does *not* hold for subsorted specifications, see [63]. But we provide a way around this problem: it suffices that the logic at hand can be *represented* in a logic admitting weak amalgamation.

In order to define what such a representation means, we need some auxiliary notions. Given an arbitrary institution, a *theory* is a pair $T = \langle \Sigma, \Psi \rangle$, where $\Sigma \in \mathbf{Sig}$ and $\Psi \subseteq \mathbf{Sen}(\Sigma)$ (we set $Sig(T) = \Sigma$ and $Ax(T) = \Psi$). *Theory morphisms* $\sigma\colon \langle \Sigma, \Psi \rangle \to \langle \Sigma', \Psi' \rangle$ are those signature morphisms $\sigma\colon \Sigma \to \Sigma'$ for which $\Psi' \models_{\Sigma'} \sigma(\Psi)$, that is, axioms are mapped to logical consequences. By inheriting composition and identities from **Sig**, we obtain a category **Th** of theories. It is easy to extend **Sen** and **Mod** to **Th** by putting $\mathbf{Sen}(\langle \Sigma, \Psi \rangle) = \mathbf{Sen}(\Sigma)$ and letting $\mathbf{Mod}(\langle \Sigma, \Psi \rangle)$ be the full subcategory of $\mathbf{Mod}(\Sigma)$ induced by the class of those models M satisfying Ψ. The category **Pres** of *presentations* (also called *flat specifications*) is just the full subcategory of theories having finite sets of axioms.

Given institutions I and J, a *simple theoroidal institution comorphism* [23, 34, 65] (also called *simple map of institutions* [34] or simple institution representation) $R = (\Phi, \alpha, \beta)\colon I \to J$ consists of

[1] Diagrams have been introduced in Sect. III:5.6.

- a functor $\Phi\colon \mathbf{Sig}^I \to \mathbf{Pres}^J$,[2]
- a natural transformation $\alpha\colon \mathbf{Sen}^I \to \mathbf{Sen}^J \circ \Phi$,
- a natural transformation $\beta\colon \mathbf{Mod}^J \circ \Phi^{op} \to \mathbf{Mod}^I$

such that the following *comorphism condition* is satisfied for all $\Sigma \in \mathbf{Sig}^I$, $M' \in \mathbf{Mod}^J(\Phi(\Sigma))$ and $\varphi \in \mathbf{Sen}^I(\Sigma)$:

$$M' \models^J_{Sig(\Phi(\Sigma))} \alpha_\Sigma(\varphi) \Leftrightarrow \beta_\Sigma(M') \models^I_\Sigma \varphi.$$

A comorphism is called *model-isomorphic*, if each model translation β_Σ is an isomorphism.

Remark 4.3. **General assumption:** The given institution is embedded via a model-isomorphic simple theoroidal institution comorphism $R = (\Phi, \alpha, \beta)$ into an institution that comes with an entailment system (i.e. forms a logic) and furthermore admits weak amalgamation.

Note that this assumption is fulfilled for the subsorted CASL institution: it can be embedded into the many-sorted CASL institution as indicated in the semantics of subsorted specifications. Another possibility is to embed CASL into enriched CASL as described in [63]. The advantage of using the many-sorted CASL institution is its simplicity (compared with enriched CASL), the advantage of enriched CASL is that the comorphism is simpler and moreover we keep the subsorting structure, which may be exploited by special calculi [31].

Remark 4.4. We will use the general assumption tacitly in the following sense. At one place of the proof calculus, we rely on the fact that the underlying institution comes with an entailment system, and at several places, we need the existence of weakly amalgamable cocones – and we will presume that the institution we are working with enjoys these properties. Even if the institution should fail to satisfy them, we can translate the constructed development graph along the comorphism given by the general assumption, and we can rely on the target institution having the needed properties. The translation of development graphs, together with the fact that it is sound and complete, is given in Sect. 4.3 below. Note that in practice, one might wish to use the translation in a rather flexible way, namely only in those cases where a weakly amalgamable cocone does not exist in the original institution. This will be possible in the framework of heterogeneous specifications [40]. Another possibility is to limit the rules of the proof calculus to those cases where all needed weakly amalgamable cocones already exist in the original institution. However, this will lead to a loss of completeness for institutions that do not admit weak amalgamation (although this loss might not be severe in practice).

[2] Meseguer [34] requires $\Phi\colon \mathbf{Th}^I \to \mathbf{Th}^J$, but since μ is theoroidal, both formulations are equivalent using Meseguer's α-extension (except for the fact that we use presentations instead of theories).

4.2 Development Graphs

Development graphs are structured as follows. Leaves in a graph correspond to basic specifications, which do not make use of other specifications. Inner nodes correspond to structured specifications. The links that capture the construction of structured specifications in the graph are called *definition links*. Arising proof obligations are attached as so-called *theorem links* to this graph.

Definition 4.5. *A* development graph *is an acyclic, directed graph* $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$*.*

$\mathcal{N}$ is a set of nodes[3]*. Each node $N \in \mathcal{N}$ is labelled with a pair (Σ^N, Ψ^N) such that Σ^N is a signature and $\Psi^N \subseteq \mathbf{Sen}(\Sigma^N)$ is the set of* local axioms *of N.*

$\mathcal{L}$ is a set of directed links, so-called definition links*, between elements of $\mathcal{N}$. Each definition link from a node O to a node N is either*

- global *(denoted $O \xRightarrow{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^O \to \Sigma^N$, or*
- local *(denoted $O \xrightarrow{\sigma} N$), again annotated with a signature morphism $\sigma : \Sigma^O \to \Sigma^N$, or*
- hiding *(denoted $O \xRightarrow[hide]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^N \to \Sigma^O$* going against the direction of the link*, or*
- free *(denoted $O \xRightarrow[free]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma \to \Sigma^O$ for some signature Σ, with the requirement that $\Sigma^O = \Sigma^N$.*

To simplify matters, we write $O \xRightarrow{\sigma} N \in \mathcal{DG}$ instead of $O \xRightarrow{\sigma} N \in \mathcal{L}$ when $\mathcal{L}$ are the links of $\mathcal{DG}$. We use N, O, P, Q, K as variables for nodes, and L as variable for links.

Since development graphs are acyclic, we can use induction principles in definitions and proofs concerning development graphs.

The next definition captures the existence of a path of local and global definition links between two nodes. Notice that such a path must not contain any hiding links.

Definition 4.6. *Let $\mathcal{DG}$ be a development graph. The notion of* global reachability *is defined inductively: a node N is globally reachable from a node O via a signature morphism σ, $O \overset{\sigma}{\Longrightarrow\mkern-14mu\Rightarrow} N$ for short, iff*

- *either $O = N$ and $\sigma = id$, or*
- *$O \xRightarrow{\sigma'} P \in \mathcal{DG}$, and $P \overset{\sigma''}{\Longrightarrow\mkern-14mu\Rightarrow} N$, with $\sigma = \sigma'' \circ \sigma'$, or*
- *$O \xRightarrow[free]{\sigma'} P \in \mathcal{DG}$ and $P \overset{\sigma}{\Longrightarrow\mkern-14mu\Rightarrow} N$ (note that σ' is just ignored here).*

[3] The structure of nodes is left unspecified here; we assume that they come from some set *Nodes* of nodes, and new nodes are available as discussed in Sect. III:5.6.

A node N is locally reachable *from a node O via a signature morphism σ, $O{>}\!\!\overset{\sigma}{\Longrightarrow\!\!\!\!\Rightarrow} N$ for short, iff $O \overset{\sigma}{\Longrightarrow\!\!\!\!\Rightarrow} N$ or there is a node P with $O \xrightarrow{\sigma'} P \in \mathcal{DG}$ and $P \overset{\sigma''}{\Longrightarrow\!\!\!\!\Rightarrow} N$, such that $\sigma = \sigma'' \circ \sigma'$. Note that, in contrast to global reachability, local reachability is* not *transitive.*

Obviously global reachability implies local reachability.

Definition 4.7. *Given a node $N \in \mathcal{N}$, its associated class $\mathbf{Mod}_{\mathcal{DG}}(N)$ of models (or N-models for short) is inductively defined to consist of those Σ^N-models M for which*

1. *M satisfies the local axioms Ψ^N,*
2. *for each $O \xRightarrow{\sigma} N \in \mathcal{DG}$, $M|_\sigma$ is an O-model,*
3. *for each $O \xrightarrow{\sigma} N \in \mathcal{DG}$, $M|_\sigma$ satisfies the local axioms Ψ^O,*
4. *for each $O \xRightarrow[hide]{\sigma} N \in \mathcal{DG}$, M has a σ-expansion M' (i.e. $M'|_\sigma = M$) that is an O-model, and*
5. *for each $O \xRightarrow[free]{\sigma} N \in \mathcal{DG}$, M is an O-model that is σ-free in $\mathbf{Mod}(O)$.*

   The latter means that for each O-model M' and each model morphism $h\colon M|_\sigma \to M'|_\sigma$, there exists a unique model morphism $h^\# : M \to M'$ with $h^\#|_\sigma = h$.

Definition 4.8. *Let $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$ be a development graph. A node $N \in \mathcal{N}$ is* flattenable *iff for all nodes $O \in \mathcal{N}$ with incoming hiding or free definition links, it holds that N is not globally reachable from O.*

Definition 4.9. *Let $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$ be a development graph. For $N \in \mathcal{N}$, the* theory *$Th_{\mathcal{DG}}(N)$ of N is defined by*

$$\Psi^N \cup \bigcup_{P{>}\!\!\overset{\sigma}{\Longrightarrow\!\!\!\!\Rightarrow} N} \sigma(\Psi^P)$$

Proposition 4.10. *1. $O \overset{\sigma}{\Longrightarrow\!\!\!\!\Rightarrow} N$ and $M \in \mathbf{Mod}(N)$ imply $M|_\sigma \in \mathbf{Mod}(O)$.*
2. If $O{>}\!\!\overset{\sigma}{\Longrightarrow\!\!\!\!\Rightarrow} N$ and $M \in \mathbf{Mod}(N)$, then $M|_\sigma \models \Psi^O$.

Proof. 1. Easy induction over the definition of global reachability.
2. By 1 and Definition 4.7, 3.

Proposition 4.11. *1. $\mathbf{Mod}(N) \subseteq \mathbf{Mod}(Th_{\mathcal{DG}}(N))$.*
2. If N is flattenable, then $\mathbf{Mod}(N) = \mathbf{Mod}(Th_{\mathcal{DG}}(N))$.

Proof. 1. By Proposition 4.10, 2 and Definition 4.7, 1.
2. By 1, it suffices to prove the '$\supseteq$' direction. Let M be a $Th_{\mathcal{DG}}(N)$-model. Let $len(p)$ be the length of a path p witnessing $O \overset{\tau}{\Longrightarrow\!\!\!\!\Rightarrow} N$. Let $maxp$ be

the maximal such length in $\mathcal{DG}$. We show that for any $O \overset{\tau}{\Longrightarrow\!\!\!\!>} N$, $M|_\tau$ is an O-model. We proceed by induction over $maxp - len(p)$ with p witnessing $O \overset{\tau}{\Longrightarrow\!\!\!\!>} N$. Since N is flattenable, we only have to show clauses 1 to 3 of Definition 4.7:

1. Since global implies local reachability, $O\!>\!\overset{\tau}{\Longrightarrow\!\!\!\!>} N$, and $\tau(\Psi^O) \subseteq Th_{\mathcal{DG}}(N)$; hence $M \models \tau(\Psi^O)$. By the satisfaction condition for institutions, $M|_\tau \models \Psi^O$.
2. Let $P \xRightarrow{\theta} O$, hence $P \overset{\tau\circ\theta}{\Longrightarrow\!\!\!\!>} N$. By the induction hypothesis, $M|_{\tau\circ\theta}$ is a P-model.
3. Let $P \xrightarrow{\theta} O$, hence $P\!>\!\overset{\tau\circ\theta}{\Longrightarrow\!\!\!\!>} N$. With a similar argument as for 1, we get $M|_{\tau\circ\theta} \models \Psi^P$.

This completes the induction. Since $N \overset{id}{\Longrightarrow\!\!\!\!>} N$, M is an N-model. □

Complementary to definition links, which *define* the theories of related nodes, we introduce the notion of a *theorem link* with the help of which we are able to *postulate* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. Theorem links come, like definition links, in four different versions:

- *global* theorem links $O \overset{\sigma}{=\!=\!\Rightarrow} N$, where $\sigma\colon \Sigma^O \to \Sigma^N$,
- *local* theorem links $O \overset{\sigma}{-\,-\!\!>} N$, where $\sigma\colon \Sigma^O \to \Sigma^N$,
- *hiding* theorem links $O \underset{hide\ \theta}{\overset{\sigma}{=\!=\!\Rightarrow}} N$, where for some Σ, $\theta\colon \Sigma \to \Sigma^O$ and $\sigma\colon \Sigma \to \Sigma^N$ [4], and
- *free* theorem links $O \underset{free\ \theta}{\overset{\sigma}{=\!=\!\Rightarrow}} N$, where $\sigma\colon \Sigma^O \to \Sigma^N$ and for some Σ, $\theta\colon \Sigma \to \Sigma^O$. In case that Σ is the initial signature and θ is the unique signature morphism, the link is written as $O \underset{free\ !}{\overset{\sigma}{=\!=\!\Rightarrow}} N$.

Moreover, we will also need *local implications* of the form $N \Rightarrow \Psi$, where Ψ is a set of Σ^N-sentences. $N \Rightarrow \{\varphi\}$ also is written $N \Rightarrow \varphi$. The semantics of local implications and of theorem links is given by the next definition.

Definition 4.12. *Let $\mathcal{DG}$ be a development graph and O, N nodes in $\mathcal{DG}$.*

- *$\mathcal{DG}$ implies a local implication $N \Rightarrow \Psi$, written $\mathcal{DG} \models N \Rightarrow \Psi$, if for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M \models \Psi$.*
- *$\mathcal{DG}$ implies a global theorem link $O \overset{\sigma}{=\!=\!\Rightarrow} N$ (denoted $\mathcal{DG} \models O \overset{\sigma}{=\!=\!\Rightarrow} N$) iff for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M|_\sigma \in \mathbf{Mod}_{\mathcal{DG}}(O)$.*

[4] Here, σ is the reduction morphism (comparable to that of global theorem links), and θ is the hiding morphism (extending a signature with hidden parts).

- *$\mathcal{DG}$ implies a local theorem link $O \overset{\sigma}{- - \succ} N$ (denoted $\mathcal{DG} \models O \overset{\sigma}{- - \succ} N$) iff for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M|_\sigma \models \Psi^O$. (Note that by the satisfaction condition, this is equivalent to $\mathcal{DG} \models N \Rightarrow \sigma(\Psi^O)$.)*
- *$\mathcal{DG}$ implies a hiding theorem link $O \overset{\sigma}{\underset{hide\ \theta}{=\!=\!\Rightarrow}} N$ (denoted $\mathcal{DG} \models O \overset{\sigma}{\underset{hide\ \theta}{=\!=\!\Rightarrow}} N$) iff for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M|_\sigma$ has a θ-expansion to some O-model.*
- *$\mathcal{DG}$ implies a free theorem link $O \overset{\sigma}{\underset{free\ \theta}{=\!=\!\Rightarrow}} N$ (denoted $\mathcal{DG} \models O \overset{\sigma}{\underset{free\ \theta}{=\!=\!\Rightarrow}} N$) iff for all $M \in \mathbf{Mod}_{\mathcal{DG}}(N)$, $M|_\sigma$ is an O-model which is θ-free in $\mathbf{Mod}_{\mathcal{DG}}(O)$.*

Remark 4.13. Note that theorem links may be captured by inclusion between model classes of some (additional) nodes in the development graph. For instance, consider a hiding theorem link $O \overset{\sigma}{\underset{hide\ \theta}{=\!=\!\Rightarrow}} N$ in a development graph $\mathcal{DG}$, where $\theta\colon \Sigma \to \Sigma^O$ and $\sigma\colon \Sigma \to \Sigma^N$. One can add nodes N' and N'' to $\mathcal{DG}$, with $\Sigma^{N'} = \Sigma$, $\Sigma^{N''} = \Sigma^N$, and $\Psi^{N'} = \Psi^{N''} = \emptyset$, together with definition links $O \xRightarrow[hide]{\theta} N'$ and $N' \xRightarrow{\sigma} N''$. For the thus obtained development graph $\mathcal{DG}'$, we then have

$$\mathcal{DG} \models O \overset{\sigma}{\underset{hide\ \theta}{=\!=\!\Rightarrow}} N \text{ iff } \mathbf{Mod}_{\mathcal{DG}'}(N) \subseteq \mathbf{Mod}_{\mathcal{DG}'}(N'')$$

A similar construction can be performed for the other types of theorem link. □

Finally, we introduce the analogues of the semantic annotations in Casl. A global theorem link $O \overset{\sigma}{=\!=\!\Rightarrow} N$ can be strengthened to

- a *conservative extension*[5] (denoted as $O \overset{\sigma}{\underset{cons}{=\!=\!\Rightarrow}} N$); it holds if, additionally to the holding of the theorem link, every O-model has a σ-expansion to an N-model,
- a *monomorphic extension* (denoted as $O \overset{\sigma}{\underset{mono}{=\!=\!\Rightarrow}} N$); it holds if, additionally to the holding of the theorem link, every O-model has a σ-expansion to an N-model that is unique up to isomorphism, or
- a *definitional extension* (denoted as $O \overset{\sigma}{\underset{def}{=\!=\!\Rightarrow}} N$); it holds if, additionally to the holding of the theorem link, every O-model has a unique σ-expansion to an N-model.

These annotations can be seen as another kind of proof obligations. If there happens to be a global definition link $O \xRightarrow{\sigma} N$ in the development graph, we

[5] In the literature on model theory, this property is often called *model expansion property*, while the term *conservative extension* refers to a (weaker) proof-theoretic principle.

also write $O \xRightarrow[cons]{\sigma} N$, $O \xRightarrow[mono]{\sigma} N$, or $O \xRightarrow[def]{\sigma} N$, respectively. In this case, the theorem link part holds trivially, and only the conservativity, monomorphicity or definitionality statement is relevant.

We also allow for annotating *nodes* with *cons*, *mono* or *def*. This shall express that the trivial theorem link using the unique signature morphism from the empty signature[6] could be annotated with the same word[7]. Thus, the annotation *cons* for a node means that there is a model of the node (consistency), *mono* means that the node has exactly one model up to isomorphism (i.e. it is monomorphic), and *def* means that the node has exactly one model (the latter will occur only rarely).

4.3 Translating Development Graphs along Institution Comorphisms

Given a model-isomorphic simple theoroidal institution comorphism $R = (\Phi, \alpha, \beta)\colon I \to J$, we can extend this comorphism to a translation of development graphs over I into development graphs over J in the following way:

Given a development graph $\mathcal{DG}$ over I, let $R(\mathcal{DG})$ have the same nodes and links as $\mathcal{DG}$ (for clarity, given a node $N \in \mathcal{DG}$, we call the corresponding node $R(N) \in R(\mathcal{DG})$, and similarly for definition links). The associated signatures, local axioms and signature morphisms differ, of course:

- if $N \in \mathcal{DG}$, then $\Sigma^{R(N)} = Sig(\Phi(\Sigma^N))$, and
$$\Psi^{R(N)} = \alpha_{\Sigma^N}(\Psi^N) \cup Ax(\Phi(\Sigma^N))$$
- the signature morphisms decorating a link L are translated along Φ, and intermediate signatures Σ are replaced with $Sig(\Phi(\Sigma))$, yielding a link $R(L)$.

Theorem 4.14. *Given a model-isomorphic simple theoroidal institution comorphism $R = (\Phi, \alpha, \beta)\colon I \to J$ and a development graph $\mathcal{DG}$ over I, for each $N \in \mathcal{DG}$, the isomorphism*

$$\beta_{\Sigma^N} : \mathbf{Mod}(\Sigma^N) \to \mathbf{Mod}(\Phi(\Sigma^N))$$

restricts to the isomorphism

$$\beta_{\Sigma^N} : \mathbf{Mod}(N) \to \mathbf{Mod}(R(N))$$

Proof. First, note that indeed $\mathbf{Mod}(R(N)) \subseteq \mathbf{Mod}(\Phi(\Sigma^N))$, because $\Psi^{R(N)}$ includes $Ax(\Phi(\Sigma^N))$. We now proceed by induction over $\mathcal{DG}$. Hence, it suffices to show for each $M \in \mathbf{Mod}(\Phi(\Sigma))$:

[6] We here assume that the empty signature is initial.

[7] Here we tacitly assume that there is some special node having the initial signature and the empty set of axioms.

1. $\beta_{\Sigma^N}(M) \models \Psi^N$ iff $M \models \Psi^{R(N)}$,
2. for any ingoing definition link L into N, $\beta_{\Sigma^N}(M)$ satisfies L iff M satisfies $R(L)$.

Both can be shown in a straightforward way, using the satisfaction condition of the comorphism, naturality and isomorphism property of β and the fact that for any I-signature morphism σ, $\Phi(\sigma)$ is a theory morphism. □

Theorem 4.15. *Given a model-isomorphic simple theoroidal institution comorphism $R = (\Phi, \alpha, \beta)\colon I \to J$ and a development graph $\mathcal{DG}$ over I, let L be a theorem link over $\mathcal{DG}$. Then*

$$\mathcal{DG} \models L \textit{ iff } R(\mathcal{DG}) \models R(L)$$

Proof. By Theorem 4.14 and Remark 4.13. □

Note that with this translation of development graphs along comorphisms, new local axioms coming from $Ax(\Phi(\Sigma^N))$ are often partly repeated. One can optimize this by adding at each node only those axioms from $Ax(\Phi(\Sigma^N))$ that are not already present via links from other nodes.

4.4 Proof Rules for Development Graphs

In this section, we introduce logic-independent proof rules for development graphs. These rely on a logic-specific entailment relation for basic specifications as introduced in Chap. 1, as well as on logic-specific proof rules for conservativity and freeness, which will be covered in Sect. 4.6.

The proof rules work on judgements of the form $\mathcal{DG} \vdash L$, where $\mathcal{DG}$ is a development graph and L is a theorem link (of any kind) over $\mathcal{DG}$. As in the calculus for basic specifications, we follow a natural deduction style presentation and additionally use a graph-grammar like notation. We hope that this is still largely self-explanatory while improving readability.

The proof rules for development graphs presented below are typically applied backwards: given proof goal in form of a theorem link relative to some development graph, find a rule whose conclusion matches the proof goal, and recursively prove the premises of the rule. Note that within one rule, the judgements may refer to different development graphs. Often, the premises are formulated over development graphs that are larger than that for the conclusion. This means that applying rules backwards possibly adds some new nodes and edges to the development graph.

The rules allow for decomposing global theorem links into simpler ones. In a first step, one typically tries to get rid of hiding theorem links and to decompose global into local theorem links. This is done by applying the hiding decomposition rules. Thereby, new conservativity proof goals can be generated, which need to be tackled by the conservativity rules. The simple decomposition rules then allow for proving global theorem links when there is some parallel definition link, and for proving local theorem links and local implications by reasoning with the entailment system of the logic.

For the sake of readability, each rule is followed by its soundness proof.

4.4.1 Hiding Decomposition Rules

In order to get rid of hiding links going into the *source* of a global theorem link, one first applies **(Glob-Decomposition)**, ending up with some local and hiding theorem links. The rule **(Hide-Theorem-Shift)** allows to prove the latter, using conservativity of definition links. **(Borrowing)** can be used for shifting a proof goal along a conservative extension; hence, it also exploits conservativity of theorem links. Conservativity is dealt with in the next section. The central rule of the proof system is the rule **(Theorem-Hide-Shift)**. It is used to get rid of hiding definition links going into the *target* of a global theorem link.

$$\begin{tikzcd} & N' \\ O' \arrow[ur, Rightarrow, dashed, "\sigma'"] & N \arrow[u, Rightarrow, dashed, "\theta'", "cons"'] \end{tikzcd}$$

$$\dfrac{}{O' \overset{\sigma}{\underset{hide\ \theta}{=\!=\!\Rightarrow}} N}$$

$$\text{if } \sigma' \circ \theta = \theta' \circ \sigma$$

(Hide-Theorem-Shift)

The proof rules are written in a concise notation as above. We will spell out in detail what this notation means for the rule **(Hide-Theorem-Shift)**:

$$\dfrac{\begin{array}{c} \sigma' \circ \theta = \theta' \circ \sigma \\ N \xRightarrow{\theta'} N' \in \mathcal{DG} \\ \mathcal{DG} \vdash N \overset{\theta'}{\underset{cons}{=\!=\!\Rightarrow}} N' \\ \mathcal{DG} \vdash O' \overset{\sigma'}{=\!=\!\Rightarrow} N' \end{array}}{\mathcal{DG} \vdash O' \overset{\sigma}{\underset{hide\ \theta}{=\!=\!\Rightarrow}} N}$$

Soundness of **(Hide-Theorem-shift)**: assume that $\mathcal{DG} \models O' \overset{\sigma'}{=\!=\!\Rightarrow} N'$ and $N \overset{\theta'}{\underset{cons}{=\!=\!\Rightarrow}} N'$ is conservative. We have to show that $\mathcal{DG} \models O' \overset{\sigma}{\underset{hide\ \theta}{=\!=\!\Rightarrow}} N$. Let M be an N-model. Since $N \overset{\theta'}{\underset{cons}{=\!=\!\Rightarrow}} N'$ is conservative, M can be expanded to an N'-model M' with $M'|_{\theta'} = M$. By the assumption, $M'|_{\sigma'}$ is an O'-model. Thus, $M'|_{\sigma' \circ \theta} = M'|_{\theta' \circ \sigma} = M|_{\sigma}$ has a θ-expansion to an O'-model. □

$$\begin{tikzcd} & G_N(i) \arrow[d, Rightarrow, "\mu_i"] & (i \in |J|) \\ O \arrow[r, Rightarrow, dashed, "\mu_{\langle N\rangle}\circ\sigma"] & K \end{tikzcd}$$

$$\begin{tikzcd} & \mathit{Diag}_N \arrow[d, shift right=2] \arrow[d, shift left=2] \\ O \arrow[r, Rightarrow, dashed, "\sigma"] & N \end{tikzcd}$$

with K isolated and (μ_i) a weakly amalgamable cocone for the diagram Diag_N of nodes going into N (see explanation below)

(Theorem-Hide-Shift)

Since this rule is quite powerful, we need some preliminary notions. Given a node N in a development graph $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$, the idea is that we unfold the subgraph below N into a tree and form a diagram with this tree. More formally, define the *diagram* $\mathit{Diag}_N \colon J \to \mathbf{Sig}$ *associated with* N together with a map $G_N \colon |J| \to \mathcal{N}$ inductively as follows:

- $\langle N \rangle$ is an object in J, with $\mathit{Diag}_N(\langle N \rangle) = \Sigma^N$. Let $G_N(\langle N \rangle)$ be just N.
- if $i = \langle\, O \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \,\rangle$ is an object in J with $L_1, \ldots, L_n$ non-local definition links in $\mathcal{L}$, and $L = P \xRightarrow{\sigma} O$ or $L = P \xrightarrow{\sigma} O$ is a local or global definition link in $\mathcal{L}$, then
$$j = \langle\, P \xrightarrow{L} O \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \,\rangle$$
is an object in J with $\mathit{Diag}_N(j) = \Sigma^P$, and L is a morphism from j to i in J with $\mathit{Diag}_N(L) = \sigma$. We set $G_N(j) = P$.
- if $i = \langle\, O \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \,\rangle$ is an object in J with $L_1, \ldots, L_n$ non-local definition links in $\mathcal{L}$, and $L = P \underset{\mathit{hide}}{\xRightarrow{\sigma}} O$ is a hiding definition link in $\mathcal{L}$, then
$$j = \langle\, P \xrightarrow{L} O \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \,\rangle$$
is an object in J with $\mathit{Diag}_N(j) = \Sigma^P$, and L is a morphism from i to j in J with $\mathit{Diag}_N(L) = \sigma$. We set $G_N(j) = P$.

Now in order to apply **(Theorem-Hide-Shift)**, take a weakly amalgamable cocone $(\Sigma, (\mu_i \colon \mathit{Diag}_N(i) \to \Sigma)_{i\in|J|})$ for Diag_N (it exists by Remark 4.4), and let K be a new isolated node with signature Σ and with ingoing global definition links $G_N(i) \xRightarrow{\mu_i} K$ for $i \in |J|$ (if $G_N(i)$ has no ingoing free definition links, a local definition link $G_N(i) \xrightarrow{\mu_i} K$ would suffice). Here, an isolated node is one with no local axioms and no ingoing definition links other than those shown in the rule.

We once more spell the rule in detail:

$$\frac{\begin{array}{c}(\Sigma, (\mu_i \colon \mathit{Diag}_N(i) \to \Sigma)_{i \in |J|}) \text{ is a weakly amalgamable cocone for } \mathit{Diag}_N \\ \mathcal{DG}' = \mathcal{DG} \uplus \{K \text{ with } (\Sigma, \emptyset)\} \uplus \{\, G_N(i) \xRightarrow{\mu_i} K \mid i \in |J|\} \\ \mathcal{DG}' \vdash O \overset{\mu_{\langle N \rangle} \circ \sigma}{=\!=\!\Rightarrow} K\end{array}}{\mathcal{DG} \vdash O \overset{\sigma}{=\!=\!\Rightarrow} N}$$

Here, if we want to extend a given development graph $\mathcal{DG}$, we use a suggestive concise notation like $\mathcal{DG}' = \mathcal{DG} \uplus \{N' \text{ with } (\Sigma', \Psi);\ N \xRightarrow{\Sigma \hookrightarrow \Sigma'} N'\}$ which should be largely self-explanatory (in particular, 'N' with (Σ', Ψ)' means that we introduce a new node N' with $\Sigma^{N'} = \Sigma'$ and $\Psi^{N'} = \Psi$).

Soundness of **(Theorem-Hide-Shift)**: assume that $\mathcal{DG} \models O \overset{\mu_{\langle N \rangle} \circ \sigma}{=\!=\!\Rightarrow} K$. Let M be an N-model. We have to show $M|_\sigma$ to be an O-model in order to establish the holding of $O \overset{\sigma}{=\!=\!\Rightarrow} N$. We inductively define a family $(M_i)_{i \in |I|}$ of models $M_i \in \mathbf{Mod}(G_N(i))$ by putting

- $M_{\langle N \rangle} = M$,
- $M_{\langle P \xrightarrow{L} Q \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \rangle} = M'|_\sigma$, where $L = P \xRightarrow{\sigma} Q$ or $L = P \xrightarrow{\sigma} Q$ and $M' = M_{\langle Q \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \rangle}$, and
- $M_{\langle P \xrightarrow{L} Q \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \rangle}$ is a σ-expansion of M' to a P-model (existing since M' is a Q-model),

  where $L = P \underset{hide}{\overset{\sigma}{\Longrightarrow}} Q$ and $M' = M_{\langle Q \xrightarrow{L_1} \cdots \xrightarrow{L_n} N \rangle}$.

It is easy to show that this family is consistent with Diag_N. Since by the side condition of the rule, $(\Sigma, (\mu_i \colon \mathit{Diag}_N(i) \to \Sigma)_{i \in |J|})$ is a weakly amalgamable cocone, there is a Σ^K-model M_K with $M_K|_{\mu_i} = M_i$. The latter implies that M_K is a K-model. By the assumption, $M_K|_{\mu_{\langle N \rangle} \circ \sigma} = M_{\langle N \rangle}|_\sigma = M|_\sigma$ is an O-model. □

$$\frac{\begin{tikzcd} O \arrow[d, Rightarrow, "\theta"'] & N \arrow[d, Rightarrow, "\theta'"', "cons"] \\ O' \arrow[r, Rightarrow, dashed, "\sigma'"'] & N' \end{tikzcd}}{\begin{tikzcd} O \arrow[r, Rightarrow, dashed, "\sigma"'] \arrow[d, Rightarrow, "\theta"'] & N \arrow[d, Rightarrow, "\theta'"', "cons"] \\ O' & N' \end{tikzcd}}$$

if $\sigma' \circ \theta = \theta' \circ \sigma$

(Borrowing)

Soundness of **(Borrowing)**: Assume that (1) $\mathcal{DG} \models O \overset{\theta}{=\!=\!\Rightarrow} O'$, (2) $\mathcal{DG} \models N \underset{cons}{\overset{\theta'}{=\!=\!\Rightarrow}} N'$, and that (3) $\mathcal{DG} \models O' \overset{\sigma'}{=\!=\!\Rightarrow} N'$. Let M be an N-model. By (2), M has an expansion to an N'-model M' with $M'|_{\theta'} = M$. By (3), $M'|_{\sigma'}$ is an O'-model, and hence, by (1) $M'|_{\sigma'\circ\theta} = M'|_{\theta'\circ\sigma} = M|_\sigma$ is an O-model. □

$$\frac{\begin{array}{l} P \overset{\sigma\circ\tau}{-\,-\!\!\triangleright} O \text{ for each } P \rhd\!\overset{\tau}{\Longrightarrow\!\!\!\!\!>} N \\ Q \underset{hide\ \theta}{\overset{\sigma\circ\tau}{=\!=\!\Rightarrow}} O \text{ for each } Q \underset{hide}{\overset{\theta}{\Longrightarrow}} P \text{ and } P \overset{\tau}{\Longrightarrow\!\!\!\!\!>} N \\ Q \underset{free\ \theta}{\overset{\sigma\circ\tau}{=\!=\!\Rightarrow}} O \text{ for each } Q \underset{free}{\overset{\theta}{\Longrightarrow}} P \text{ and } P \overset{\tau}{\Longrightarrow\!\!\!\!\!>} N \end{array}}{N \overset{\sigma}{=\!=\!\Rightarrow} O}$$

(Glob-Decomposition)

Soundness of **(Glob-Decomposition)**: assume that

1. $\mathcal{DG} \models P \overset{\sigma\circ\tau}{-\,-\!\!\triangleright} O$ for each $P \rhd\!\overset{\tau}{\Longrightarrow\!\!\!\!\!>} N$,
2. $\mathcal{DG} \models Q \underset{hide\ \theta}{\overset{\sigma\circ\tau}{=\!=\!\Rightarrow}} O$ for each $Q \underset{hide}{\overset{\theta}{\Longrightarrow}} P$ and $P \overset{\tau}{\Longrightarrow\!\!\!\!\!>} N$, and
3. $\mathcal{DG} \models Q \underset{free\ \theta}{\overset{\sigma\circ\tau}{=\!=\!\Rightarrow}} O$ for each $Q \underset{free}{\overset{\theta}{\Longrightarrow}} P$ and $P \overset{\tau}{\Longrightarrow\!\!\!\!\!>} N$.

In order to show $\mathcal{DG} \models N \overset{\sigma}{=\!=\!\Rightarrow} O$, let M be an O-model. Let $len(p)$ be the length of a path p witnessing $P \overset{\tau}{\Longrightarrow\!\!\!\!\!>} N$. Let $maxp$ be the maximal such length in $\mathcal{DG}$. We show that for any $P \overset{\tau}{\Longrightarrow\!\!\!\!\!>} N$, $M|_{\sigma\circ\tau}$ is a P-model. We proceed by induction over $maxp - len(p)$ for p witnessing $P \overset{\tau}{\Longrightarrow\!\!\!\!\!>} N$. We have to show clauses 1 to 5 of Definition 4.7:

1. By the first assumption, $M|_{\sigma\circ\tau} \models \Psi^P$.
2. By the induction hypothesis, $M|_{\sigma\circ\tau}$ satisfies any global definition link going into P.
3. By the first assumption, $M|_{\sigma\circ\tau}$ satisfies any local definition link going into P.
4. By the second assumption, $M|_{\sigma\circ\tau}$ satisfies any hiding definition link going into P.
5. By the third assumption, $M|_{\sigma\circ\tau}$ satisfies any free definition link going into P.

This completes the induction. Since $N \overset{id}{\Longrightarrow\!\!\!\!\!>} N$, $M|_\sigma$ is an N-model. □

4.4.2 Conservativity Rules

$$\dfrac{\begin{tikzcd} O \arrow[r, Rightarrow, dashed, "\sigma"] \arrow[d, Rightarrow, dashed, "\theta"', "cons"] & N \arrow[d, Rightarrow, "\theta'"'] \\ O' \arrow[r, Rightarrow, "\sigma'"] & N' \end{tikzcd}}{\begin{tikzcd} N \arrow[d, Rightarrow, "\theta'"', "cons"] \\ N' \end{tikzcd}} \quad \text{if} \quad \begin{tikzcd} \Sigma^{O} \arrow[r, "\sigma"] \arrow[d, "\theta"] & \Sigma^{N} \arrow[d, "\theta'"] \\ \Sigma^{O'} \arrow[r, "\sigma'"] & \Sigma^{N'} \end{tikzcd}$$

is weakly amalgamable and N' is isolated.

(Cons-Shift)

Soundness of **(Cons-Shift)**: Assume that $O \overset{\theta}{\underset{cons}{=\!=\!\Rightarrow}} O'$ is conservative. We have to prove that $N \overset{\theta'}{\underset{cons}{=\!=\!\Rightarrow}} N'$ is conservative as well. Let M be an N-model. Since $O \overset{\theta}{\underset{cons}{=\!=\!\Rightarrow}} O'$ is conservative, $M|_\sigma$ has a θ-expansion M' being an O'-model. By weak amalgamation, there is some $\Sigma^{N'}$-model M' with $M'|_{\sigma'} = M'$ and $M'|_{\theta'} = M$. Since N' is isolated, M' is an N'-model. □

$$\dfrac{\begin{tikzcd} O \arrow[r, Rightarrow, dashed, "\sigma"] \arrow[d, Rightarrow, dashed, "\theta"', "def"] & N \arrow[d, Rightarrow, "\theta'"'] \\ O' \arrow[r, Rightarrow, "\sigma'"] & N' \end{tikzcd}}{\begin{tikzcd} N \arrow[d, Rightarrow, "\theta'"', "def"] \\ N' \end{tikzcd}} \quad \text{if} \quad \begin{tikzcd} \Sigma^{O} \arrow[r, "\sigma"] \arrow[d, "\theta"] & \Sigma^{N} \arrow[d, "\theta'"] \\ \Sigma^{O'} \arrow[r, "\sigma'"] & \Sigma^{N'} \end{tikzcd}$$

is amalgamable and N' is isolated.

(Def-Shift)

Soundness of **(Def-Shift)**: assume that $O \overset{\theta}{\underset{def}{=\!=\!\Rightarrow}} O'$ is definitional. We have to prove that $N \overset{\theta'}{\underset{def}{=\!=\!\Rightarrow}} N'$ is definitional as well. Let M be an N-model. By the argument used for the proof of soundness of **(Cons-shift)**, M has a θ'-expansion to an N'-model M'. Now let M'' be another N'-model with $M''|_{\theta'} = M = M'|_{\theta'}$. Then $M'|_{\sigma' \circ \theta} = M'|_{\theta' \circ \sigma} = M''|_{\theta' \circ \sigma} = M''|_{\sigma' \circ \theta}$. Since $O \overset{\theta}{\underset{def}{=\!=\!\Rightarrow}} O'$ is definitional, $M'|_{\sigma'} = M''|_{\sigma'}$. By uniqueness of the amalgamation, $M' = M''$. □

$$\dfrac{\begin{tikzcd} O \arrow[r, Rightarrow, "\sigma", "cons"'] & N \arrow[d, Rightarrow, "cons"', "\theta"] \\ & N' \end{tikzcd}}{\begin{tikzcd} O \arrow[r, Rightarrow, "\sigma", "cons"'] \arrow[rd, Rightarrow, "\theta\circ\sigma", "cons"'] & N \arrow[d, Rightarrow, "\theta"', "cons"] \\ & N' \end{tikzcd}}$$

(Cons-Composition)

Soundness of **(Cons-Composition)**: any O-model can be σ-expanded to an N-model, which in turn can be θ-expanded to an N'-model. Hence, each O-model can be $\theta \circ \sigma$-expanded to an N'-model. □

$$\dfrac{\begin{tikzcd} O \arrow[r, Rightarrow, "\sigma", "mono"'] & N \arrow[d, Rightarrow, "mono"', "\theta"] \\ & N' \end{tikzcd}}{\begin{tikzcd} O \arrow[r, Rightarrow, "\sigma", "mono"'] \arrow[rd, Rightarrow, "\theta\circ\sigma", "mono"'] & N \arrow[d, Rightarrow, "\theta"', "mono"] \\ & N' \end{tikzcd}}$$

if θ is transportable, any hiding link going directly or indirectly into N' has a transportable signature morphism, and satisfaction in the institution is closed under isomorphism

(Mono-Composition)

For the rule **(Mono-composition)**, we need some technical notion: call a signature morphism $\sigma\colon \Sigma_1 \rightarrow \Sigma_2$ *transportable*, if for any Σ_1-model M_1 and Σ_2-model M_2 and any isomorphism $h_1\colon M_2|_\sigma \rightarrow M_1$, there is a Σ_2-model M_2' and an isomorphism $h_2\colon M_2 \rightarrow M_2'$ with $h_2|_\sigma = h_1$ (which of course includes $M_2'|_\sigma = M_1$). Usually, transportability can be characterized syntactically. For example we have:

Proposition 4.16. *In the* Casl *institution, a signature morphism is transportable iff it is injective on sorts.*

Proof. Let a sort-injective $\sigma\colon \Sigma_1 \rightarrow \Sigma_2$, a Σ_1-model M_1, a Σ_2-model M_2 and an isomorphism $h_1\colon M_2|_\sigma \rightarrow M_1$ be given. M_2' is constructed by taking M_1, and extending it with the carriers of M_2 for sorts in $\Sigma_2 \setminus \Sigma_1$. Operations and predicates in $\Sigma_2 \setminus \Sigma_1$ are interpreted as in M_2, possibly composed with appropriate parts of h_1 whenever sorts from Σ_1 are involved as source or target sorts. This construction works if σ is injective on sorts. If not, take sorts s, t with $\sigma(s) = \sigma(t)$, take M_2 arbitrary and take M_1 as $M_2|_\sigma$ except that t^{M_1} is not s^{M_1}, but is replaced by some isomorphic copy of t^{M_1} (and again operations are composed with this iso if necessary). Then it is impossible to find a Σ_2-expansion of M_1. □

Soundness of **(Mono-Composition)**: we first show that the model class of N' is closed under isomorphism. Let $len(p)$ be the length of a path p witnessing $P \xRightarrow{\tau} N'$. Let $maxp$ be the maximal such length in $\mathcal{DG}$. We show that for any $P \xRightarrow{\tau} N'$, the model class of P is closed under isomorphism. We proceed by induction over $maxp - len(p)$ for p witnessing $P \xRightarrow{\tau} N'$. We have to show that the conditions of clauses 1 to 5 of Definition 4.7 are invariant under model isomorphism:

1. Holding of sentences in a model is invariant under model isomorphism, by the assumption that satisfaction in the institution is closed under isomorphism.
2. Since reduct functors preserve isomorphisms, we can apply the induction hypothesis.
3. Here, a combination of the above two arguments applies.
4. Let $O \xRightarrow[hide]{\sigma} P \in \mathcal{DG}$, M' be a P-model and M'' be isomorphic to M'. Since M' is a P-model, it has a σ-expansion to an O-model M. By transportability of σ, there is a Σ^O-model M' isomorphic to M with $M'|_\sigma = M''$. By induction hypothesis, $\mathbf{Mod}(O)$ is closed under isomorphism, hence $M' \in \mathbf{Mod}(O)$ as well, and thus $M'' \in \mathbf{Mod}(P)$.
5. Freeness is closed under isomorphism.

This completes the induction. Since $N' \xRightarrow{id} N'$, the model class of N' is closed under isomorphism.

Now we come to monomorphicity of the theorem link $O \overset{\theta\circ\sigma}{=\!=\!\Rightarrow} N'$. Let M be an O-model. By the two monomorphicity assumptions, it has at least one N'-expansion. So it remains to prove that all N'-expansions are isomorphic. Let M_3 and M_3' be two N'-expansions of M. By monomorphicity of $O \overset{\sigma}{=\!=\!\Rightarrow} N$, $M_3|_\theta$ and $M_3{}'|_\theta$ are isomorphic. By transportability of θ, there is some M_3'' isomorphic to M_3 with $M_3{}''|_\theta = M_3{}'|_\theta$. Since the model class of N' is closed under isomorphism, M_3'' is an N'-model as well. By monomorphicity of $N \overset{\theta}{=\!=\!\Rightarrow} N'$, M_3'' is isomorphic to M_3'. □

$$\begin{tikzcd} O \arrow[r, Rightarrow, dashed, "\sigma", "def"'] & N \arrow[d, Rightarrow, dashed, "def"', "\theta"] \\ & N' \end{tikzcd}$$

$$\begin{tikzcd} O \arrow[r, Rightarrow, dashed, "\sigma", "def"'] \arrow[dr, Rightarrow, dashed, "\theta\circ\sigma", "def"'] & N \arrow[d, Rightarrow, dashed, "\theta"', "def"] \\ & N' \end{tikzcd}$$

(Def-Composition)

Soundness of **(Def-Composition)**: a global theorem link $O \xRightarrow{\sigma} N$ is definitional iff $\mathbf{Mod}(\sigma)\colon \mathbf{Mod}(N) \to \mathbf{Mod}(O)$ is bijective. Bijective maps compose. □

$$\frac{O \xRightarrow[def]{\sigma} N}{O \xRightarrow[mono]{\sigma} N}$$

(Def-to-mono)

Soundness of **(Def-to-mono)**: obvious. □

$$\frac{O \xRightarrow[mono]{\sigma} N}{O \xRightarrow[cons]{\sigma} N}$$

(Mono-to-cons)

Soundness of **(Mono-to-cons)**: obvious. □

$$\frac{\begin{array}{c} O \xRightarrow[free]{\sigma} N \\ K \xRightarrow[cons]{\sigma} N \end{array}}{K \xRightarrow[mono]{\sigma} N}$$

(Free-is-mono)

Soundness of **(Free-is-mono)**: by the second premise, each K-model has a σ-expansion to an N-model. It remains to show that these σ-expansions are unique up to isomorphism. But this follows since N-models are free (and hence unique up to isomorphism) over their σ-reducts. (Notice that the same signature morphism is used in both premises.) □

$$\frac{\underset{mono}{O} \overset{\sigma}{=\,=\,=\,=\,=\,=\Rightarrow} N}{\underset{mono}{O} \underset{free\ !}{\overset{\sigma}{=\,=\,=\,=\,=\,=\Rightarrow}} N}$$

(Mono-is-free)

Recall that $! \colon \emptyset \to \Sigma^O$ is the signature morphism starting from the initial signature.

Soundness of **(Mono-is-free)**: recall that the free theorem link holds if for any N-model M, $M|_\sigma$ is an O-model that is !-free in $\mathbf{Mod}_{\mathcal{DG}}(O)$. Now for an N-model M, $M|_\sigma$ is an O-model by the premise of the rule, and it is !-free since in a monomorphic model class, any model is initial (and initiality is just !-freeness). □

4.4.3 Simple Structural Rules

The calculus finally provides a set of decomposition rules not interacting with hiding nor freeness, and a rule allowing for reducing local implications to inference in the calculus of the logic for basic specifications.

$$\frac{O \overset{\sigma}{\Longrightarrow} N}{O \overset{\sigma}{=\,=\Rightarrow} N}$$

(Subsumption)

Soundness of **(Subsumption)**: Obvious. □

$$\frac{P \overset{\sigma}{-\,-\succ} Q}{P \overset{\tau}{-\,-\succ} O} \text{ if } Q \overset{\theta}{\Longrightarrow} O \text{ and } \tau(\Psi^P) = \theta(\sigma(\Psi^P))$$

(Loc-Decomposition I)

Soundness of **(Loc-Decomposition I)**: assume $\mathcal{DG} \models P \overset{\sigma}{-\,-\succ} Q$ and $Q \overset{\theta}{\Longrightarrow} O$ and $\tau(\Psi^P) = \theta(\sigma(\Psi^P))$. In order to show $\mathcal{DG} \models P \overset{\tau}{-\,-\succ} O$, let M be an O-model. By Prop. 4.10, $M|_\theta$ is a Q-model, and by the assumption, $M|_{\theta\circ\sigma} \models \Psi^P$. By the satisfaction condition for institutions, $M \models \theta \circ \sigma(\Psi^P) = \tau(\Psi^P)$. Again by the satisfaction condition, $M|_\tau \models \Psi^P$. □

$$\frac{O\!>\!\overset{\theta}{\Longrightarrow}\!\!\!> N}{O \overset{\sigma}{\dashrightarrow} N} \text{ if } \sigma(\Psi^O) = \theta(\Psi^O)$$

(Loc-Decomposition II)

Soundness of **(Loc-Decomposition II)**: assume that $O\!>\!\overset{\theta}{\Longrightarrow}\!\!\!> N$ and $\sigma(\Psi^O) = \theta(\Psi^O)$. Let M be an N-model. By Proposition 4.10, $M|_\theta \models \Psi^O$. By the satisfaction condition for institutions, $M \models \theta(\Psi^O) = \sigma(\Psi^O)$. Again by the satisfaction condition, $M|_\sigma \models \Psi^O$. □

$$\frac{N \Rightarrow \sigma(\Psi^O)}{O \overset{\sigma}{\dashrightarrow} N}$$

(Local Inference)

Soundness of **(Local Inference)**: assume that $M \models \sigma(\Psi^O)$ for each N-model M. In order to show $\mathcal{DG} \models O \overset{\sigma}{\dashrightarrow} N$, let M be an N-model. By assumption, $M \models \sigma(\Psi^O)$. By the satisfaction condition for institutions, $M|_\sigma \models \Psi^O$. □

$$\frac{Th_{\mathcal{DG}}(N) \vdash_{\Sigma^N} \varphi \text{ for each } \varphi \in \Psi}{N \Rightarrow \Psi}$$

(Basic Inference)

Soundness of **(Basic Inference)**: assume that $Th_{\mathcal{DG}}(N) \vdash_{\Sigma^N} \varphi$ for each $\varphi \in \Psi$. By soundness of $\vdash_{\Sigma^N}$, we get $Th_{\mathcal{DG}}(N) \models_{\Sigma^N} \Psi$. In order to show $\mathcal{DG} \models N \Rightarrow \Psi$, let M be an N-model. By Proposition 4.11, $M \models Th_{\mathcal{DG}}(N)$. Since $Th_{\mathcal{DG}}(N) \models_{\Sigma^N} \Psi$, also $M \models \Psi$.

4.5 Soundness and Completeness

Proposition 4.17. *The rules in Sect. 4.4 are sound.*

Proof. For each rule, in Sect. 4.4, a soundness proof has been given.

Another question is the completeness of our rules. We have the following counterexample:

Proposition 4.18. *Let FOL be the usual first-order logic with a recursively axiomatized complete entailment system. The problem to decide whether a global theorem link holds in a development graph with hiding over FOL is not recursively enumerable. Thus, any recursively axiomatized calculus for development graphs with hiding is incomplete.*

Proof. This can be seen as follows. Let Σ be the FOL-signature with a sort nat and operations for zero and successor, addition and multiplication. Consider the axiom set consisting of the usual second-order Peano axioms characterizing the natural numbers uniquely up to isomorphism, plus the defining axioms for addition and multiplication. Without loss of generality, we can assume that these axioms are combined into a single axiom of the form

$$\forall P : pred(nat) \ . \ \varphi$$

where φ is a first-order formula. Let ψ be any sentence over Σ. Let $\theta \colon \Sigma \to \Sigma'$ add a predicate $P : pred(nat)$ to Σ. Consider the development graph

$$\begin{tikzcd}
\textsc{Peano} \arrow[d, Rightarrow, dashed, "id"] & \textsc{PeanoDef} \arrow[l, Rightarrow, "h"', "\theta"] \\
\Sigma &
\end{tikzcd}$$

where Σ and Peano are nodes with signature Σ and no local axioms, whereas PeanoDef is a node with signature Σ' and local axiom $\varphi \Rightarrow \psi$.

Now we have that $\textsc{Peano} \overset{id}{=\!=\!\Rightarrow} \Sigma$ holds iff each Σ-model has a PeanoDef-expansion. It is easy to see that this holds iff the second-order formula $\exists P : pred(nat). \varphi \Rightarrow \psi$ is valid. This is equivalent to $(\forall P : pred(nat). \varphi) \models \psi$, i.e. equivalent to the fact that ψ holds in the second-order axiomatization of Peano arithmetic. By Gödels incompleteness theorem, the problem to decide whether this holds is not recursively enumerable. □

In spite of this negative result, there is still the question of a relative completeness w.r.t. a given oracle deciding conservative extensions. Such a completeness result has been proved by Borzyszkowski [7] in a similar setting. We are going to state an analogous result here, which additionally is based on oracles for freeness (the latter has not been covered by Borzyszkowski).

An *oracle for conservative extensions* is a sound logic-specific rule that allows to infer conservativity annotations for global definition links. It is called complete if for any global definition links that enjoys the model expansion property, the conservativity annotation may actually be inferred.

An *oracle for free theorem links* is a sound logic-specific rule that allows to infer free theorem links. It is called complete if any free theorem link that semantically holds also can be inferred by the rule.

An *elimination oracle for free definition links* is a sound logic-specific rule of the form

$$\frac{O \overset{\sigma}{=\!=\!\Rightarrow} K}{O \overset{\sigma}{=\!=\!\Rightarrow} N}$$

where $O \overset{\sigma}{=\!=\!\Rightarrow} N$ is arbitrary and K is constructed out of N such that K does not contain any directly or indirectly ingoing free definition links. Here, soundness just means $\mathbf{Mod}(N) \subseteq \mathbf{Mod}(K)$. Such a rule is called *complete*, if also $\mathbf{Mod}(K) \subseteq \mathbf{Mod}(N)$.

Theorem 4.19 (Completeness). *Assume that the underlying logic is complete. Then the rule system for development graphs with hiding is complete relative to complete oracles for conservative extensions and free theorem links and a complete elimination oracle for free definition links.*

Proof. See [44].

□

Corollary 4.20. *If the underlying logic is complete, the simple structural rules are complete for proving theorem links between flattenable nodes.*

Proof. See [44]. □

We should note that a complete oracle for conservative extensions is very powerful: it can be used to obtain a complete proof calculus for development graphs. Namely, in order to decide whether $\mathcal{DG} \models O \overset{\sigma}{=\,=\!\Rightarrow} N$, we just add a node P with

$$\begin{tikzcd}
O \arrow[dr, "\sigma"] & & N \arrow[dl, "id"] \\
& P &
\end{tikzcd}$$

and ask the oracle whether $N \xRightarrow{id} P$ is conservative.

Nevertheless, our completeness theorem is still meaningful. This is because the completeness proof uses the oracle for conservative extensions only in a limited way. The extensions considered are those obtained from hiding theorem links in the development graph (pushed along some morphism into a 'big' signature collecting everything). This means, for example, if we use hiding links only to hide symbols that have been defined using some logic-specific definition scheme, we will need the oracle for conservative extensions only for checking this definition scheme. This will be important in the next section.

4.6 Checking Conservativity and Freeness

Casl has annotations expressing that an extension of a specification is *conservative*, *monomorphic* or *definitional*, meaning that every model of the 'small' specification can be expanded to some model, some model unique up to isomorphism, or some unique model, of the 'large' specification, respectively. Moreover, as can be seen from the calculus studied in the previous sections, checks for conservative extensions already arise from the presence of hiding. Furthermore, during the development process, it may be desirable to check the specification for *consistency* at an early stage – and consistency is just conservativity over the empty specification. Finally, using consistency, also *non-consequence* can be checked: an axiom does *not* follow from a specification if the specification augmented by the negation of the axiom is consistent.

So far there is no hope to tackle these questions in an institution independent way. Therefore, in this section we deal with the specific institution of Casl only. However, unfortunately already for first-order logic, neither the check for conservative, nor monomorphic, nor definitional extension are recursively enumerable, which means that there cannot be a complete (recursively axiomatized) calculus for them. For conservativity, this follows from Theorem 4.19 and the proof of Theorem 4.18: a recursively axiomatized calculus for conservativity would provide the needed oracle for Theorem 4.19, contradicting the example from the proof of Theorem 4.18.

Although there is no general approach to verify that an extension of a specification is conservative (or monomorphic, or definitional), several schemes for extending specifications have been developed in the past which guarantee these properties by construction. We only very informally list some possible rules here:

- extensions declaring new signature elements are conservative, provided the new symbols are not constrained in any way (by axioms, by requirements on the subsort and overloading relations, etc.) to be related to old symbols, and the new symbols themselves are constrained by a positive theory (i.e. not involving negation),
- free datatypes are monomorphic extensions of the local environemnt in which they are introduced,
- structured free Horn theories are monomorphic extensions,
- subsort definitions are definitional extensions, and
- inductive definitions over free datatypes are definitional extensions.

There is no hope to tackle freeness in an institution independent way either. But is is possible to define Casl-specific oracles for free theorem links and elimination of free definition links. They basically introduce a new node that is used to mimic the quotient term algebra construction.

4.7 Translation from Structured Specifications to Development Graphs

Roughly speaking, the translation of some Casl-specification `SPEC` to a development graph works as follows:

- it maps basic specifications, like the specification of simple abstract datatypes, into development graph nodes that are labelled with the signature and the axioms of the basic specification,
- it translates the structuring operations of Casl into definition links, and

- it reformulates proof obligations given in the specification either into theorem links connecting corresponding nodes or into local implications in the development graph.

Obviously, the definition of such a translation entails the requirement to prove the adequacy of the translation of CASL-specifications into development graphs.

We first informally explain how the transformation works, using a set of rewrite rules shown in Figs. 4.1, 4.2 and 4.3. To translate a CASL construct one starts with a *pre-development graph* consisting of a node which contains the (not-yet translated) CASL construct. In the figures, nodes which are not yet translated are represented as shaded boxes, translated nodes as circles.

A pre-development graph is then processed by successively rewriting the boxes occurring in it, using the rewrite rules in Figs. 4.1, 4.2 and 4.3. These boxes are decomposed until one eventually arrives at a graph in which there are no boxes left, i.e. a development graph representation. During this process, boxes may be created which have in-going or out-going links. The thick arrows indicate how these links are inherited when such a box is rewritten.

The formal translation is defined by extending the static semantic rules for structured specifications to a *verification semantics*. In the verification semantics, signatures are replaced with nodes in a global development graph, where the latter is being carried around as an additional parameter. The intention is that the development graph captures the semantics of the specification, while a set of theorem links over this graph captures the proof obligations that have to be discharged in order to ensure that the model semantics succeeds. Note that in this point the verification semantics goes beyond the aims of the extended static semantics for architectural specifications introduced in Sect. III:5.6: the latter captures the sharing arising in architectural specifications, but not the model semantics of specifications. A verification semantics for architectural specification will be sketched in Chap. 6.

4.7.1 Concepts for the Verification Semantics

We now modify a number of concepts from the static semantics of structured specification (Sect. III:4.1.5). Basically, signatures are replaced with nodes in a development graph. By abuse of language, a pair $(\mathcal{DG}, \mathit{Th})$, with $\mathcal{DG}$ a development graph and Th a set of theorem links over $\mathcal{DG}$, is called a development graph as well. It will be always clear from the context whether a development graph comes with a set of theorem links or not.

A *verification generic signature* GS_s over a development graph consists of two nodes (the import and the body) and a sequence of nodes (the formal parameters) in the development graph.

$$GS_s \in \mathit{VerGenSig} = \mathit{Node} \times \mathit{FinSeq}(\mathit{Node}) \times \mathit{Node}$$

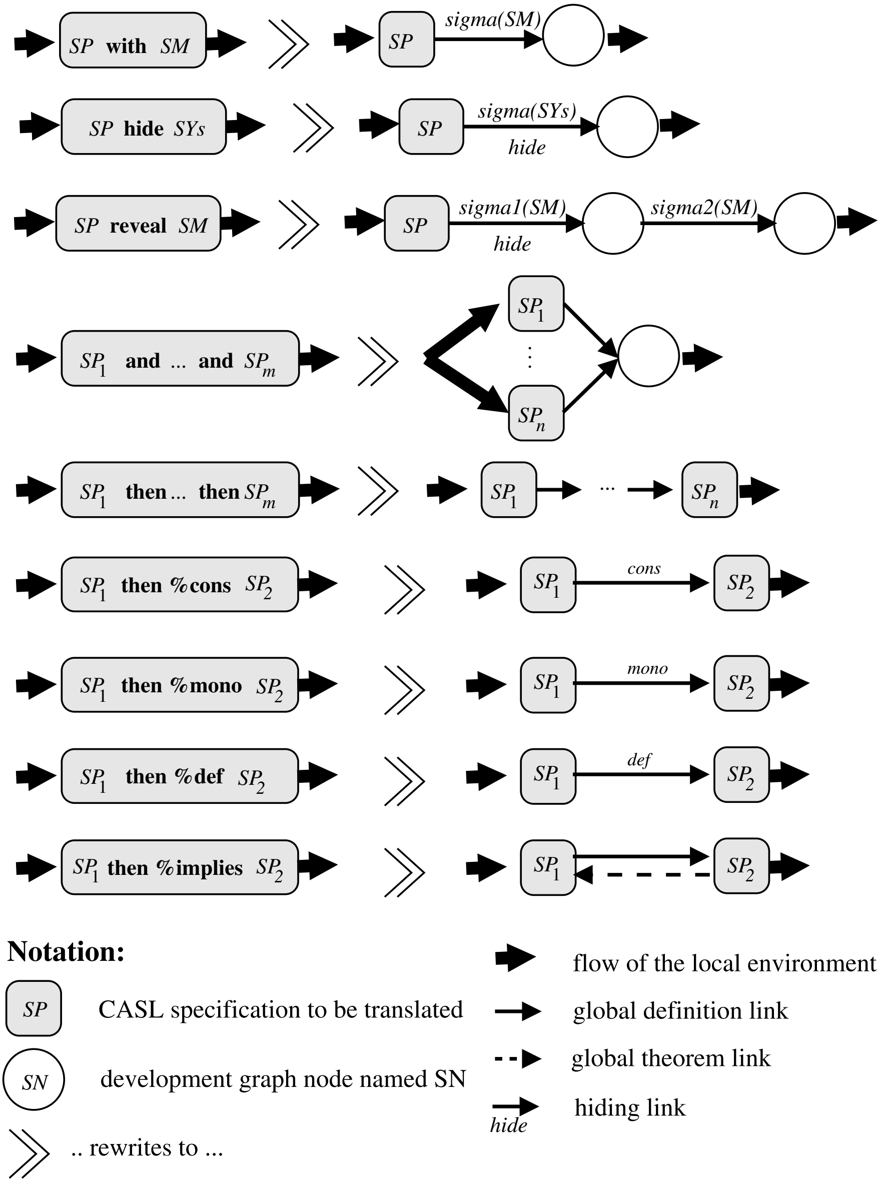

Fig. 4.1. Translating Casl specifications into development graphs with hiding, part 1: translations, reductions, unions and extensions

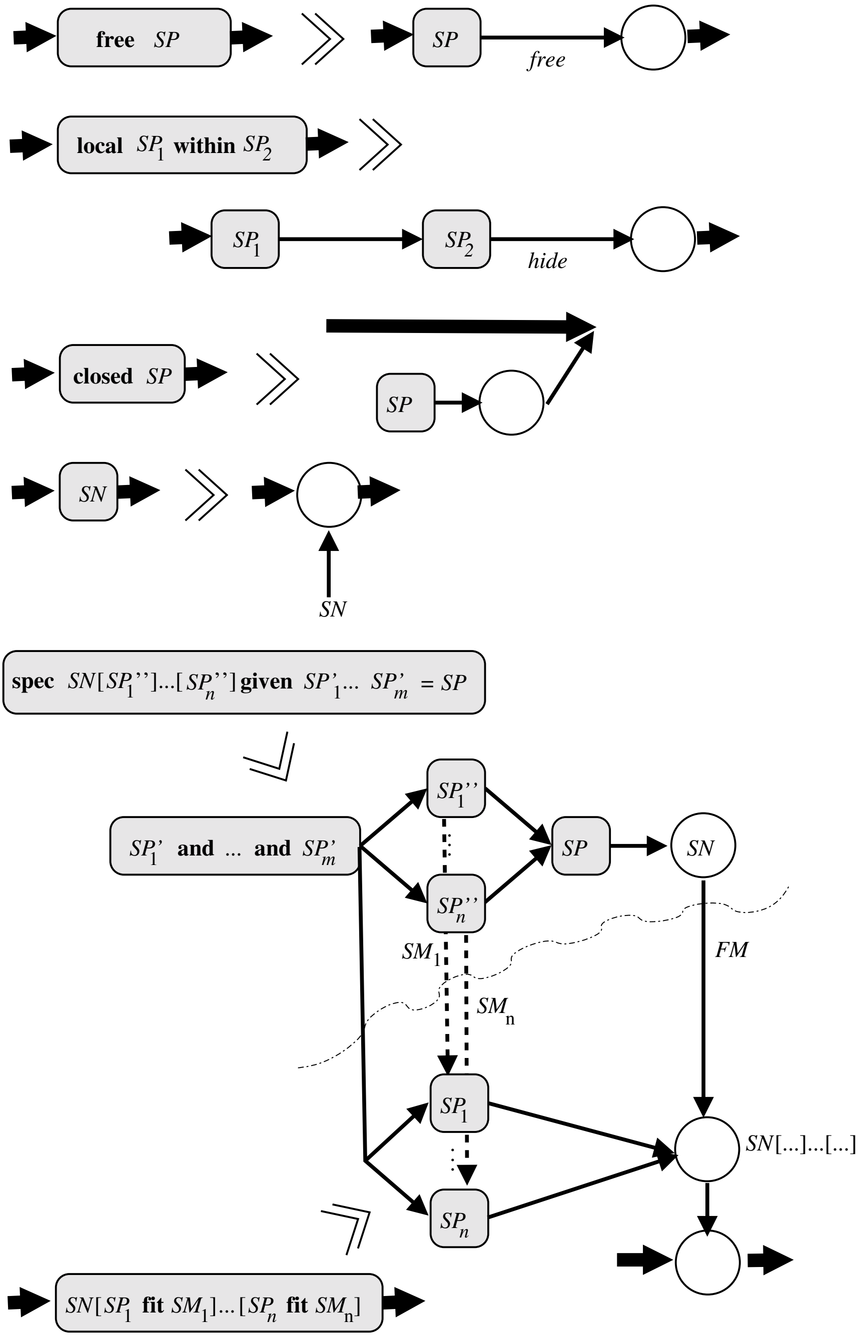

Fig. 4.2. Translating CASL specifications into development graphs with hiding, part 2: free, local, closed and generic specifications

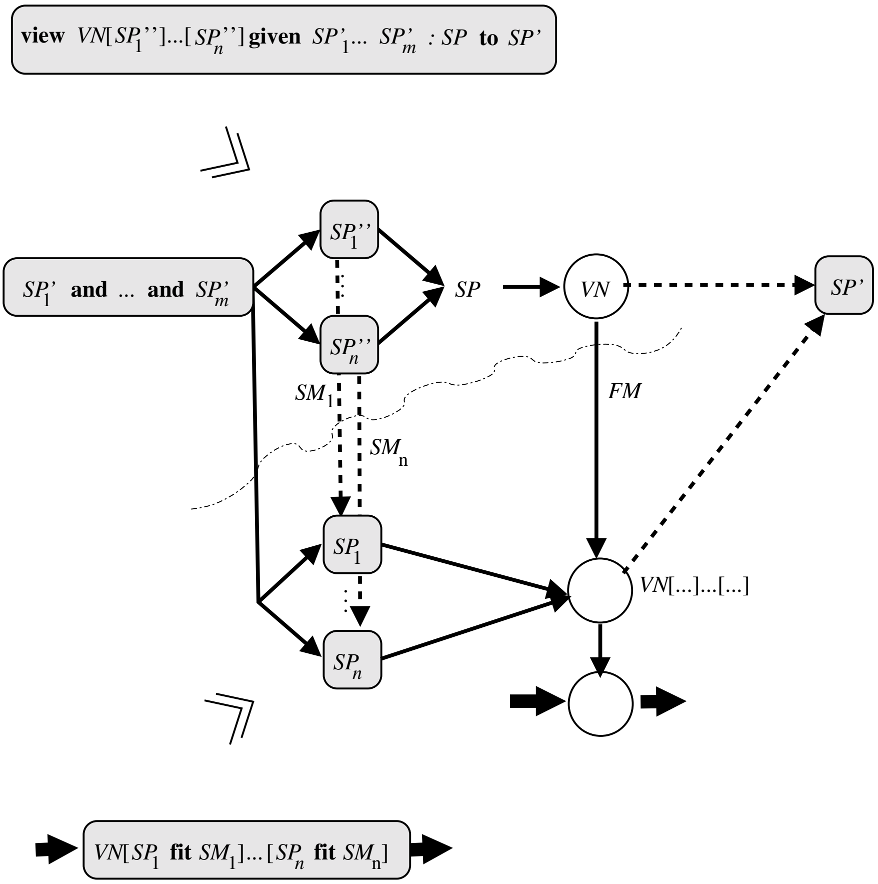
view VN[SP1'']...[SPn''] given SP'1... SP'm : SP to SP'
SP'1 and ... and SP'm
SP''1
SP''n
SP
VN
SP'
SM1
SMn
FM
SP1
SPn
VN[...]...[...]
VN[SP1 fit SM1]...[SPn fit SMn]

Fig. 4.3. Translating CASL specifications into development graphs with hiding, part 3: views

The requirement on a generic signature $GS_s = (N_I, \langle N_1, \ldots, N_n\rangle, N_B)$ over a development graph $(\mathcal{DG}, \mathit{Th})$ is that all the nodes involved in GS_s are contained in $(\mathcal{DG}, \mathit{Th})$, and that

$$\mathit{strip}(GS_s) = (\Sigma^{N_I}, \langle \Sigma^{N_1}, \ldots, \Sigma^{N_n}\rangle, \Sigma^{N_B})$$

is indeed an (ordinary) generic signature in the sense of Sect. III:4.1.5.

The notion of compatibility between signatures and models can be carried over: given an element $GS_m = (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_m\rangle, \mathcal{M}_B)$ of **GenSpec** (cf. Sect. III:4.1.5) and a verification generic signature $GS_s = (N_I, \langle N_1, \ldots, N_n\rangle, N_B)$ over a development graph $(\mathcal{DG}, \mathit{Th})$, we say that GS_s and GS_m are *compatible*, if

- $m = n$,
- $\mathbf{Mod}_{\mathcal{DG}}(N_I) = \mathcal{M}_I$,
- $\mathbf{Mod}_{\mathcal{DG}}(N_i) = \mathcal{M}_i$ $(i = 1, \ldots, n)$, and
- $\mathbf{Mod}_{\mathcal{DG}}(N_B) = \mathcal{M}_B$.

A *verification view signature* consists of a node (the source node), a signature morphism, and a verification generic signature (the target of the view).

$$V_s \in \mathit{VerViewSig} = \mathit{Node} \times \mathit{SignatureMorphism} \times \mathit{VerGenSig}$$

For a verification view signature $V_s = (N_s, \sigma, \mathit{GS}_s)$ in *VerViewSig* over a development graph $(\mathcal{DG}, \mathit{Th})$, we require that all the nodes involved in V_s are contained in $(\mathcal{DG}, \mathit{Th})$, and that

$$\mathit{strip}(V_s) = (\Sigma^{N_s}, \sigma, \mathit{strip}(\mathit{GS}_s))$$

is an element of *ViewSig* as defined in Sect. III:4.1.5.

Given an element $V_m = (\mathcal{M}_s, \mathit{GS}_m)$ of **ViewSpec** and a verification view signature $V_s = (N_s, \sigma, \mathit{GS}_s)$ over a development graph $(\mathcal{DG}, \mathit{Th})$, we say that V_s and V_m are *compatible*, if

- $\mathbf{Mod}_{\mathcal{DG}}(N_s) = \mathcal{M}_s$,
- $\mathit{GS}_s = (N_I, \langle N_1, \ldots, N_n \rangle, N_B)$ and $\mathit{GS}_m = (\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_m \rangle, \mathcal{M}_B)$ are compatible, and
- $\mathcal{M}_B|_\sigma \subseteq \mathcal{M}_s$.

We now come to verification global environments. Apart from the usual global environment components (cf. Sect. III:6.1), these need to carry around a global development graph together with a set of theorem links over this graph. This is necessary in order to achieve the intended sharing between different instantiations of one and the same generic specification. The global development graph is successively extended by the rules for specifications. This finally leads to an update of the global development graph in the rules for specification and view definitions. We also assume that the global development graph always contains a node named $\emptyset$ which consists of the empty signature and the empty set of axioms.

A *verification global environment*

$$\Gamma_s, (\mathcal{DG}, \mathit{Th}) = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s), (\mathcal{DG}, \mathit{Th})$$

consists of a development graph $(\mathcal{DG}, \mathit{Th})$ and finite functions from names to the *verification* denotations of generic specifications, views, architectural specifications and unit specifications (cf. Chap. III:6):

- $\mathcal{G}_s : \mathit{SpecName} \overset{\text{fin}}{\to} \mathit{VerGenSig}$
- $\mathcal{V}_s : \mathit{ViewName} \overset{\text{fin}}{\to} \mathit{VerViewSig}$
- $\mathcal{A}_s : \mathit{ArchSpecName} \overset{\text{fin}}{\to} \mathit{VerArchSig}$
- $\mathcal{T}_s : \mathit{UnitSpecName} \overset{\text{fin}}{\to} \mathit{VerUnitSig}$

The domains *VerGenSig* and *VerViewSig* have been defined above, while *VerArchSig* and *VerUnitSig* will be defined in Chap. 6.

Requirements on a verification global environment $(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)$, $(\mathcal{DG}, Th)$: the domains of the various components are disjoint $(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)$, and each involved verification signature (i.e. verification generic signature, verification view signature etc.) is well-formed over $(\mathcal{DG}, Th)$.

Both the stripping operations and the compatibility relation extend to global environments in a pointwise way (domains of compatible environments have to be equal).

In the rules below, we often use the notation Σ^N and Ψ^N, which of course only makes sense only relative to a given development graph. Which development graph is meant can be seen from the descriptions of the formats of the judgements: there, for each node the development graph it lives in is mentioned. Moreover, the development graphs occurring in a rule are all extensions of another, such that any of them can be equally chosen to compute Σ^N and Ψ^N, as long as N is contained in the graph.

We say that a development graph $\mathcal{DG}'$ *extends* a development graph $\mathcal{DG}$ if $\mathcal{DG}$ is a subgraph of $\mathcal{DG}'$ in the obvious sense. Moreover, we say that a family of development graphs $\mathcal{DG}_i$, $i \in I$ (for an arbitrary index set I) *disjointly extends* $\mathcal{DG}$ if each $\mathcal{DG}_i$ extends $\mathcal{DG}$ (for $i \in I$) and moreover, for all distinct $i, j \in I$, $\mathcal{DG}_i \cap \mathcal{DG}_j = \mathcal{DG}$. If this is the case, then the union $\bigcup_{i \in I} \mathcal{DG}_i$ is well-defined and extends $\mathcal{DG}$.

Whenever such a disjointness requirement appears in the rules below, in principle it could be eliminated by appropriate choices of new names. However, spelling this out would only add uninteresting detail, which we omit here and state the disjointness requirement explicitly.

In the verification semantics, we use judgements of form

$$\mathit{context} \vdash \mathit{phrase} \rhd\!\!\rhd\!\!\rhd \mathit{result}.$$

We will rely on notions and judgements (of form *context* $\vdash$ *phrase* $\rhd$ *result*) of the static semantics of structured specifications, which the reader should consult whenever necessary (Chap. III:4).

For readability, the definition and theorem links are decorated with $\Sigma \hookrightarrow \Sigma'$ whenever $\iota_{\Sigma \subseteq \Sigma'}$ is meant (which of course assumes that $\Sigma \subseteq \Sigma'$).

4.7.2 Structured Specifications

$$\boxed{N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{SPEC} \rhd\!\!\rhd\!\!\rhd N', (\mathcal{DG}', Th)}$$

$\Gamma_s, (\mathcal{DG}, Th)$ is a verification global environment, N is a node in $\mathcal{DG}$; then $(\mathcal{DG}', Th')$ is a development graph extending $(\mathcal{DG}, Th)$, N' a node in $\mathcal{DG}'$, and $\Sigma^{N'}$ is an extension of Σ^N.

Basic Specifications

$$\frac{\begin{array}{c}\Sigma^{basic} \vdash \texttt{BASIC-SPEC} \rhd (\Delta, \Psi) \\ \Sigma^N = (\Sigma^{basic}, SSY) \qquad \Sigma' = (\Sigma^{basic} \cup \Delta, SSY \cup |\Delta|)\end{array}}{\begin{array}{l} N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{BASIC-SPEC}\ \text{qua}\ \texttt{SPEC} \ggg \\ \qquad\qquad N', (\mathcal{DG} \uplus \{N' \text{ with } (\Sigma', \Psi);\ N \xRightarrow{\Sigma \hookrightarrow \Sigma'} N'\}, Th)\end{array}}$$

Translations

$$\frac{\begin{array}{c} N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{SPEC} \ggg N', (\mathcal{DG}', Th') \\ \Sigma^{N'} \vdash \texttt{RENAMING} \rhd \sigma\colon \Sigma^{N'} \to \Sigma'' \\ |\sigma| \text{ is the identity on signature symbols in } |\Sigma^N| \end{array}}{\begin{array}{l} N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{translation}\ \texttt{SPEC}\ \texttt{RENAMING} \ggg \\ \qquad N'', (\mathcal{DG}' \uplus \{N'' \text{ with } (\Sigma'', \emptyset);\ N' \xRightarrow{\sigma} N''\}, Th') \end{array}}$$

Reductions

$$\frac{\begin{array}{c} N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{SPEC} \ggg N', (\mathcal{DG}', Th') \\ (\Sigma^N, \Sigma^{N'}) \vdash \texttt{RESTRICTION} \rhd \sigma\colon \Sigma_1 \to \Sigma'' \\ \mathcal{DG}'' = \mathcal{DG}' \uplus \{\ N_1 \text{ with } (\Sigma_1, \emptyset);\ N'' \text{ with } (\Sigma'', \emptyset); \\ \qquad\qquad N' \xRightarrow[hide]{\Sigma_1 \hookrightarrow \Sigma^{N'}} N_1;\ N_1 \xRightarrow{\sigma} N''\} \end{array}}{N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{reduction}\ \texttt{SPEC}\ \texttt{RESTRICTION} \ggg N'', (\mathcal{DG}'', Th')}$$

Unions

$$\frac{\begin{array}{c} N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{SPEC}_1 \ggg N_1, (\mathcal{DG}_1, Th_1) \\ \ldots \\ N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{SPEC}_n \ggg N_n, (\mathcal{DG}_n, Th_n) \\ \Sigma' = \Sigma^{N_1} \cup \ldots \cup \Sigma^{N_n} \\ \mathcal{DG}_1, \ldots, \mathcal{DG}_n \text{ disjointly extend } \mathcal{DG} \\ \mathcal{DG}' = \bigcup_{i=1,\ldots,n} \mathcal{DG}_i \uplus \{N' \text{ with } (\Sigma', \emptyset)\} \uplus \{\,N_i \xRightarrow{\Sigma^{N_i} \hookrightarrow \Sigma'} N' | i = 1, \ldots, n\} \\ Th' = \bigcup_{i=1,\ldots,n} Th_i \end{array}}{N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{union}\ \texttt{SPEC}_1 \ldots \texttt{SPEC}_n \ggg N', (\mathcal{DG}', \Sigma', Th')}$$

Extensions

$$\frac{\begin{array}{c} N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{SPEC}_1 \ggg N_1, (\mathcal{DG}_1, Th_1) \\ \ldots \\ N_{n-1}, \Gamma_s, (\mathcal{DG}_{n-1}, Th_{n-1}) \vdash \texttt{SPEC}_n \ggg N_n, (\mathcal{DG}_n, Th_n) \end{array}}{N, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{extension}\ \texttt{SPEC}_1 \ldots \texttt{SPEC}_n \ggg N_n, (\mathcal{DG}_n, Th_n)}$$

Moreover, for each extension from $\mathtt{SPEC}_{i-1}$ to $\mathtt{SPEC}_i$ that has been annotated to be conservative, monomorphic, or definitional, a proof obligation $N_{i-1} \overset{\Sigma^{N_{i-1}} \hookrightarrow \Sigma^{N_i}}{\underset{cons}{=\,=\,=\,=\,=\!\Rightarrow}} N_i$, $N_{i-1} \overset{\Sigma^{N_{i-1}} \hookrightarrow \Sigma^{N_i}}{\underset{mono}{=\,=\,=\,=\,=\!\Rightarrow}} N_i$ or $N_{i-1} \overset{\Sigma^{N_{i-1}} \hookrightarrow \Sigma^{N_i}}{\underset{def}{=\,=\,=\,=\,=\!\Rightarrow}} N_i$, respectively, should be added to Th_i. (Note that CASL requires that symbols from the local environment must not be affected by hidings; therefore, we always have $\Sigma^{N_{i-1}} \subseteq \Sigma^{N_i}$, and even $N_{i-1} \overset{\Sigma^{N_{i-1}} \hookrightarrow \Sigma^{N_i}}{=\,=\,=\,=\,=\!\Rightarrow} N_i$ can easily be seen to hold.) Finally, for an extension from $\mathtt{SPEC}_{i-1}$ to $\mathtt{SPEC}_i$ that has been annotated to be implied, a proof obligation $N_i \overset{id}{=\,=\!\Rightarrow} N_{i-1}$ is added (note that for such an annotation, $\Sigma^{N_{i-1}} = \Sigma^{N_i}$ is required). In the case that the extension from $\mathtt{SPEC}_{i-1}$ to $\mathtt{SPEC}_i$ is simply a basic specification, a proof obligation $N_{i-1} \Rightarrow \Psi^{N_i}$ suffices instead (but note that in general, $\mathtt{SPEC}_i$ may involve hiding etc.).

Free Specifications

$$\frac{N, \Gamma_s, (\mathcal{DG}, Th) \vdash \mathtt{SPEC} \mathrel{\rhd\!\!\rhd\!\!\rhd} N', (\mathcal{DG}', Th')}{\begin{array}{l} N, \Gamma_s, (\mathcal{DG}, Th) \vdash \mathtt{free\text{-}spec}\ \mathtt{SPEC} \mathrel{\rhd\!\!\rhd\!\!\rhd} \\ \qquad N'', (\mathcal{DG}' \uplus \{N'' \text{ with } (\Sigma^{N'}, \emptyset);\ N' \xRightarrow[free]{\Sigma^N \hookrightarrow \Sigma^{N'}} N''\}, Th') \end{array}}$$

Local Specifications

$$\frac{\begin{array}{c} N, \Gamma_s, (\mathcal{DG}, Th) \vdash \mathtt{SPEC} \mathrel{\rhd\!\!\rhd\!\!\rhd} N', (\mathcal{DG}', Th') \\ N', \Gamma_s, (\mathcal{DG}', Th') \vdash \mathtt{SPEC'} \mathrel{\rhd\!\!\rhd\!\!\rhd} N'', (\mathcal{DG}'', Th'') \\ \Sigma_1 = \Sigma^{N''}|^{|\Sigma^{N'}| \setminus |\Sigma^N|} \qquad |\Sigma^{N''}| \setminus |\Sigma^{N'}| \subseteq |\Sigma_1| \end{array}}{\begin{array}{l} N, \Gamma_s, (\mathcal{DG}, Th) \vdash \mathtt{local\text{-}spec}\ \mathtt{SPEC}\ \mathtt{SPEC'} \mathrel{\rhd\!\!\rhd\!\!\rhd} \\ \qquad N''', (\mathcal{DG}'' \uplus \{N''' \text{ with } (\Sigma_1, \emptyset);\ N'' \xRightarrow[hide]{\Sigma_1 \hookrightarrow \Sigma^{N''}} N'''\}, Th'') \end{array}}$$

Closed Specifications

$$\frac{\begin{array}{c} \emptyset, \Gamma_s, (\mathcal{DG}, Th) \vdash \mathtt{SPEC} \mathrel{\rhd\!\!\rhd\!\!\rhd} N', (\mathcal{DG}', Th') \\ \Sigma'' = \Sigma^N \cup \Sigma^{N'} \\ \mathcal{DG}'' = \mathcal{DG}' \uplus \{N'' \text{ with } (\Sigma'', \emptyset);\ N \xRightarrow{\Sigma^N \hookrightarrow \Sigma''} N'';\ N' \xRightarrow{\Sigma^{N'} \hookrightarrow \Sigma''} N''\} \end{array}}{N, \Gamma_s, (\mathcal{DG}, Th) \vdash \mathtt{closed\text{-}spec}\ \mathtt{SPEC} \mathrel{\rhd\!\!\rhd\!\!\rhd} N'', (\mathcal{DG}'', Th \cup Th')}$$

4.7.3 Named and Generic Specifications

Specification Definitions

$$\boxed{\Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{SPEC\text{-}DEFN} \mathrel{\rhd\!\!\rhd\!\!\rhd} \Gamma_s', (\mathcal{DG}', \mathit{Th}')}$$

$\Gamma_s, (\mathcal{DG}, \mathit{Th})$ is a verification global environment; then $\Gamma_s', (\mathcal{DG}', \mathit{Th}')$ is a verification global environment extending $\Gamma_s, (\mathcal{DG}, \mathit{Th})$.

$$\frac{\begin{array}{c} \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \\ \mathit{SN} \notin \operatorname{dom} \mathcal{G}_s \cup \operatorname{dom} \mathcal{V}_s \cup \operatorname{dom} \mathcal{A}_s \cup \operatorname{dom} \mathcal{T}_s \\ \Gamma_s \vdash \mathtt{GENERICITY} \mathrel{\rhd\!\!\rhd\!\!\rhd} (N_I, \langle N_1, \ldots, N_n \rangle, N_P), (\mathcal{DG}_P, \mathit{Th}_P) \\ N_P, \Gamma_s, (\mathcal{DG}_P, \mathit{Th}_P) \vdash \mathtt{SPEC} \mathrel{\rhd\!\!\rhd\!\!\rhd} N_B, (\mathcal{DG}_B, \mathit{Th}_B) \end{array}}{\begin{array}{l} \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{spec\text{-}defn}\ \mathit{SN}\ \mathtt{GENERICITY}\ \mathtt{SPEC} \mathrel{\rhd\!\!\rhd\!\!\rhd} \\ \qquad (\mathcal{G}_s \cup \{\mathit{SN} \mapsto (N_I, \langle N_1, \ldots, N_n \rangle, N_B)\}, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s), (\mathcal{DG}_B, \mathit{Th}_B) \end{array}}$$

Genericity: Parameters and Imports

$$\boxed{\Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{GENERICITY} \mathrel{\rhd\!\!\rhd\!\!\rhd} (N_I, \langle N_1, \ldots, N_n \rangle, N_P), (\mathcal{DG}_P, \mathit{Th}_P)}$$

$\Gamma_s, (\mathcal{DG}, \mathit{Th})$ is a verification global environment; then $(\mathcal{DG}_P, \mathit{Th}_P)$ is a development graph extending $(\mathcal{DG}, \mathit{Th})$, N_I, N_P and the N_i are nodes in $\mathcal{DG}_P$, such that Σ^{N_i} is an extension of Σ^{N_I}, and Σ^{N_P} is the union of the Σ^{N_i} $(i = 1, \ldots, n)$.

$$\frac{\begin{array}{c} \Gamma_s \vdash \mathtt{IMPORTS} \mathrel{\rhd\!\!\rhd\!\!\rhd} N_I, (\mathcal{DG}, \mathit{Th}) \\ N_I, \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{PARAMS} \mathrel{\rhd\!\!\rhd\!\!\rhd} \langle N_1, \ldots, N_n \rangle, (\mathcal{DG}_P, \mathit{Th}_P) \qquad n \geq 1 \\ \Sigma_P = \Sigma^{N_1} \cup \cdots \cup \Sigma^{N_n} \\ \mathcal{DG}_P' = \mathcal{DG}_P \uplus \{N_P \text{ with } (\Sigma_P, \emptyset)\} \uplus \{\, N_i \xRightarrow{\Sigma^{N_i} \hookrightarrow \Sigma_P} N_P \mid i = 1, \ldots, n\} \end{array}}{\begin{array}{l} \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{genericity}\ \mathtt{PARAMS}\ \mathtt{IMPORTS} \mathrel{\rhd\!\!\rhd\!\!\rhd} \\ \qquad\qquad (N_I, \langle N_1, \ldots, N_n \rangle, N_P), (\mathcal{DG}_P', \mathit{Th}_P) \end{array}}$$

$$\frac{\begin{array}{c} \Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \\ \Gamma_s \vdash \mathtt{IMPORTS} \mathrel{\rhd\!\!\rhd\!\!\rhd} \emptyset, (\mathcal{DG}, \mathit{Th}) \\ \emptyset, \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{PARAMS} \mathrel{\rhd\!\!\rhd\!\!\rhd} \langle\rangle, (\mathcal{DG}, \mathit{Th}) \end{array}}{\Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{genericity}\ \mathtt{PARAMS}\ \mathtt{IMPORTS} \mathrel{\rhd\!\!\rhd\!\!\rhd} (\emptyset, \langle\rangle, \emptyset), (\mathcal{DG}, \mathit{Th})}$$

Parameters

$$\boxed{N_I, \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{PARAMS} \mathrel{\rhd\!\!\rhd\!\!\rhd} \langle N_1, \ldots, N_n \rangle, (\mathcal{DG}_P, \mathit{Th}_P)}$$

$\Gamma_s, (\mathcal{DG}, \mathit{Th})$ is a verification global environment and N_I is a node in $\mathcal{DG}$; then $(\mathcal{DG}_P, \mathit{Th}_P)$ is a development graph extending $(\mathcal{DG}, \mathit{Th})$, the N_i are nodes in $\mathcal{DG}_P$ $(i = 1, \ldots, n)$, and Σ^{N_i} is an extension of Σ^{N_I} for $1 \leq i \leq n$.

$$\frac{\begin{array}{c} N_I, \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{SPEC}_1 \ggg N_1, (\mathcal{DG}_1, \mathit{Th}_1) \\ \ldots \\ N_I, \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{SPEC}_n \ggg N_n, (\mathcal{DG}_n, \mathit{Th}_n) \\ \mathcal{DG}_1, \ldots, \mathcal{DG}_n \text{ disjointly extend } \mathcal{DG} \end{array}}{\begin{array}{l} N_I, \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{params}\, \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \ggg \\ \qquad \langle N_1, \ldots, N_n \rangle, (\bigcup_{i=1\ldots,n} \mathcal{DG}_i, \bigcup_{i=1\ldots,n} \mathit{Th}_i) \end{array}}$$

Imports

$$\boxed{\Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{IMPORTS} \ggg N, (\mathcal{DG}', \mathit{Th}')}$$

$\Gamma_s, (\mathcal{DG}, \mathit{Th})$ is a verification global environment; then $(\mathcal{DG}', \mathit{Th}')$ a development graph extending $(\mathcal{DG}, \mathit{Th})$ and N is a node in $\mathcal{DG}'$.

If the sequence of imported specifications is empty, then the semantics of the import construct is the empty environment $\emptyset$, and the development graph is from the global environment not extended but just retained.

$$\frac{}{\Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{imports} \ggg \emptyset, (\mathcal{DG}, \mathit{Th})}$$

$$\frac{\begin{array}{c} \emptyset, \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{union}\, \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \ggg N_I, (\mathcal{DG}_I, \mathit{Th}_I) \\ \sigma = \iota_{\mathit{EmptyExplicit}(\Sigma^{N_I}) \subseteq \Sigma^{N_I}} \end{array}}{\begin{array}{l} \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{imports}\, \mathtt{SPEC}_1 \ldots \mathtt{SPEC}_n \ggg \\ \qquad N, (\mathcal{DG}_I \uplus \{N \text{ with } (\mathit{EmptyExplicit}(\Sigma^{N_I}), \emptyset);\;\, N_I \xRightarrow[\mathit{hide}]{\sigma} N\}, \mathit{Th}_I) \end{array}}$$

Specification Instantiation

We repeat the rule format for structured specifications:

$$\boxed{N, \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{SPEC} \ggg N', (\mathcal{DG}', \mathit{Th})}$$

$\Gamma_s, (\mathcal{DG}, \mathit{Th})$ is a verification global environment, N is a node in $\mathcal{DG}$; then $(\mathcal{DG}', \mathit{Th}')$ is a development graph extending $(\mathcal{DG}, \mathit{Th})$, N' a node in $\mathcal{DG}'$, and $\Sigma^{N'}$ is an extension of Σ^N.

$$\frac{\begin{array}{c} \mathcal{G}_s(\mathit{SN}) = (\emptyset, \langle\rangle, N_B)) \\ \mathcal{DG}' = \{N' \text{ with } (\Sigma^N \cup \Sigma^{N_B}, \emptyset);\;\, N \xRightarrow{\Sigma^N \hookrightarrow \Sigma^N \cup \Sigma^{N_B}} N';\;\, N_B \xRightarrow{\Sigma^{N_B} \hookrightarrow \Sigma^N \cup \Sigma^{N_B}} N'\} \end{array}}{N, \Gamma_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{spec\text{-}inst}\; \mathit{SN} \ggg N', (\mathcal{DG}', \mathit{Th})}$$

$$\dfrac{\begin{array}{c}
\mathcal{G}_s(SN) = GS_s = (N_I, \langle N_1, \ldots, N_n \rangle, N_B) \qquad n \geq 1 \\
N_I, N_1, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \texttt{FIT-ARG}_1 \ggg \sigma_1, N_1^A, (\mathcal{DG}_1, \mathit{Th}_1) \\
\ldots \\
N_I, N_n, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \texttt{FIT-ARG}_n \ggg \sigma_n, N_n^A, (\mathcal{DG}_n, \mathit{Th}_n) \\
(\mathit{\Sigma}', \sigma_f) = \mathit{strip}(GS_s)((\mathit{\Sigma}^{N_1^A}, \sigma_1), \ldots, (\mathit{\Sigma}^{N_n^A}, \sigma_n)) \text{ is defined[8]} \\
\mathcal{DG}_1, \ldots, \mathcal{DG}_n \text{ disjointly extend } \mathcal{DG} \\
\sigma = \iota_{\mathit{\Sigma}' \subseteq \mathit{\Sigma}^N \cup \mathit{\Sigma}'} \circ \sigma_f \\
\sigma_i' = \iota_{\mathit{\Sigma}' \subseteq \mathit{\Sigma}^N \cup \mathit{\Sigma}'} \circ \sigma_i \ (i = 1, \ldots, n) \\
\mathcal{DG}' = \bigcup_{i=1,\ldots,n} \mathcal{DG}_i \uplus \{N' \text{ with } (\mathit{\Sigma}^N \cup \mathit{\Sigma}', \emptyset)\} \\
\qquad\qquad \uplus \{ N \xRightarrow{\mathit{\Sigma}^N \hookrightarrow \mathit{\Sigma}^N \cup \mathit{\Sigma}'} N';\ N_B \xRightarrow{\sigma} N' \} \\
\qquad\qquad \uplus \{ N_i^A \xRightarrow{\sigma_i'} N' \mid i = 1, \ldots, n \}
\end{array}}{\begin{array}{l}
N, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \texttt{spec-inst}\ SN\ \texttt{FIT-ARG}_1 \ldots \texttt{FIT-ARG}_n \ggg \\
\qquad\qquad\qquad\qquad N', (\mathcal{DG}', \bigcup_{i=1,\ldots,n} \mathit{Th}_i)
\end{array}}$$

Fitting Arguments

$$\boxed{N_I, N_P, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \texttt{FIT-ARG} \ggg \sigma, N_A, (\mathcal{DG}', \mathit{Th}')}$$

$\mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th})$ is a verification global environment and N_I and N_P are nodes in $\mathcal{DG}$ such that $\mathit{\Sigma}^{N_P}$ is an extension of $\mathit{\Sigma}^{N_I}$; then $(\mathcal{DG}', \mathit{Th}')$ is a development graph extending $(\mathcal{DG}, \mathit{Th})$, N_A is a node in $\mathcal{DG}'$ such that $\mathit{\Sigma}^{N_A}$ is an extension of $\mathit{\Sigma}^{N_I}$, and σ is a signature morphism from $\mathit{\Sigma}^{N_P}$ to $\mathit{\Sigma}^{N_A}$ which is the identity on $\mathit{\Sigma}^{N_I}$.

$$\dfrac{\begin{array}{c}
N_I, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \texttt{SPEC} \ggg N_A, (\mathcal{DG}', \mathit{Th}') \\
\vdash \texttt{SYMB-MAP-ITEMS*} \rhd r \\
\sigma = r|^{\mathit{\Sigma}^{N_P}}_{\mathit{\Sigma}^{N_A}} \qquad \sigma \text{ is the identity on } \mathit{\Sigma}^{N_I}
\end{array}}{\begin{array}{l}
N_I, N_P, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \texttt{fit-spec}\ \texttt{SPEC}\ \texttt{SYMB-MAP-ITEMS*} \ggg \\
\qquad\qquad\qquad\qquad \sigma, N_A, (\mathcal{DG}', \mathit{Th}' \cup \{ N_P \overset{\sigma}{=\!=\!\Rightarrow} N_A \})
\end{array}}$$

4.7.4 Views

View Definitions

$$\boxed{\mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \texttt{VIEW-DEFN} \ggg \mathit{\Gamma}_s', (\mathcal{DG}', \mathit{Th}')}$$

$\mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th})$ is a verification global environment; then $\mathit{\Gamma}_s', (\mathcal{DG}', \mathit{Th}')$ is a verification global environment extending $\mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th})$.

[8] See Sect. III:4.1.5 for an explanation of the notation $GS_s(\ldots)$.

$$\dfrac{\begin{array}{c}\mathit{\Gamma}_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)\\ \mathit{VN} \notin \mathrm{dom}\,\mathcal{V}_s \cup \mathrm{dom}\,\mathcal{G}_s \cup \mathrm{dom}\,\mathcal{A}_s \cup \mathrm{dom}\,\mathcal{T}_s\\ \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{GENERICITY} \rhd\!\!\rhd\!\!\rhd (N_I, \langle N_1, \ldots, N_n\rangle, N_P), (\mathcal{DG}_P, \mathit{Th}_P)\\ N_P, \mathit{\Gamma}_s, (\mathcal{DG}_P, \mathit{Th}_P) \vdash \mathtt{VIEW\text{-}TYPE} \rhd\!\!\rhd\!\!\rhd (N_s, N_t), (\mathcal{DG}', \mathit{Th}')\\ \vdash \mathtt{SYMB\text{-}MAP\text{-}ITEMS*} \rhd r \qquad \sigma = r|^{\Sigma^{N_s}}_{\Sigma^{N_t}}\\ \mathcal{V}_s' = \mathcal{V}_s \cup \{\, \mathit{VN} \mapsto (N_s, \sigma, (N_I, \langle N_1, \ldots, N_n\rangle, N_t))\}\end{array}}{\begin{array}{r}\mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{view\text{-}defn}\ \mathit{VN}\ \mathtt{GENERICITY}\ \mathtt{VIEW\text{-}TYPE}\ \mathtt{SYMB\text{-}MAP\text{-}ITEMS*} \rhd\!\!\rhd\!\!\rhd\\ (\mathcal{G}_s, \mathcal{V}_s', \mathcal{A}_s, \mathcal{T}_s), (\mathcal{DG}', \mathit{Th}' \cup \{\, N_s \overset{\sigma}{=\!=\!\Rightarrow} N_t\})\end{array}}$$

View Types

$$\boxed{N, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{VIEW\text{-}TYPE} \rhd\!\!\rhd\!\!\rhd (N_s, N_t), (\mathcal{DG}', \mathit{Th}')}$$

$\mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th})$ is a verification global environment and N is a node in $\mathcal{DG}$; then $(\mathcal{DG}', \mathit{Th}')$ is a development graph extending $(\mathcal{DG}, \mathit{Th})$, N_s and N_t are nodes in $\mathcal{DG}'$, and the signature Σ^{N_t} is an extension of Σ^N.

$$\dfrac{\begin{array}{c}\emptyset, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{SPEC}_1 \rhd\!\!\rhd\!\!\rhd N_s, (\mathcal{DG}', \mathit{Th}')\\ N, \mathit{\Gamma}_s, (\mathcal{DG}', \mathit{Th}') \vdash \mathtt{SPEC}_2 \rhd\!\!\rhd\!\!\rhd N_t, (\mathcal{DG}'', \mathit{Th}'')\end{array}}{N, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{view\text{-}type}\ \mathtt{SPEC}_1\ \mathtt{SPEC}_2 \rhd\!\!\rhd\!\!\rhd (N_s, N_t), (\mathcal{DG}'', \mathit{Th}'')}$$

Fitting Views

We repeat the rule format for fitting arguments:

$$\boxed{N_I, N_P, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{FIT\text{-}ARG} \rhd\!\!\rhd\!\!\rhd \sigma, N_A, (\mathcal{DG}', \mathit{Th}')}$$

$\mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th})$ is a verification global environment and N_I and N_P are nodes in $\mathcal{DG}$ such that Σ^{N_P} is an extension of Σ^{N_I}; then $(\mathcal{DG}', \mathit{Th}')$ is a development graph extending $(\mathcal{DG}, \mathit{Th})$, N_A is a node in $\mathcal{DG}'$ such that Σ^{N_A} is an extension of Σ^{N_I}, and σ is a signature morphism from Σ^{N_P} to Σ^{N_A} which is the identity on Σ^{N_I}.

$$\dfrac{\begin{array}{c}\mathit{\Gamma}_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \quad \mathcal{V}_s(\mathit{VN}) = (N_s, \sigma, (\emptyset, \langle\rangle, N_t)))\\ \Sigma^{N_s} \cup \Sigma^{N_I} = \Sigma^{N_P}\\ \begin{array}{rl}\mathcal{DG}' = \mathcal{DG} & \uplus\{N_A \text{ with } (\Sigma^{N_I}\} \cup \Sigma^{N_t}, \emptyset)\}\\ & \uplus\{\, N_I \xRightarrow{\Sigma^{N_I} \hookrightarrow \Sigma^{N_I} \cup \Sigma^{N_t}} N_A;\ \ N_t \xRightarrow{\Sigma^{N_t} \hookrightarrow \Sigma^{N_I} \cup \Sigma^{N_t}} N_A\}\\ & \uplus\{N' \text{ with } (\Sigma^{N_P}, \emptyset)\}\\ & \uplus\{\, N_I \xRightarrow{\Sigma^{N_I} \hookrightarrow \Sigma^{N_P}} N';\ \ N_s \xRightarrow{\Sigma^{N_s} \hookrightarrow \Sigma^{N_P}} N'\}\end{array}\end{array}}{\begin{array}{l}N_I, N_P, \mathit{\Gamma}_s, (\mathcal{DG}, \mathit{Th}) \vdash \mathtt{fit\text{-}view}\ \mathit{VN} \rhd\!\!\rhd\!\!\rhd\\ \qquad \sigma \cup \mathit{id}_{\Sigma_I}, N_A, (\mathcal{DG}', \bigcup_{i=1,\ldots,n} \mathit{Th}_i \cup \{\, N_P \overset{\mathit{id}}{=\!=\!\Rightarrow} N'\})\end{array}}$$

$$\begin{tikzcd}
N_s \arrow[rr, dashed, "\sigma"] \arrow[d, hook] & & N_t \arrow[d, hook] \\
N' & N_I \arrow[l, hook'] \arrow[r, hook] & N_A \\
N_P \arrow[u, dashed, "id"]
\end{tikzcd}$$

Fig. 4.4. Non-generic fitting views: relation between sigantures of nodes

Note that the import N_I is already included in N_P.

$$\begin{array}{c}
\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s) \qquad \mathcal{V}_s(VN) = (N_s, \sigma, GS_s)) \\
\Sigma^{N_s} \cup \Sigma^{N_I} = \Sigma N_P \\
GS_s = (N_I', \langle N_1, \ldots, N_n \rangle, N_B) \qquad n \geq 1 \\
N_I', N_1, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{FIT-ARG}_1 \ggg \sigma_1, N_1^A, (\mathcal{DG}_1, Th_1) \\
\ldots \\
N_I', N_n, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{FIT-ARG}_n \ggg \sigma_n, N_n^A, (\mathcal{DG}_n, Th_n) \\
(\Sigma_A, \sigma_f') = strip(GS_s)((\Sigma^{N_1^A}, \sigma_1), \ldots, (\Sigma^{N_n^A}, \sigma_n)) \text{ is defined} \\
\sigma' = \iota_{\Sigma_A \subseteq \Sigma^{N_I} \cup \Sigma_A} \circ \sigma_f' \\
\mathcal{DG}_1, \ldots, \mathcal{DG}_n \text{ disjointly extend } \mathcal{DG} \\
\begin{array}{ll}
\mathcal{DG}' = \mathcal{DG} & \uplus \bigcup_{i=1,\ldots,n} \mathcal{DG}_i \\
& \uplus \{N_A \text{ with } (\Sigma^{N_I} \cup \Sigma_A, \emptyset)\} \\
& \uplus \{ N \xRightarrow{\Sigma^{N_I} \hookrightarrow \Sigma^{N_I} \cup \Sigma_A} N_A;\ N_B \xRightarrow{\sigma'} N_A\} \\
& \uplus \{ N_i^A \xRightarrow{\Sigma^{N_i^A} \hookrightarrow \Sigma^{N_I} \cup \Sigma_A} N_A \mid i = 1, \ldots, n\} \\
& \uplus \{N' \text{ with } (\Sigma^{N_P}, \emptyset)\} \\
& \uplus \{ N_I \xRightarrow{\Sigma^{N_I} \hookrightarrow \Sigma^{N_P}} N';\ N_s \xRightarrow{\Sigma^{N_s} \hookrightarrow \Sigma^{N_P}} N'\}
\end{array} \\
\hline
N_I, N_P, \Gamma_s, (\mathcal{DG}, Th) \vdash \texttt{fit-view}\ VN\ \texttt{FIT-ARG}_1 \ldots \texttt{FIT-ARG}_n \ggg \\
(\sigma_f' \circ \sigma) \cup id_{\Sigma^{N_I}}, N_A, (\mathcal{DG}', \bigcup_{i=1,\ldots,n} Th_i \cup \{ N_P \overset{id}{=\!=\!\Rightarrow} N'\})
\end{array}$$

4.7.5 Adequacy of the Translation

In order to be useful, the translation of Casl specifications to development graphs has to satisfy appropriate adequacy theorems.

Theorem 4.21. *Let a static global environment Γ_s and a verification global environment $\Gamma_s', (\mathcal{DG}, Th)$ with $strip(\Gamma_s', (\mathcal{DG}, Th)) = \Gamma_s$ and a node N in $\mathcal{DG}$ be given. If*

$$\Sigma^N, \Gamma_s \vdash \texttt{SPEC} \rhd \Sigma',$$

$$\begin{tikzcd}
 & N'_I \arrow[d, hook] & \\
 & N_i \arrow[r, dashed, "\sigma_i"] \arrow[d, hook] & N_i^A \arrow[dd, hook] \\
N_s \arrow[r, dashed, "\sigma"] \arrow[d, hook] & N_B \arrow[dr, "\sigma'"] & \\
N' & N_I \arrow[l, hook] \arrow[r, hook] & N_A \\
N_P \arrow[u, dashed, "id"] & &
\end{tikzcd}$$

Fig. 4.5. Generic fitting views: relation between sigantures of nodes

then

$$N, \Gamma'_s, (\mathcal{DG}, Th) \vdash \texttt{SPEC} \ggg N', (\mathcal{DG}', Th')$$

for some $(\mathcal{DG}', Th')$*, with* $\Sigma^{N'} = \Sigma'$ *in* $\mathcal{DG}'$*. Vice versa, if*

$$N, \Gamma'_s, (\mathcal{DG}, Th) \vdash \texttt{SPEC} \ggg N', (\mathcal{DG}', Th')$$

then

$$\Sigma^N, \Gamma_s \vdash \texttt{SPEC} \rhd \Sigma^{N'},$$

Moreover, in either of these equivalent cases, given a model global environment Γ_m *compatible with* $\Gamma'_s, (\mathcal{DG}, Th)$ *and furthermore given a class of* Σ^N*-models* $\mathcal{M}$*, the following are equivalent:*

1. *there is a model class* $\mathcal{M}'$ *with* $\Sigma, \mathcal{M}, \Gamma_s, \Gamma_m \vdash \texttt{SPEC} \Rightarrow \mathcal{M}'$*.*
2. $\mathcal{DG}' \models Th''$*,*

where $Th'' \subseteq Th'$ *contains those theorem links not generated by semantic annotations*[9]*. Furthermore, if these two equivalent conditions hold, then*

$$\mathcal{M}' = \{M \in \mathbf{Mod}_{\mathcal{DG}'}(N') \mid M|_\Sigma \in \mathcal{M}\}.$$

The proof of this theorem follows an induction over the rules for the verification semantics. The theorem states that the verification semantics, when the development graph information is stripped off, does the same as the static

[9] In the sequel, we will tacitly assume that the theorem links corresponding to semantic annotations are removed from the set of theorem links - only then, the model-theoretic semantics is captured. On the other hand, usually one will keep these theorem links as proof obligations, since once they are proven, they provide further trust in the specification.

semantics. Moreover, the extra development graph information (most importantly, the theorem links *Th* that capture the proof obligations) constructed by the verification semantics is sufficient to capture the model semantics.

Note that the verification global environment $\Gamma'_s, (\mathcal{DG}, Th)$ in the above theorem usually will be obtained from the verification semantics of libraries, see Chap. 6.

We now reformulate consequences of and refinements between CASL structured specifications in terms of development graphs.

Definition 4.22. *Consider a static global environment $\Gamma_s = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)$, compatible with some model global environment Γ_m. Let $\Gamma_s^{triv}, (\mathcal{DG}^{triv}, \emptyset)$ be the trivial verification global environment with $strip(\Gamma_s^{triv}) = \Gamma_s$, that is, in $\mathcal{DG}^{triv}$, for each ocurrence of signature being part of a generic signature in $Dom(\mathcal{G}_s)$ or of a view signature in $Dom(\mathcal{V}_s)$, a new node with that signature and no local axioms is introduced, $\Gamma_s^{triv} = (\mathcal{G}'_s, \mathcal{V}'_s, \emptyset, \emptyset)$ is such that $\mathcal{G}'_s$ maps each $SN \in Dom(\mathcal{G}_s)$ to the verification signature obtained from the generic static signature $\mathcal{G}_s(SN)$ by replacing the signatures with the corresponding nodes, and $\mathcal{V}'_s$ is obtained similarly. Obviously, $\Gamma_s^{triv}, (\mathcal{DG}^{triv}, \emptyset)$ is compatible with Γ_m as well.*

Consider now CASL structured specifications SPEC*,* SPEC$_1$ *and* SPEC$_2$*. If*

$$\emptyset, \Gamma_s \vdash \mathtt{SPEC} \rhd \Sigma,$$

and Ψ is a set of Σ-sentences, then

$$\Gamma_s, \Gamma_m : \mathtt{SPEC} \models \Psi$$

is defined to mean that $\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC} \Rightarrow \mathcal{M}$, and $M \models \Psi$ for each $M \in \mathcal{M}$. By Theorem 4.21, we have

$$\emptyset, \Gamma_s^{triv}, (\mathcal{DG}^{triv}, \emptyset) \vdash \mathtt{SPEC} \ggg N, (\mathcal{DG}', Th')$$

for some N and $(\mathcal{DG}', Th')$. We write

$$\Gamma_s : \mathtt{SPEC} \vdash \Psi$$

for

$$\mathcal{DG}' \vdash N \Rightarrow \Psi.$$

Furthermore, if

$$\emptyset, \Gamma_s \vdash \mathtt{SPEC}_1 \rhd \Sigma \quad \text{and} \quad \emptyset, \Gamma_s \vdash \mathtt{SPEC}_2 \rhd \Sigma,$$

then

$$\Gamma_s, \Gamma_m : \mathtt{SPEC}_1 \rightsquigarrow \mathtt{SPEC}_2$$

is defined to mean that $\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash \mathtt{SPEC}_1 \Rightarrow \mathcal{M}_1$ and $\emptyset, \mathcal{M}_\perp, \Gamma_s, \Gamma_m \vdash$ SPEC$_2$ $\Rightarrow \mathcal{M}_2$*, for some model classes $\mathcal{M}_1$ and $\mathcal{M}_2$ such that $\mathcal{M}_2 \subseteq \mathcal{M}_1$. By Theorem 4.21,*

$$\emptyset, \Gamma_s^{triv}, (\mathcal{DG}^{triv}, \emptyset) \vdash \mathtt{SPEC}_1 \ggg N_1, (\mathcal{DG}', Th'), \text{ and}$$

$$\emptyset, \Gamma_s^{triv}, (\mathcal{DG}', Th') \vdash \mathtt{SPEC}_2 \ggg N_2, (\mathcal{DG}'', Th'')$$

for some $(\mathcal{DG}', Th')$, $(\mathcal{DG}'', Th'')$, N_1 *and* N_2 *with* $\Sigma^{N_1} = \Sigma^{N_2} = \Sigma$, *and we write*

$$\Gamma_s : \mathtt{SPEC}_1 \rightsquigarrow \mathtt{SPEC}_2$$

for

$$\mathcal{DG}'' \vdash N_1 \overset{id}{==\Rightarrow} N_2.$$

Using Theorem 4.17, we now obtain

Proposition 4.23. *Inference for structured specifications is sound as well, that is, given a static global environment* Γ_s,

$$\Gamma_s : \mathtt{SPEC} \vdash \Psi$$

implies

for each Γ_m *compatible with* Γ_s, *it holds that* $\Gamma_s, \Gamma_m : \mathtt{SPEC} \models \Psi$;

and

$$\Gamma_s : \mathtt{SPEC}_1 \rightsquigarrow \mathtt{SPEC}_2$$

implies

for each Γ_m *compatible with* Γ_s, *it holds that* $\Gamma_s, \Gamma_m : \mathtt{SPEC}_1 \approx\!\!> \mathtt{SPEC}_2$.

□

With Theorem 4.19, we obtain

Proposition 4.24. *Assume that the underlying logic is complete. Then inference for structured specifications is complete relative to complete oracles for conservative extensions and free theorem links and a complete elimination rule for free definition links, that is, given a static global environment* Γ_s,

for each Γ_m *compatible with* Γ_s, *it holds that* $\Gamma_s, \Gamma_m : \mathtt{SPEC} \models \Psi$

implies

$$\Gamma_s : \mathtt{SPEC} \vdash \Psi;$$

and

for each Γ_m *compatible with* Γ_s, *it holds that* $\Gamma_s, \Gamma_m : \mathtt{SPEC}_1 \approx\!\!> \mathtt{SPEC}_2$

implies

$$\Gamma_s : \mathtt{SPEC}_1 \rightsquigarrow \mathtt{SPEC}_2.$$

□

Remark 4.25. The proof-theoretic notions of consequence and refinement for CASL structured specifications have been introduced above with reference to the trivial verification global environment. This is quite restrictive (since it disallows the use of the properties of those specifications and views used which have been defined externally) but can be generalised to the case of an arbitrary verification global environment used to describe the specification and view names that the specifications in question use. Props. 4.23 and 4.24 hold then for global model environments that are compatible with verification global environments made explicit in such a way.

Remark 4.26. In Sect. 5.3.1 of the architectural proof calculus, there appear conditions of the form

$$\eta(R(\texttt{SPEC})) \leadsto^{J}_{\Sigma} \bigcup_{i=1\ldots n} \eta_i(\overline{R}(\texttt{SPEC}_i))$$

where η_i, η are signature morphisms with common target signature Σ, R is the action of the global comorphism given by the general assumption made in Remark 4.3 on structured specifications, and $\overline{R}$ adds the sentences resulting from signature translations (see Sect. 5.3.1 for formal definitions of the latter two)[10].

In the context above, this can be interpreted as follows. We take the needed runs of the verification semantics:

$$\emptyset, \Gamma_s^{triv}, (\mathcal{DG}^{triv}, \emptyset) \vdash \texttt{SPEC} \ggg O, (\mathcal{DG}_0, Th_0), \text{ and}$$

$$\emptyset, \Gamma_s^{triv}, (\mathcal{DG}_{i-1}, Th_{i-1}) \vdash \texttt{SPEC}_i \ggg O_i, (\mathcal{DG}_i, Th_i).$$

Well-formedness of the involved specifications is captured by

$$\mathcal{DG}_n \vdash Th_n.$$

In order to interpret the proof obligation, take the translation $R(\mathcal{DG}_n)$ of $\mathcal{DG}_n$ along R. Now take

$$\begin{aligned}\mathcal{DG}' = R(\mathcal{DG}_n) \uplus \{&N_1 \text{ with } (\Sigma, \emptyset);\ N_2 \text{ with } (\Sigma, \emptyset);\ R(O) \xRightarrow{\eta} N_1\} \\ \cup\{& R(O_i) \xRightarrow{\eta_i} N_2 \mid i = 1 \ldots n\}\end{aligned}$$

Then

$$\eta(R(\texttt{SPEC})) \leadsto^{J}_{\Sigma} \bigcup_{i=1\ldots n} \eta_i(\overline{R}(\texttt{SPEC}_i))$$

just means that

$$\mathcal{DG}' \vdash N_1 \overset{id}{=\!=\!\Rightarrow} N_2.$$

[10] In Chap. 5, finite *sets* of specifications and their translations are considered. However, these can be replaced with their CASL unions.

5

Architectural Specification Calculus

In this chapter we consider the problem of proving that a given architectural specification has a denotation (i.e., is correct) and that the units produced using it satisfy a given unit specification. In order to make the presentation readable we deal with a slightly restricted language of architectural specifications (Fig. 5.1). For the same reason we prefer to use a concrete syntax instead of the abstract one (the **reduction** construct represents both **hide** and **reveal**), and to assume that signature morphisms are explicit elements of the syntax, which frees us from introducing the notions of symbol and symbol map.

$$
\begin{array}{l}
\mathit{ASP} ::= \textbf{units}\ \mathit{UD}_1 \ldots \mathit{UD}_n\ \textbf{result}\ \mathit{UE} \\
\mathit{UD} ::= A : \mathit{USP} \\
\mathit{USP} ::= \mathit{SP} \mid \\
\qquad\qquad \mathit{SP} \rightarrow \mathit{SP} \\
\mathit{UE} ::= \mathit{UT} \mid \\
\qquad\qquad \lambda\ A : \mathit{SP} \bullet \mathit{UT} \\
\mathit{UT} ::= A \mid \\
\qquad\quad A\ [\ \mathit{UT}\ \textbf{fit}\ \sigma : \Sigma \rightarrow \Sigma'\ ] \mid \\
\qquad\quad \mathit{UT}\ \textbf{and}\ \mathit{UT} \mid \\
\qquad\quad \mathit{UT}\ \textbf{with}\ \sigma : \Sigma \rightarrow \Sigma' \mid \\
\qquad\quad \mathit{UT}\ \textbf{reduction}\ \sigma : \Sigma \rightarrow \Sigma' \mid \\
\qquad\quad \textbf{local}\ A = \mathit{UT}\ \textbf{within}\ \mathit{UT}
\end{array}
$$

Fig. 5.1. Restricted language of architectural specifications.

Thus, we do not take the following features of Casl architectural specifications into account:

- imports (the **given** construct);
- multiparameter units;
- complex forms of unit specifications (using local or global environments);
- local definitions of generic units;
- definitions mixed with declarations.

Of the above only imports add genuine complexity to the proof calculus.

5.1 Semantics

In this section we will give the static, model, and extended static semantics for the restricted language of architectural specifications presented above; we also introduce many concepts that we then expand in Sect. 5.3, which presents the proof calculus. This section may serve not only as a necessary preliminary for the subsequent sections, but also as a less formal introduction to Chap. III:5, with which it is fully consistent. We take concepts such as signature inclusion, as defined in Chap. III:4, for granted. We also assume that the signature category of the underlying institution has *selected pushouts*; for CASL, they have been defined in Sect. III:4.1.3.

5.1.1 Static and Model Semantics

By a *unit signature* $U\Sigma$ we understand either a (regular) signature Σ or a *generic unit signature* $\Sigma \to \Sigma'$, where Σ is a subsignature of Σ'. By $\mathbf{Unit}(U\Sigma)$ we denote the class of all units over $U\Sigma$, i.e., $\mathbf{Unit}(\Sigma) = \mathbf{Mod}(\Sigma)$ and $\mathbf{Unit}(\Sigma \to \Sigma')$ is the set of all partial functions $F : \mathbf{Mod}(\Sigma) \to \mathbf{Mod}(\Sigma')$ which are *persistent*, that is, $F(M)|_{\Sigma} = M$ for all $M \in \mathrm{dom}(F)$.

In this chapter unit names will be denoted by the letters U, A, B, and so on. A *static context* C_s assigns unit signatures to a finite number of unit names. An *environment* E *fitting* C_s assigns an element of $\mathbf{Unit}(C_s(U))$ to any unit name $U \in \mathrm{dom}(C_s)$. A *context* C *fitting* C_s is any class of environments fitting C_s. A *model evaluator* MEv *from* C *to* Σ is simply a function from C to $\mathbf{Mod}(\Sigma)$. Similarly, a *unit evaluator* UEv *from* C *to* $U\Sigma$ is a function from C to $\mathbf{Unit}(U\Sigma)$. In the semantics, by $\mathcal{M}$ we will denote classes of models over a common signature, and by $\mathcal{U}$ classes of units over a common unit signature.

We assume that for any specification SP the semantics defines a signature $sig[SP]$ and a class of models $[\![SP]\!] \subseteq \mathbf{Mod}(sig[SP])$. Thus we disregard any (purely technical) problems arising from incorrect specifications or from local environments. We say that $SP \to SP'$ is a *generic unit specification* if $sig[SP]$ is a subsignature of $sig[SP']$; such a specification defines the class of units $[\![SP \to SP']\!]$, containing all units $F \in \mathbf{Unit}(sig[SP] \to sig[SP'])$ such that $\mathrm{dom}(F) = [\![SP]\!]$ and for all $M \in \mathrm{dom}(F)$ we have $F(M) \in [\![SP']\!]$. A specification SP is *inconsistent* if $[\![SP]\!] = \emptyset$; a generic unit specification $SP \to SP'$ is *inconsistent* if $[\![SP \to SP']\!] = \emptyset$.

The static (using $\rhd$) and model (using $\Rightarrow$) semantics may be presented by the following rules. It is assumed that the model semantics will be run only after a successful run of the static semantics.

$$\boxed{\vdash \mathit{ASP} \rhd C_s, U\Sigma \qquad\qquad \vdash \mathit{ASP} \Rightarrow \mathit{UEv}}$$

The context dom(UEv) fits C_s and UEv is a unit evaluator into $U\Sigma$.

$$\dfrac{\begin{array}{c} \vdash \mathit{UD}_1 \rhd C_s^1 \\ \vdots \\ \vdash \mathit{UD}_n \rhd C_s^n \\ \mathrm{dom}(C_s^i) \cap \mathrm{dom}(C_s^j) = \emptyset \text{ for all } 1 \leq i < j \leq n \\ C_s = C_s^1 \cup \cdots \cup C_s^n \\ C_s \vdash \mathit{UE} \rhd U\Sigma \end{array}}{\vdash \textbf{units } \mathit{UD}_1 \ldots \mathit{UD}_n \textbf{ result } \mathit{UE} \rhd C_s, U\Sigma}$$

$$\dfrac{\begin{array}{c} \begin{array}{cc} \vdash \mathit{UD}_1 \rhd C_s^1 & \vdash \mathit{UD}_1 \Rightarrow C^1 \\ \vdots & \vdots \\ \vdash \mathit{UD}_n \rhd C_s^n & \vdash \mathit{UD}_n \Rightarrow C^n \end{array} \\ C_s = C_s^1 \cup \cdots \cup C_s^n \\ C = \{ E_1 \cup \cdots \cup E_n \mid E_1 \in C^1, \ldots, E_n \in C^n \} \\ C_s, C \vdash \mathit{UE} \Rightarrow \mathit{UEv} \end{array}}{\vdash \textbf{units } \mathit{UD}_1 \ldots \mathit{UD}_n \textbf{ result } \mathit{UE} \Rightarrow \mathit{UEv}}$$

$$\boxed{\vdash \mathit{UD} \rhd C_s \qquad\qquad \vdash \mathit{UD} \Rightarrow C}$$

The context C fits C_s.

$$\dfrac{}{\vdash A : \mathit{SP} \rhd \{ A \mapsto sig[\mathit{SP}] \}}$$

$$\dfrac{}{\vdash A : \mathit{SP} \Rightarrow \{ \{A \mapsto M\} \mid M \in \llbracket \mathit{SP} \rrbracket \}}$$

$$\dfrac{sig[\mathit{SP}] \text{ is a subsignature of } sig[\mathit{SP}']}{\vdash A : \mathit{SP} \to \mathit{SP}' \rhd \{ A \mapsto sig[\mathit{SP}] \to sig[\mathit{SP}'] \}}$$

$$\dfrac{}{\vdash A : \mathit{SP} \to \mathit{SP}' \Rightarrow \{ \{A \mapsto F\} \mid F \in \llbracket \mathit{SP} \to \mathit{SP}' \rrbracket \}}$$

$$\boxed{C_s \vdash UE \rhd U\Sigma \qquad\qquad C_s, C \vdash UE \Rightarrow UEv}$$

The context C should fit C_s. Then UEv is a unit evaluator from C to $U\Sigma$.

$$\frac{C_s \vdash UT \rhd \Sigma}{C_s \vdash UT \,\mathrm{qua}\, UE \rhd \Sigma}$$

$$\frac{C_s, C \vdash UT \Rightarrow MEv}{C_s, C \vdash UT \,\mathrm{qua}\, UE \Rightarrow MEv}$$

$$\frac{\begin{array}{c} A \notin \mathrm{dom}(C_s) \\ C_s \cup \{ A \mapsto sig[SP] \} \vdash UT \rhd \Sigma \\ sig[SP] \text{ is a subsignature of } \Sigma \end{array}}{C_s \vdash \lambda A : SP \bullet UT \rhd sig[SP] \to \Sigma}$$

$$\frac{\begin{array}{c} C_s \cup \{A \mapsto sig[SP]\} \vdash UT \rhd \Sigma \\ C_s \cup \{A \mapsto sig[SP]\}, \{ E \cup \{A \mapsto M\} \mid E \in C, M \in [\![SP]\!] \} \vdash UT \Rightarrow MEv \\ \text{for all } E \in C \text{ and } M \in [\![SP]\!] \text{ we have } MEv(E \cup \{A \mapsto M\})|_{sig[SP]} = M \end{array}}{C_s, C \vdash \lambda A : SP \bullet UT \Rightarrow \lambda E \in C \cdot \lambda M \in [\![SP]\!] \cdot MEv(E \cup \{A \mapsto M\})}$$

$$\boxed{C_s \vdash UT \rhd \Sigma \qquad\qquad C_s, C \vdash UT \Rightarrow MEv}$$

The context C should fit C_s. Then MEv is a model evaluator from C to Σ.

$$\frac{C_s(A) = \Sigma}{C_s \vdash A \rhd \Sigma}$$

$$\frac{}{C_s, C \vdash A \Rightarrow \lambda E \in C \cdot E(A)}$$

$$\frac{\begin{array}{c} C_s(A) = \Sigma_f \to \Sigma_r \\ C_s \vdash UT \rhd \Sigma_a \\ \tau : \Sigma_a \to \Delta \text{ and } \sigma' : \Sigma_r \to \Delta \text{ form the selected pushout for } (\sigma, \iota_{\Sigma_f \subseteq \Sigma_r}) \end{array}}{C_s \vdash A \, [\, UT \ \mathbf{fit}\ \sigma : \Sigma_f \to \Sigma_a \,] \rhd \Delta}$$

$$\frac{\begin{array}{c} C_s, C \vdash \mathit{UT} \Rightarrow \mathit{MEv} \\ \text{for all } E \in C,\ \mathit{MEv}(E)|_\sigma \in \mathrm{dom}(E(A)) \\ C_s(A) = \Sigma_f \to \Sigma_r \\ \tau : \Sigma_a \to \Delta \text{ and } \sigma' : \Sigma_r \to \Delta \text{ form the selected pushout for } (\sigma, \iota_{\Sigma_f \subseteq \Sigma_r}) \\ \text{for all } E \in C \text{ there exists a unique } \Delta\text{-model } M_E \text{ such} \\ \text{that } M_E|_\tau = \mathit{MEv}(E) \text{ and } M_E|_{\sigma'} = E(A)(\mathit{MEv}(E)|_\sigma) \end{array}}{C_s, C \vdash A\ [\ \mathit{UT}\ \mathbf{fit}\ \sigma : \Sigma_f \to \Sigma_a\ ] \Rightarrow \lambda E \in C \cdot M_E}$$

$$\frac{\begin{array}{c} C_s \vdash \mathit{UT}_1 \rhd \Sigma_1 \\ C_s \vdash \mathit{UT}_2 \rhd \Sigma_2 \end{array}}{C_s \vdash \mathit{UT}_1\ \mathbf{and}\ \mathit{UT}_2 \rhd \Sigma_1 \cup \Sigma_2}$$

$$\frac{\begin{array}{c} C_s \vdash \mathit{UT}_1 \rhd \Sigma_1 \\ C_s \vdash \mathit{UT}_2 \rhd \Sigma_2 \\ C_s, C \vdash \mathit{UT}_1 \Rightarrow \mathit{MEv}_1 \\ C_s, C \vdash \mathit{UT}_2 \Rightarrow \mathit{MEv}_2 \\ \Sigma = \Sigma_1 \cup \Sigma_2 \\ \text{for all } E \in C \text{ there exists a unique } \Sigma\text{-model } M_E \text{ such} \\ \text{that } M_E|_{\Sigma_1} = \mathit{MEv}_1(E) \text{ and } M_E|_{\Sigma_2} = \mathit{MEv}_2(E) \end{array}}{C_s, C \vdash \mathit{UT}_1\ \mathbf{and}\ \mathit{UT}_2 \Rightarrow \lambda E \in C \cdot M_E}$$

$$\frac{C_s \vdash \mathit{UT} \rhd \Sigma}{C_s \vdash \mathit{UT}\ \mathbf{with}\ \sigma : \Sigma \to \Sigma' \rhd \Sigma'}$$

$$\frac{\begin{array}{c} C_s, C \vdash \mathit{UT} \Rightarrow \mathit{MEv} \\ \text{for all } E \in C \text{ there exists a unique } \Sigma'\text{-model } M_E \text{ such that } M_E|_\sigma = \mathit{MEv}(E) \end{array}}{C_s, C \vdash \mathit{UT}\ \mathbf{with}\ \sigma : \Sigma \to \Sigma' \Rightarrow \lambda E \in C \cdot M_E}$$

$$\frac{C_s \vdash \mathit{UT} \rhd \Sigma'}{C_s \vdash \mathit{UT}\ \mathbf{reduction}\ \sigma : \Sigma \to \Sigma' \rhd \Sigma}$$

$$\frac{C_s, C \vdash \mathit{UT} \Rightarrow \mathit{MEv}}{C_s, C \vdash \mathit{UT}\ \mathbf{reduction}\ \sigma : \Sigma \to \Sigma' \Rightarrow \lambda E \in C \cdot \mathit{MEv}(E)|_\sigma}$$

$$\frac{\begin{array}{c} A \notin \mathrm{dom}(C_s) \\ C_s \vdash \mathit{UT} \rhd \Sigma \\ C_s \cup \{A \mapsto \Sigma\} \vdash \mathit{UT}' \rhd \Sigma' \end{array}}{C_s \vdash \mathbf{local}\ A = \mathit{UT}\ \mathbf{within}\ \mathit{UT}' \rhd \Sigma'}$$

$$\frac{\begin{array}{c} C_s \vdash \mathit{UT} \rhd \Sigma \\ C_s,\, C \vdash \mathit{UT} \Rightarrow \mathit{MEv} \\ C_s \cup \{A \mapsto \Sigma\},\, \{\, E \cup \{A \mapsto \mathit{MEv}(E)\} \mid E \in C\,\} \vdash \mathit{UT}' \Rightarrow \mathit{MEv}' \end{array}}{C_s,\, C \vdash \textbf{local } A = \mathit{UT} \textbf{ within } \mathit{UT}' \Rightarrow \lambda E \in C \cdot \mathit{MEv}'(E \cup \{A \mapsto \mathit{MEv}(E)\})}$$

5.1.2 Extended Static Semantics

As has been argued in the introduction to Sect. III:5.6, one would expect that the four premises of the model semantics starting with "for all $E \in C\ldots$", found in the rules for **and**, **with**, for generic unit application and for λ-abstraction, will be dismissed not by means of theorem-proving, but rather in a semi-automatic manner. This is motivated by their static nature and by the fact that the similar sharing conditions of languages such as Standard ML are checked automatically.

Toward this end one introduces an extended static semantics[1] (denoted using $\rhd\!\!\!\rhd$). It can be then proven that if *ASP* has a denotation (we use $\Box$ as the dummy denotation) w.r.t. the extended static semantics, then it has a denotation w.r.t. the static semantics and in the model semantics none of the above-mentioned 4 premises need to be checked (see Sect. III:5.6). Using the extended static semantics also makes it easier to develop a proof-calculus. The extended static semantics presented below is essentially equivalent to that of Sect. III:5.6. *contexts* instead of the However, in order to make the semantics easily extendible to a readable proof calculus, in the sequel we will be using *contexts* instead of the diagrams used there. Thanks to that, the proof-calculus presented in Sect. 5.3 will then be a rather straightforward extension of the extended static semantics. The link between 'standard' (i.e., static and model) semantics and the extended static semantics is described in the next section.

A *generic context* Γ_{gen} is a finite set of declarations $A : \Sigma \to \Sigma'$, where $\Sigma \to \Sigma'$ is a generic unit signature.

A *context* Γ is a finite set of declarations of two forms:

- $A : \Sigma$;
- $\sigma : A \to B$, where $A : \Sigma_A$ and $B : \Sigma_B$ in Γ and $\sigma : \Sigma_A \to \Sigma_B$ is a signature morphism.

Any unit name may be declared at most once in a context Γ; the same applies to unit names in generic contexts Γ_{gen}.

By $\mathrm{dom}(\Gamma)$ we denote the set of unit names A such that $A : \Sigma$ in Γ for some signature Σ; we define $\mathrm{dom}(\Gamma_{gen})$ analogously. If Γ_1 and Γ_2 are two contexts and for all $A \in \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2)$ we have $A : \Sigma$ in Γ_1 if and only if $A : \Sigma$ in Γ_2, then their sum $\Gamma_1 \cup \Gamma_2$ is a context as well.We say that a context Γ is a *subcontext* of the context Γ' if the declarations of Γ form a subset of the declarations of Γ'.

[1] Extended static semantics is described in Sect. III:5.6 and, e.g., in [62].

If Γ is a context and A, B and B' are unit names, then $\Gamma[B'/B]$ and $A[B'/B]$ arise from Γ and A in the obvious way by substituting B' for B (that is, $A[B'/B]$ is A if $A \neq B$ and it is B' if $A = B$).

A model family $M = \{M_U\}_{U \in \mathrm{dom}(\Gamma)}$ is *consistent with* Γ if:

- for all $A : \Sigma$ in Γ, we have $M_A \in \mathbf{Mod}(\Sigma)$;
- for all $\sigma : A \to B$ in Γ, we have $M_A = M_B|_\sigma$.

Let Γ be a subcontext of Γ'. We say that Γ *ensures amalgamability for* Γ' if every model family consistent with Γ uniquely extends to a model family consistent with Γ'.

The rules for deriving extended static semantics statements follow. The general idea is that the statement $\mathcal{A}$, $\Gamma_{gen}, \Gamma \vdash \mathit{UT} \rhd\!\!\rhd \Gamma', A$ expresses the following fact: assume that we have a model family consistent with Γ – this model family consists of units declared in the architectural specification's **units** section, of units locally defined in it, and of 'dummy' units used for terms etc.; $\mathcal{A}$ is the set of names of declared or locally defined units. Assume also that we have generic units as described by Γ_{gen}. Then the unit described by UT may be built and, moreover, one may obtain it by extending the given model family in a unique way to a model family consistent with Γ' and then taking the unit labeled A. The symbol '$\Box$' is used as a dummy denotation in cases where we are only interested whether for a given construct the extended static semantics is successful or not.

$$\boxed{\vdash \mathit{ASP} \rhd\!\!\rhd \Box}$$

$$\frac{\begin{array}{c}\vdash \mathit{UD}_1 \rhd\!\!\rhd \Gamma^1_{gen}, \Gamma^1 \\ \vdots \\ \vdash \mathit{UD}_n \rhd\!\!\rhd \Gamma^n_{gen}, \Gamma^n \\ \mathrm{dom}(\Gamma^i_{gen}) \cap \mathrm{dom}(\Gamma^j_{gen}) = \emptyset \text{ for all } 1 \leq i < j \leq n \\ \mathrm{dom}(\Gamma^i) \cap \mathrm{dom}(\Gamma^j) = \emptyset \text{ for all } 1 \leq i < j \leq n \\ \Gamma_{gen} = \Gamma^1_{gen} \cup \cdots \cup \Gamma^n_{gen} \\ \Gamma = \Gamma^1 \cup \cdots \cup \Gamma^n \\ \mathrm{dom}(\Gamma_{gen}) \cap \mathrm{dom}(\Gamma) = \emptyset \\ \Gamma_{gen}, \Gamma \vdash \mathit{UE} \rhd\!\!\rhd \Box\end{array}}{\vdash \textbf{units } \mathit{UD}_1 \ldots \mathit{UD}_n \textbf{ result } \mathit{UE} \rhd\!\!\rhd \Box}$$

$$\boxed{\vdash \mathit{UD} \rhd\!\!\rhd \Gamma_{gen}, \Gamma}$$

$$\frac{}{\vdash A : \mathit{SP} \rhd\!\!\rhd \{A : \mathit{sig}[\mathit{SP}]\}, \emptyset}$$

$$\frac{\mathit{sig}[\mathit{SP}] \text{ is a subsignature of } \mathit{sig}[\mathit{SP}']}{\vdash A : \mathit{SP} \to \mathit{SP}' \rhd\!\!\rhd \emptyset, \{A : \mathit{sig}[\mathit{SP}] \to \mathit{sig}[\mathit{SP}']\}}$$

$$\boxed{\varGamma_{gen},\ \varGamma \vdash \mathit{UE} \rhd\!\!\!\rhd \Box}$$

The sets $\mathrm{dom}(\varGamma_{gen})$ and $\mathrm{dom}(\varGamma)$ should be disjoint.

$$\frac{\mathrm{dom}(\varGamma),\ \varGamma_{gen},\ \varGamma \vdash \mathit{UT} \rhd\!\!\!\rhd \varGamma', A}{\varGamma_{gen},\ \varGamma \vdash \mathit{UT}\ \mathrm{qua}\ \mathit{UE} \rhd\!\!\!\rhd \Box}$$

$$\frac{\begin{array}{c} A \notin \mathrm{dom}(\varGamma) \cup \mathrm{dom}(\varGamma_{gen}) \\ \mathrm{dom}(\varGamma) \cup \{A\},\ \varGamma_{gen},\ \varGamma \cup \{A : \mathit{sig}[\mathit{SP}]\} \vdash \mathit{UT} \rhd\!\!\!\rhd \varGamma', B \\ B : \varSigma \text{ in } \varGamma' \text{ and } \mathit{sig}[\mathit{SP}] \text{ is a subsignature of } \varSigma \\ \varGamma' \text{ ensures amalgamability for } \varGamma' \cup \{\mathrm{id}_{\varSigma} : A \to B\} \end{array}}{\varGamma_{gen},\ \varGamma \vdash \lambda A : \mathit{SP} \bullet \mathit{UT} \rhd\!\!\!\rhd \Box}$$

$$\boxed{\mathcal{A},\ \varGamma_{gen},\ \varGamma \vdash \mathit{UT} \rhd\!\!\!\rhd \varGamma', A}$$

The inclusion $\mathcal{A} \subseteq \mathrm{dom}(\varGamma) \setminus \mathrm{dom}(\varGamma_{gen})$ should hold. Then $\varGamma$ is a subcontext of $\varGamma'$ and $A \in \mathrm{dom}(\varGamma')$.

$$\frac{A \in \mathcal{A}}{\mathcal{A},\ \varGamma_{gen},\ \varGamma \vdash A \rhd\!\!\!\rhd \varGamma, A}$$

$$\frac{\begin{array}{c} A : \varSigma_f \to \varSigma_r \text{ in } \varGamma_{gen} \\ \mathcal{A},\ \varGamma_{gen},\ \varGamma \vdash \mathit{UT} \rhd\!\!\!\rhd \varGamma_a, A_a \\ A_a : \varSigma_a \text{ in } \varGamma_a \\ \tau : \varSigma_a \to \varDelta \text{ and } \sigma' : \varSigma_r \to \varDelta \text{ form the selected pushout for } (\sigma, \iota_{\varSigma_f \subseteq \varSigma_r}) \\ A_f, A_r, B \notin \mathrm{dom}(\varGamma_a) \text{ are distinct} \\ \varGamma' = \varGamma_a \cup \{A_f : \varSigma_f,\ \sigma : A_f \to A_a,\ A_r : \varSigma_r,\ \iota_{\varSigma_f \subseteq \varSigma_r} : A_f \to A_r\} \\ \varGamma'' = \varGamma' \cup \{B : \varDelta,\ \tau : A_a \to B,\ \sigma' : A_r \to B\} \\ \varGamma' \text{ ensures amalgamability for } \varGamma'' \end{array}}{\mathcal{A},\ \varGamma_{gen},\ \varGamma \vdash A\ [\ \mathit{UT}\ \mathbf{fit}\ \sigma : \varSigma_f \to \varSigma_a\ ] \rhd\!\!\!\rhd \varGamma'', B}$$

$$\frac{\begin{array}{c}\mathcal{A},\, \Gamma_{gen},\, \Gamma \vdash \mathit{UT}_1 \mathrel{\rhd\!\!\rhd} \Gamma_1, A_1 \\ \mathcal{A},\, \Gamma_{gen},\, \Gamma \vdash \mathit{UT}_2 \mathrel{\rhd\!\!\rhd} \Gamma_2, A_2 \\ A_1 : \Sigma_1 \text{ in } \Gamma_1 \\ A_2 : \Sigma_2 \text{ in } \Gamma_2 \\ \Sigma = \Sigma_1 \cup \Sigma_2 \\ \mathrm{dom}(\Gamma_1) \cap \mathrm{dom}(\Gamma_2) = \mathrm{dom}(\Gamma) \\ B \notin \mathrm{dom}(\Gamma_1) \cup \mathrm{dom}(\Gamma_2) \\ \Gamma' = \Gamma_1 \cup \Gamma_2 \cup \{B : \Sigma,\ \iota_{\Sigma_1 \subseteq \Sigma} : A_1 \to B,\ \iota_{\Sigma_2 \subseteq \Sigma} : A_2 \to B\} \\ \Gamma_1 \cup \Gamma_2 \text{ ensures amalgamability for } \Gamma'\end{array}}{\mathcal{A},\, \Gamma_{gen},\, \Gamma \vdash \mathit{UT}_1 \textbf{ and } \mathit{UT}_2 \mathrel{\rhd\!\!\rhd} \Gamma', B}$$

$$\frac{\begin{array}{c}\mathcal{A},\, \Gamma_{gen},\, \Gamma \vdash \mathit{UT} \mathrel{\rhd\!\!\rhd} \Gamma', A \\ B \notin \mathrm{dom}(\Gamma') \\ \Gamma'' = \Gamma' \cup \{B : \Sigma',\ \sigma : A \to B\} \\ \Gamma' \text{ ensures amalgamability for } \Gamma''\end{array}}{\mathcal{A},\, \Gamma_{gen},\, \Gamma \vdash \mathit{UT} \textbf{ with } \sigma : \Sigma \to \Sigma' \mathrel{\rhd\!\!\rhd} \Gamma'', B}$$

$$\frac{\begin{array}{c}\mathcal{A},\, \Gamma_{gen},\, \Gamma \vdash \mathit{UT} \rhd \Gamma', A \\ B \notin \mathrm{dom}(\Gamma')\end{array}}{\mathcal{A},\, \Gamma_{gen},\, \Gamma \vdash \mathit{UT} \textbf{ reduction } \sigma : \Sigma \to \Sigma' \mathrel{\rhd\!\!\rhd} \Gamma' \cup \{B : \Sigma,\ \sigma : B \to A\}, B}$$

$$\frac{\begin{array}{c}\mathcal{A},\, \Gamma_{gen},\, \Gamma \vdash \mathit{UT} \mathrel{\rhd\!\!\rhd} \Gamma', B \\ B : \Sigma \text{ in } \Gamma' \\ A \notin \mathrm{dom}(\Gamma') \cup \mathrm{dom}(\Gamma_{gen}) \\ \mathcal{A} \cup \{A\},\, \Gamma_{gen}, \Gamma' \cup \{A : \Sigma,\ \mathrm{id}_\Sigma : A \to B\} \vdash \mathit{UT}' \mathrel{\rhd\!\!\rhd} \Gamma'', E \\ D \notin \mathrm{dom}(\Gamma'')\end{array}}{\mathcal{A},\, \Gamma_{gen},\, \Gamma \vdash \textbf{local } A = \mathit{UT} \textbf{ within } \mathit{UT}' \mathrel{\rhd\!\!\rhd} \Gamma''[D/A], E[D/A]}$$

It should be noted that, as a result of Casl lacking the amalgamation property, checking whether a subcontext ensures amalgamability of a context turns out to be undecidable; the same applies to the problem of checking whether an architectural specification has a denotation w.r.t. the extended static semantics. For these results, as well as for algorithms designed to cope with many typical cases, see [31].

We will make use of the following, purely syntactical, lemma:

Lemma 5.1. *Assume $A \notin \mathcal{A}$, $\mathcal{B}$ is a finite set of unit names and $\mathcal{A}, \Gamma_{gen}, \Gamma \vdash \mathit{UT} \mathrel{\rhd\!\!\rhd} \Gamma', Z$. Then there exists $B \notin \mathcal{B}$ such that $\mathcal{A}, \Gamma_{gen}, \Gamma[B/A] \vdash \mathit{UT} \mathrel{\rhd\!\!\rhd} \Gamma'[B/A], Z[B/A]$.* □

5.2 Soundness and Completeness of the Extended Static Semantics

In this section we state and prove a soundness and completeness theorem for the extended static semantics, linking it to the more primitive static and model semantics. This serves both as an introduction to the soundness and completeness theorem for the proof calculus (see the next section) and as an addendum to the description of the extended static semantics in Sect. III:5.6.

5.2.1 Concepts

In order to describe the relations between the extended static semantics and the model semantics we introduce a syntactic operation $|\cdot|$, which removes everything except the signature from specifications. Thus, $|\mathit{ASP}|$ is the architectural specification *ASP* with specifications used in declarations replaced by pure signatures.

One might expect that an architectural specification *ASP* has a denotation w.r.t. the extended static semantics iff *ASP* has a denotation w.r.t. the static semantics and $|\mathit{ASP}|$ has a denotation w.r.t. the model semantics. Observe that the condition on the right side of the 'iff' seems to be the truly static concept that we are after, while the extended static semantics on the left side is merely our way of approximating that concept.

That the implication from left to right holds (i.e, soundness) is fairly obvious. Unfortunately, the reverse implication is false (i.e., no completeness). This means that in some cases the extended static semantics will fail for statically perfectly correct architectural specifications. There are two reasons for this and, as we will see, they show that those 'statically perfectly correct' specifications are not actually that perfect – hence, the extended static semantics is in fact complete w.r.t. a slightly adjusted 'truly static concept'.

First, the model semantics is *applicative*, that is, repetitive application of the same generic unit to the same argument must yield the same result. The extended static semantics does not keep track of which unit is applied where, and because of this there are examples of architectural specifications *ASP* such that *ASP* has no denotation w.r.t. the extended static semantics, although $|\mathit{ASP}|$ does have a denotation w.r.t. the model semantics (see [28]). To circumvent this problem, we introduce a modified, *generative* version of the model semantics, which we will denote by $\Rightarrow^g$. Note that using a generative semantics is quite common, as exemplified by the generative functors of Standard ML, and that it serves a methodological purpose, making the design more transparent. The generative semantics may be defined either by allowing units to be multi-valued functions, or syntactically, by treating each declaration $F : \mathit{SP} \to \mathit{SP}'$ such that F is applied $n > 0$ times in the given architectural specification as a list of declarations $F_1, \ldots, F_n : \mathit{SP} \to \mathit{SP}'$, and then treating the ith application of F as an application of F_i. We will be only interested whether an architectural specification has a denotation at all w.r.t. the generative semantics and therefore it makes no difference for us which of the above approaches is taken. Note that if no generic unit in *ASP* is applied more than once then the generative and applicative semantics are equivalent.

The reason for which even the equivalence "$\vdash ASP \mathrel{\rhd\!\!\!\rhd} \Box$ iff *ASP* has a denotation w.r.t. the static semantics and $|ASP|$ has a denotation w.r.t. the *generative* model semantics" does not hold is that in CASL the consistency of a generic unit specification $SP \to SP'$ does not guarantee that $|SP \to SP'|$ is consistent. The simplest example is when $SP = \{$ **sorts** s, s'; **axioms** $\forall x, y : s \cdot x = y$; $\}$ and $SP' = SP$ **then** $\{$ **sorts** $s < s'$; $\}$. As a consequence, it may happen that *ASP* has no denotation w.r.t. the extended static semantics, while $|ASP|$ does have a denotation w.r.t. the model semantics for a trivial reason: *ASP*'s declarations were consistent, while $|ASP|$'s are not.

To cope with this issue, we define a *partial model semantics*, denoted using $\Rightarrow_\bot$. The idea here is that we make all generic unit signatures into *consistent* specifications, simply by supplying an additional possible value, namely $\bot$. However, before we dive into the details needed to state in full generality the soundness and completeness of extended static semantics (which we do in Theorem 5.4), we state the following simpler corollary of that theorem:

Corollary 5.2. *Let ASP be an architectural specification and assume that no generic unit in ASP is applied more than once and no generic unit specification in $|ASP|$ is inconsistent. Then ASP has a denotation w.r.t. the extended static semantics if and only if ASP has a denotation w.r.t. the static semantics and $|ASP|$ has a denotation w.r.t. the model semantics.* □

We now introduce the machinery that is necessary in order to drop the special assumptions about generic units in the above result. Let $\mathbf{Mod}_\bot(\Sigma) = \mathbf{Mod}(\Sigma) \cup \{\bot\}$, $[\![SP]\!]_\bot = [\![SP]\!] \cup \{\bot\}$, $\mathbf{Unit}_\bot(\Sigma) = \mathbf{Mod}_\bot(\Sigma)$ and $\mathbf{Unit}_\bot(\Sigma \to \Sigma')$ be the set of all partial functions $F : \mathbf{Mod}_\bot(\Sigma) \to \mathbf{Mod}_\bot(\Sigma')$ such that $\bot \in \mathrm{dom}(F)$, $F(\bot) = \bot$, and for all $M \in \mathrm{dom}(F)$, $F(M) \neq \bot$ implies $F(M)|_\Sigma = M$. Finally, for any generic unit specification $SP \to SP'$, by $[\![SP \to SP]\!]_\bot$ we denote the set of all $F \in \mathbf{Unit}_\bot(sig[SP] \to sig[SP'])$ such that $\mathrm{dom}(F) = [\![SP]\!]_\bot$ and for all $M \in \mathrm{dom}(F)$, $F(M) \in [\![SP']\!]_\bot$. For any signature Σ by a $\bot$-Σ-*model* we will mean an element of $\mathbf{Mod}_\bot(\Sigma)$. Further, for $\sigma : \Sigma \to \Sigma'$ let $\cdot|^\bot_\sigma : \mathbf{Mod}_\bot(\Sigma') \to \mathbf{Mod}_\bot(\Sigma)$ denote the standard reduct $\cdot|_\sigma$ which additionally takes $\bot$ to $\bot$.

Below we present those rules of the partial model semantics, which differ from the standard rules:

$$\frac{}{\vdash A : SP \to SP' \Rightarrow_\bot \{\, \{A \mapsto F\} \mid F \in [\![SP \to SP']\!]_\bot \,\}}$$

$$\frac{\begin{array}{c} C_s \cup \{A \mapsto sig[SP]\} \vdash UT \rhd \Sigma \\ C_s \cup \{A \mapsto sig[SP]\},\ \{\, E \cup \{A \mapsto M\} \mid E \in C, M \in \llbracket SP \rrbracket \,\} \vdash UT \Rightarrow_\perp MEv \\ \text{for all } E \in C \text{ and } M \in \llbracket SP \rrbracket \text{ if } MEv(E \cup \{A \mapsto M\}) \neq \perp \text{ then} \\ MEv(E \cup \{A \mapsto M\})|_{sig[SP]} = M \end{array}}{\begin{array}{l} C_s,\, C \vdash \lambda A : SP \bullet UT \Rightarrow_\perp \\ \qquad \lambda E \in C \cdot \lambda M \in \llbracket SP \rrbracket_\perp \cdot \ \text{if } M = \perp \text{ then } \perp \text{ else } MEv(E \cup \{A \mapsto M\}) \end{array}}$$

$$\frac{\begin{array}{c} C_s,\, C \vdash UT \Rightarrow_\perp MEv \\ \text{for all } E \in C,\ MEv(E)|^{\perp}_{\sigma} \in \text{dom}(E(A)) \\ C_s(A) = \Sigma_f \to \Sigma_r \\ \tau : \Sigma_a \to \Delta \text{ and } \sigma' : \Sigma_r \to \Delta \text{ form the selected pushout for } (\sigma, \iota_{\Sigma_f \subseteq \Sigma_r}) \\ \text{for all } E \in C, \text{ if } E(A)(MEv(E)|^{\perp}_{\sigma}) \neq \perp, \text{ then} \\ \text{there exists a unique } \Delta\text{-model } M_E \text{ such} \\ \text{that } M_E|_\tau = MEv(E) \text{ and } M_E|_{\sigma'} = E(A)(MEv(E)|_\sigma) \end{array}}{\begin{array}{l} C_s,\, C \vdash A\ [\ UT\ \textbf{fit}\ \sigma : \Sigma_f \to \Sigma_a\ ] \Rightarrow_\perp \\ \qquad \lambda E \in C \cdot \ \text{if } E(A)(MEv(E)|^{\perp}_{\sigma}) = \perp \text{ then } \perp \text{ else } M_E \end{array}}$$

$$\frac{\begin{array}{c} C_s \vdash UT_1 \rhd \Sigma_1 \\ C_s \vdash UT_2 \rhd \Sigma_2 \\ \Sigma = \Sigma_1 \cup \Sigma_2 \\ C_s,\, C \vdash UT_1 \Rightarrow_\perp MEv_1 \\ C_s,\, C \vdash UT_2 \Rightarrow_\perp MEv_2 \\ \text{for all } E \in C, \text{ if } MEv_1(E) \neq \perp \text{ and } MEv_2(E) \neq \perp, \\ \text{then there exists a unique } \Sigma\text{-model } M_E \text{ such} \\ \text{that } M_E|_{\Sigma_1} = MEv_1(E) \text{ and } M_E|_{\Sigma_2} = MEv_2(E) \end{array}}{\begin{array}{l} C_s,\, C \vdash UT_1\ \textbf{and}\ UT_2 \Rightarrow_\perp \\ \qquad \lambda E \in C \cdot \ \text{if } MEv_1(E) = \perp \text{ or } MEv_2(E) = \perp \text{ then } \perp \text{ else } M_E \end{array}}$$

$$\frac{\begin{array}{c} C_s,\, C \vdash UT \Rightarrow_\perp MEv \\ \text{for all } E \in C \text{ there exists a unique } \perp\text{-}\Sigma'\text{-model } M_E \text{ with } M_E|^{\perp}_{\sigma} = MEv(E) \end{array}}{C_s,\, C \vdash UT\ \textbf{with}\ \sigma : \Sigma \to \Sigma' \Rightarrow_\perp \lambda E \in C \cdot M_E}$$

$$\frac{C_s,\, C \vdash UT \Rightarrow_\perp MEv}{C_s,\, C \vdash UT\ \textbf{reduction}\ \sigma : \Sigma \to \Sigma' \Rightarrow_\perp \lambda E \in C \cdot MEv(E)|^{\perp}_{\sigma}}$$

$$\frac{\begin{array}{c} C_s \vdash UT \rhd \Sigma \\ C_s,\, C \vdash UT \Rightarrow_\perp MEv \\ C_s \cup \{A \mapsto \Sigma\}, \{E \cup \{A \mapsto MEv(E)\} | E \in C,\ MEv(E) \neq \perp\} \vdash UT' \Rightarrow_\perp MEv' \end{array}}{\begin{array}{l} C_s,\, C \vdash \textbf{local}\ A = UT\ \textbf{within}\ UT' \Rightarrow_\perp \\ \qquad \lambda E \in C \cdot \ \text{if } MEv(E) = \perp \text{ then } \perp \text{ else } MEv'(E \cup \{A \mapsto MEv(E)\}) \end{array}}$$

We will combine the partial and generative versions of the model semantics, creating a partial generative semantics, denoted by $\Rightarrow^g_\perp$. The following is an important property of the partial model semantics:

Proposition 5.3. *Suppose that ASP has a denotation w.r.t. the static semantics. If ASP has a denotation w.r.t. the partial (generative) model semantics, then it also has one w.r.t. the standard (generative) model semantics. If no generic unit declaration in ASP is inconsistent, then the reverse implication also holds.* $\square$

We may now state our main soundness and completeness theorem for the extended static semantics:

Theorem 5.4. *For any architectural specification ASP we have* $\vdash ASP \rhd\!\!\!\rhd \square$ *if and only if ASP has a denotation w.r.t. the static semantics and* $|ASP|$ *has a denotation w.r.t. the partial generative model semantics.*

Clearly, the following theorem is then a corollary:

Theorem 5.5. *For any architectural specification ASP in which no generic unit specification is applied more than once we have* $\vdash ASP \rhd\!\!\!\rhd \square$ *if and only if ASP has a denotation w.r.t. the static semantics and* $|ASP|$ *has a denotation w.r.t. the partial model semantics.*

In fact, using the syntactic definition of the generative model semantics one easily sees that Theorem 5.5 implies Theorem 5.4. Therefore we will only prove Theorem 5.5. Please note that this proof will be institution-independent.

5.2.2 Proof

In order to prove Theorem 5.5 (and thereby Theorem 5.4 as well), we will now need some auxiliary notions. A $\perp$-unit family $F = \{F_U\}_{U \in \mathrm{dom}(\Gamma_{gen})}$ is $\perp$-*consistent* with a generic context Γ_{gen} if $A : \Sigma \to \Sigma'$ in Γ_{gen} implies $F_A \in [\![\Sigma \to \Sigma']\!]_\perp$ (i.e., $F_A \in \mathbf{Unit}_\perp(\Sigma \to \Sigma')$ and additionally $\mathrm{dom}(F_A) = \mathbf{Mod}_\perp(\Sigma)$).

If $\mathcal{A} \subseteq \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma_{gen})$ then by $C_s(\mathcal{A}, \Gamma_{gen}, \Gamma)$ we denote the static context which takes any unit name in $\mathrm{dom}(\Gamma_{gen})$ to the appropriate generic unit signature defined in Γ_{gen} and any unit name in $\mathcal{A}$ to the appropriate signature defined in Γ.

Let Γ_{gen} be a generic context and UT a unit term. By $\mathcal{P}(UT)$ we will denote the set of generic unit names from $\mathrm{dom}(\Gamma_{gen})$ used in UT, and by $\overline{\mathcal{P}}(UT)$ the set of generic unit names from $\mathrm{dom}(\Gamma_{gen})$ not used in UT; these sets depend on Γ_{gen} as well, but Γ_{gen} being fixed this should not cause confusion.

Lemma 5.6. *Assume that* $\mathcal{A} \subseteq \mathrm{dom}(\Gamma) \setminus \mathrm{dom}(\Gamma_{gen})$ *and that no generic unit is applied more than once in UT. Define* $C_s = C_s(\mathcal{A}, \Gamma_{gen}, \Gamma)$ *and suppose that C fits* C_s *and that there exists a surjective function* θ *from C onto the set of all model families consistent with* Γ*, and such that:*

a) *for any* $E \in C$, $\theta(E)|_{\mathcal{A}} = E|_{\mathcal{A}}$;

b) *if* E_1 *and* E_2 *coincide on* $\mathcal{A} \cup \overline{\mathcal{P}}(UT)$, *then* $\theta(E_1) = \theta(E_2)$ *(in the strong sense, i.e., either both are undefined, or both are defined and equal).*

Then $\mathcal{A}, \Gamma_{gen}, \Gamma \vdash UT \mathrel{\rhd\!\!\!\rhd} \Gamma_1, Z$ *for some* Γ_1 *and* Z *if and only if* $C_s \vdash UT \rhd \Sigma$ *for some* Σ *and* $C_s, C \vdash UT \Rightarrow_{\perp} MEv$ *for some* MEv.

Moreover, if both sides of the equivalence hold, then:

1. $Z : \Sigma$ *in* Γ_1;
2. *there exists a surjective partial function* θ_1 *from* C *onto the set of all model families consistent with* Γ_1, *and such that:*
   a) for any $E \in C$, *if* $\theta_1(E)$ *is defined, then* $\theta_1(E)|_{\mathrm{dom}(\Gamma)} = \theta(E)$;
   b) if $E_1, E_2 \in C$ *coincide on* $\mathcal{P}(UT)$ *and if* $\theta(E_1) = \theta(E_2)$, *then* $\theta_1(E_1) = \theta_1(E_2)$ *(in the strong sense);*
   c) $MEv = \lambda E \in C \cdot$ *if* $E \in \mathrm{dom}(\theta_1)$ *then* $\theta_1(E)_Z$ *else* $\perp$.

Proof. The proof is by induction over the structure of UT.

The idea is that θ defines the way in which any environment $E \in C$ is represented by a model family consistent with Γ. We assume that all families consistent with Γ are used as representations, that they really are representations (condition **a**), and that they do not depend on generic units used in UT (condition **b**), which is possible, since no generic unit may be used both inside and outside UT at the same time.

Then the function θ_1 is a kind of extension of θ. It is undefined in those cases where the result model is $\perp$. Point (2b) again states that the result may only depend on generic units actually used in UT.

The **first case** is $UT = A$. Assume the left side of the equivalence holds. Then $\Gamma_1 = \Gamma$, $Z = A$ and $A \in \mathcal{A}$. Thus $C_s \vdash UT \rhd C_s(A)$ and trivially $C_s, C \vdash UT \Rightarrow_{\perp} \lambda E \in C \cdot E(A)$. As for the 'moreover' part, (1) is obvious.We then take $\theta_1 = \theta$. Clearly, for $E \in C$ we have $MEv(E) = E(A) = E(Z) = \theta(E)_Z = \theta_1(E)_Z$, which proves point (2).

So, assume now that the right side holds. Then $C_s(A) = \Sigma$, which implies $A \in \mathcal{A}$, and so $\mathcal{A}, \Gamma_{gen}, \Gamma \vdash UT \mathrel{\rhd\!\!\!\rhd} \Gamma, A$.

The **second case** is $UT = A$ [UT' **fit** $\sigma : \Sigma_f \to \Sigma_a$].

Assume the left side holds. Thus, by the extended static semantics rule for unit application, we have:

I.A $A : \Sigma_f \to \Sigma_r$ in Γ_{gen} for some Σ_r;
I.B $\mathcal{A}, \Gamma_{gen}, \Gamma \vdash UT' \mathrel{\rhd\!\!\!\rhd} \Gamma_a, A_a$ for some Γ_a and A_a;
I.C $A_a : \Sigma_a$ in Γ_a;
I.D there exists a selected pushout $\tau : \Sigma_a \to \Delta$, $\sigma' : \Sigma_r \to \Delta$ for $(\sigma, \iota_{\Sigma_f \subseteq \Sigma_r})$;
I.E $Z \notin \mathrm{dom}(\Gamma_a)$ and there exist $A_f, A_r \notin \mathrm{dom}(\Gamma_a)$, all distinct, such that for $\Gamma' = \Gamma_a \cup \{A_f : \Sigma_f,\ \sigma : A_f \to A_a,\ A_r : \Sigma_r,\ \iota_{\Sigma_f \subseteq \Sigma_r} : A_f \to A_r\}$ and $\Gamma_1 = \Gamma' \cup \{Z : \Delta,\ \tau : A_a \to Z,\ \sigma' : A_r \to Z\}$ we have that Γ' ensures amalgamability for Γ_1.

Using (I.B) and the induction hypothesis we infer that there exist Σ and MEv_a such that:

II.A $C_s \vdash \mathit{UT}' \rhd \Sigma$;
II.B $C_s, C \vdash \mathit{UT}' \Rightarrow_\perp \mathit{MEv}_a$;
II.C $A_a : \Sigma$ in Γ_a and so by (I.C) we have $\Sigma = \Sigma_a$;
II.D there exists a surjective partial function θ_a from C onto the set of all model families consistent with Γ_a, and such that:

1. for any $E \in C$, if $\theta_a(E)$ is defined, then $\theta_a(E)|_{\mathrm{dom}(\Gamma)} = \theta(E)$;
2. if $E_1, E_2 \in C$ coincide on $\mathcal{P}(\mathit{UT}) \setminus \{A\}$ and if $\theta(E_1) = \theta(E_2)$, then $\theta_a(E_1) = \theta_a(E_2)$ (in the strong sense);
3. $\mathit{MEv}_a = \lambda E \in C \cdot$ if $E \in \mathrm{dom}(\theta_a)$ then $\theta_a(E)_{A_a}$ else $\perp$.

Because, by (I.A), $C_s(A) = \Sigma_f \to \Sigma_r$, $\Sigma = \Sigma_a$, (II.A) and (I.D) we may conclude that $C_s \vdash \mathit{UT} \rhd \Delta$.

To prove $C_s, C \vdash \mathit{UT} \Rightarrow_\perp \mathit{MEv}$ for some MEv we need only take any $E \in C$ with $E(A)(\mathit{MEv}_a(E)|^{\perp}_{\sigma}) \neq \perp$ and show that there exists a unique Δ-model M_E such that $M_E|_\tau = \mathit{MEv}_a(E)$ and $M_E|_{\sigma'} = E(A)(\mathit{MEv}_a(E)|_\sigma)$. Since $\mathit{MEv}_a(E) \neq \perp$, the model family $N = \theta_a(E)$ is defined and consistent with Γ_a and $N_{A_a} = \mathit{MEv}_a(E)$. Defining additionally $N_{A_f} = \mathit{MEv}_a(E)|_\sigma$ and $N_{A_r} = E(A)(\mathit{MEv}_a(E)|_\sigma) \neq \perp$ we see that we have a model family consistent with Γ'. By (I.E) it uniquely extends to a family Q consistent with Γ_1, which ends the proof from left to right, since as M_E we may take Q_Z.

As for the 'moreover' part, (I.E) implies (1). For any $E \in C$ we then define the $\theta_1(E)$ of (2) as follows. If $E(A)(\mathit{MEv}_a(E)|^{\perp}_{\sigma}) = \perp$, then we let $\theta_1(E)$ be undefined; otherwise $\theta_1(E)$ is defined to be $\theta_a(E)$ extended by $\theta_1(E)_{A_f} = \theta_a(E)|_\sigma$, $\theta_1(E)_{A_r} = M_E|_{\sigma'}$ and $\theta_1(E)_Z = M_E$. Clearly, if defined, $\theta_1(E)$ is consistent with Γ_1. Also, θ_1 is onto, for let Q be any model family consistent with Γ_1. Then $Q|_{\mathrm{dom}(\Gamma_a)}$ is consistent with Γ_a, and so, by (II.D), there exists $E \in \mathrm{dom}(\theta_a)$ such that $\theta_a(E) = Q|_{\mathrm{dom}(\Gamma_a)}$. We may now define E' to be E changed so that $E'(A)$ is a unit taking Q_{A_f} to Q_{A_r} and everything else to $\perp$. We then have $\theta_1(E') = Q$, since by definition $\theta_1(E')|_{\mathrm{dom}(\Gamma')} = Q|_{\mathrm{dom}(\Gamma')}$ and Γ' ensures amalgamability for Γ_1. Finally:

1. for $E \in C$, if $\theta_1(E)$ is defined, then $\theta_1(E)|_{\mathrm{dom}(\Gamma)} = \theta_a(E)|_{\mathrm{dom}(\Gamma)} = \theta(E)$ (the second equality by virtue of point 1 of (II.D));
2. if $E_1, E_2 \in C$ coincide on $\mathcal{P}(\mathit{UT}) \ni A$ and if $\theta(E_1) = \theta(E_2)$, then by point 2 of (II.D) we have $\theta_a(E_1) = \theta_a(E_2)$, and so $\theta_1(E_1) = \theta_1(E_2)$;
3. for $E \in C$, if $E(A)(\mathit{MEv}_a(E)|^{\perp}_{\sigma}) = \perp$ then $\theta_1(E)$ is undefined and $\mathit{MEv}(E) = \perp$; otherwise $\theta_1(E) = M_E = \mathit{MEv}(E)$.

Finally, we prove the implication from right to left.

Assume that $C_s \vdash \mathit{UT} \rhd \Sigma$ and $C_s, C \vdash \mathit{UT} \Rightarrow_\perp \mathit{MEv}$. From the static and partial model semantics rule for unit application we know that:

I.A $C_s(A) = \Sigma_f \to \Sigma_r$ for some Σ_r;
I.B $C_s \vdash \mathit{UT}' \rhd \Sigma_a$;
I.C there is a selected pushout $\tau : \Sigma_a \to \Sigma$, $\sigma' : \Sigma_r \to \Sigma$ for $(\sigma, \iota_{\Sigma_f \subseteq \Sigma_r})$;
I.D $C_s, C \vdash \mathit{UT}' \Rightarrow_\perp \mathit{MEv}_a$ for some MEv_a;

I.E for all $E \in C$, if $E(A)(MEv_a(E)|^{\perp}_{\sigma}) \neq \perp$, then there exists a unique Σ-model M_E such that $M_E|_{\tau} = MEv_a(E)$ and $M_E|_{\sigma'} = E(A)(MEv_a(E)|_{\sigma})$.

By (I.B) and (I.D) and the induction hypothesis we may infer that:

II.A $\mathcal{A}, \Gamma_{gen}, \Gamma \vdash UT' \rhd\!\!\!\rhd \Gamma_a, A_a$ for some Γ_a and A_a;
II.B $A_a : \Sigma_a$ in Γ_a;
II.C there exists a surjective partial function θ_a from C onto the set of all model families consistent with Γ_a, and such that:
1. for any $E \in C$, if $\theta_a(E)$ is defined, then $\theta_a(E)|_{\mathrm{dom}(\Gamma)} = \theta(E)$;
2. if $E_1, E_2 \in C$ coincide on $\mathcal{P}(UT) \setminus \{A\}$ and if $\theta(E_1) = \theta(E_2)$, then $\theta_a(E_1) = \theta_a(E_2)$ (in the strong sense);
3. $MEv_a = \lambda E \in C \cdot$ if $E \in \mathrm{dom}(\theta_a)$ then $\theta_a(E)_{A_a}$ else $\perp$.

Take arbitrary distinct $A_f, A_r, Z \notin \mathrm{dom}(\Gamma_a)$ and let $\Gamma' = \Gamma_a \cup \{A_f : \Sigma_f,\ \sigma : A_f \to A_a,\ A_r : \Sigma_r,\ \iota_{\Sigma_f \subseteq \Sigma_r} : A_f \to A_r\}$ and $\Gamma_1 = \Gamma' \cup \{Z : \Sigma,\ \tau : A_a \to Z,\ \sigma' : A_r \to Z\}$. Because we have (I.A), (II.A), (II.B) and (I.C) all we need in order to prove $\mathcal{A}, \Gamma_{gen}, \Gamma \vdash UT \rhd\!\!\!\rhd \Gamma_1, Z$ is show that Γ' ensures amalgamability for Γ_1.

So, take any model family Q consistent with Γ'. Since, by (II.C), θ_a is onto, there must exist $E_a \in C$ such that $\theta_a(E_a) = Q|_{\mathrm{dom}(\Gamma_a)}$. Let E be E_a changed so that $E(A)$ is a unit taking Q_{A_f} to Q_{A_r} and everything else to $\perp$. Observe that by condition **b** and point 2 of (II.C) $MEv_a(E) = MEv_a(E_a) = Q_{A_a}$; also, $E(A)(MEv_a(E)|^{\perp}_{\sigma}) = E(A)(Q_{A_a}|_{\sigma}) = E(A)(Q_{a_f}) = Q_{A_r}$, so, by (I.E), there exists a unique Σ-model M_E such that $M_E|_{\tau} = MEv_a(E) = Q_{A_a}$ and $M_E|_{\sigma'} = E(A)(MEv_a(E)|_{\sigma}) = Q_{A_r}$. This is equivalent to saying that $Q \cup \{Z \mapsto M_E\}$ is the unique model family consistent with Γ_1 and extending Q.

The **cases** of **and**, **with** and **reduction** are fairly easy and we omit them.

The **final case** is $UT =$ **local** $A = UT'$ **within** UT''.

Assume the left side holds. Thus, by the extended static semantics rule for local unit definition, we have:

I.A $\mathcal{A}, \Gamma_{gen}, \Gamma \vdash UT' \rhd\!\!\!\rhd \Gamma', B$ for some Γ' and B;
I.B $B : \Sigma_B$ in Γ' for some Σ_B;
I.C $A \notin \mathrm{dom}(\Gamma') \cup \mathrm{dom}(\Gamma_{gen})$;
I.D for $\mathcal{A}_* = \mathcal{A} \cup \{A\}$ and $\Gamma_* = \Gamma' \cup \{A \mapsto \Sigma_B, \mathrm{id}_{\Sigma_B} : A \to B\}$ we have $\mathcal{A}_*, \Gamma_{gen}, \Gamma_* \vdash UT'' \rhd\!\!\!\rhd \Gamma'', E$ for some Γ'' and E;
I.E $D \notin \mathrm{dom}(\Gamma'')$;
I.F $\Gamma_1 = \Gamma''[D/A]$ and $Z = E[D/A]$.

Using (I.A) and the induction hypothesis, we infer that:

II.A $C_s \vdash UT' \rhd \Sigma'$ for some Σ';
II.B $C_s, C \vdash UT' \Rightarrow_{\perp} MEv'$ for some MEv';
II.C $B : \Sigma'$ in Γ' and so, by (I.B), $\Sigma_B = \Sigma'$;
II.D there exists a surjective partial function θ' from C onto the set of all model families consistent with Γ', and such that:

1. for any $E \in C$, if $\theta'(E)$ is defined, then $\theta'(E)|_{\mathrm{dom}(\Gamma)} = \theta(E)$;
2. if $E_1, E_2 \in C$ coincide on $\mathcal{P}(UT')$ and if $\theta(E_1) = \theta(E_2)$, then $\theta'(E_1) = \theta'(E_2)$ (in the strong sense);
3. $MEv' = \lambda E \in C \cdot$ if $E \in \mathrm{dom}(\theta')$ then $\theta'(E)_B$ else $\perp$.

Observe that $\mathcal{A}_* \subseteq \mathrm{dom}(\Gamma_*) \setminus \mathrm{dom}(\Gamma_{gen})$ (by (I.C)) and that no generic unit name in UT'' is applied more than once. Let $C_s^* = C_s(\mathcal{A}_*, \Gamma_{gen}, \Gamma_*)$, C^* be the context consisting of environments of the form $E \cup \{A \mapsto MEv'(E)\}$, where $E \in C$ and $MEv'(E) \neq \perp$, and define $\theta^*(E^*) = \theta'(E^*|_{\mathrm{dom}(E^*)\setminus\{A\}}) \cup \{A \mapsto E^*(A)\}$ for $E^* \in C^*$.

Then θ^* is a function from C^* onto the set of model families consistent with Γ^*. First, clearly $\theta^*(E^*)|_{\mathrm{dom}(\Gamma')}$ is consistent with Γ', and moreover $\theta^*(E^*)_A = E^*(A) = MEv'(E^*|_{\mathrm{dom}(E^*)\setminus\{A\}}) = \theta'(E^*|_{\mathrm{dom}(E^*)\setminus A})_B = \theta^*(E^*)_B$. Second, θ^* is onto, for take any model family Q consistent with Γ_*. There exists an environment $E \in C$ with $\theta'(E) = Q|_{\mathrm{dom}(\Gamma')}$. Then setting $E^* = E \cup \{A \mapsto MEv'(E)\}$ we have $E^* \in C^*$ and $\theta^*(E^*) = Q$.

Also, θ^* satisfies conditions **a** and **b**:

a) if $E^* \in C^*$, then $\theta^*(E^*)|_{\mathcal{A}_*} = E^*|_{\mathcal{A}_*}$, since the equalities $\theta^*(E^*)|_{\mathcal{A}} = \theta'(E^*|_{\mathrm{dom}(E^*)\setminus\{A\}})|_{\mathcal{A}} = E^*|_{\mathcal{A}}$ hold and since $\theta^*(E^*)_A = E^*(A)$;

b) if E_1^* and E_2^* coincide on $\mathcal{A}_* \cup \overline{\mathcal{P}}(UT'')$ then $\theta^*(E_1^*) = \theta'(E_1^*|_{\mathrm{dom}(E_1^*)\setminus\{A\}}) \cup \{A \mapsto E_1^*(A)\} = \theta'(E_2^*|_{\mathrm{dom}(E_2^*)\setminus\{A\}}) \cup \{A \mapsto E_2^*(A)\} = \theta^*(E_2^*)$.

By (I.D) and the induction hypothesis we may now infer that:

III.A $C_s^* \vdash UT'' \rhd \Sigma''$ for some Σ'';

III.B $C_s^*, C^* \vdash UT'' \Rightarrow_\perp MEv''$ for some MEv'';

III.C $E : \Sigma''$ in Γ'';

III.D there exists a surjective partial function θ'' from C^* onto the set of all model families consistent with Γ'', and such that:

1. for any $E^* \in C^*$, if $\theta''(E^*)$ is defined, then $\theta''(E^*)|_{\mathrm{dom}(\Gamma_*)} = \theta^*(E^*)$;
2. if $E_1^*, E_2^* \in C^*$ coincide on $\mathcal{P}(UT'')$ and if $\theta^*(E_1^*) = \theta^*(E_2^*)$, then $\theta''(E_1^*) = \theta''(E_2^*)$ (in the strong sense);
3. $MEv'' = \lambda E^* \in C^* \cdot$ if $E^* \in \mathrm{dom}(\theta'')$ then $\theta''(E^*)_E$ else $\perp$.

From (I.C), (II.A) and (III.A) we conclude that $C_s \vdash UT \rhd \Sigma''$. From (II.B) and (III.B) we conclude that $C_s, C \vdash UT \Rightarrow_\perp MEv$, where $\mathrm{dom}(MEv) = C$ and for any $E \in C$, if $MEv'(E) = \perp$, then $MEv(E) = \perp$ and otherwise $MEv(E) = MEv''(E \cup \{A \mapsto MEv'(E)\})$. This ends the proof from left to right.

As for the 'moreover' part, $Z = E[D/A] : \Sigma''$ in $\Gamma''[D/A]$, because of (III.C); this proves point (1). Now, for any $E \in C$ let $\theta_1(E)$ be undefined if $MEv'(E) = \perp$, and otherwise be the model family $\theta''(E \cup \{A \mapsto MEv'(E)\})$ with the node A relabeled to D. Clearly, if $\theta_1(E)$ is defined, then it is consistent with Γ_1. Also, θ_1 is onto the set of all model families consistent with Γ_1. To see this, take any model family Q consistent with Γ_1. Let R be Q with the node A relabeled to D. Of course R is a model family consistent with Γ''. Thus, there exists, by (III.D), an environment $E^* \in C^*$ such that $\theta^*(E^*) = R$. Let $E = E^*|_{\mathrm{dom}(E^*)\setminus\{A\}}$. We now have $\theta''(E \cup \{A \mapsto MEv'(E)\}) = \theta''(E^*) = R$. Hence, $\theta_1(E)$ equals R with the node A relabeled to D, which in turn equals Q. Further:

1. for any $E \in C$, if $\theta_1(E)$ is defined, then $\theta_1(E)|_{\mathrm{dom}(\Gamma)} = \theta(E)$ because of point 1 of (III.D) and (II.D);
2. if $E_1, E_2 \in C$ coincide on $\mathcal{P}$ and if $\theta(E_1) = \theta(E_2)$, then $\theta_1(E_1) = \theta_1(E_2)$ (in the strong sense) because of point 2 of (III.D) and (II.D);
3. $MEv = \lambda E \in C \cdot$ if $E \in \mathrm{dom}(\theta_1)$ then $\theta_1(E)_E$ else $\perp$ because of point 3 of (III.D) and (II.D).

Now we prove the implication from right to left.

Assume that $C_s \vdash UT \rhd \Sigma$ and $C_s, C \vdash UT \Rightarrow_\perp MEv$. Thus, by the static and partial model semantics rule for local unit definition, we have:

I.A $A \notin \mathrm{dom}(C_s)$;
I.B $C_s \vdash UT' \rhd \Sigma'$ for some Σ';
I.C $C_s \cup \{A \mapsto \Sigma'\} \vdash UT'' \rhd \Sigma$;
I.D $C_s, C \vdash UT' \Rightarrow_\perp MEv'$ for some MEv';
I.E $C_s \cup \{A \mapsto \Sigma'\}, \{E \cup \{A \mapsto MEv'(E)\} \mid E \in C, MEv'(E) \neq \perp\} \vdash UT'' \Rightarrow_\perp MEv''$;
I.F $\mathrm{dom}(MEv) = C$ and for all $E \in C$, if $MEv'(E) = \perp$ then $MEv(E) = \perp$ and otherwise $MEv(E) = MEv''(E \cup \{A \mapsto MEv'(E)\})$.

From (I.B) and (I.D) and the induction hypothesis we infer that:

II.A $\mathcal{A}, \Gamma_{gen}, \Gamma \vdash UT' \rhd\!\!\rhd \Gamma', B$ for some Γ' and B – since $A \notin \mathcal{A}$ (by (I.A)), we may assume, by Lemma 5.1, that $A \notin \mathrm{dom}(\Gamma')$;
II.B $B : \Sigma'$ in Γ';
II.C there exists a surjective partial function θ' from C onto the set of all model families consistent with Γ', and such that:
  1. for any $E \in C$, if $\theta'(E)$ is defined, then $\theta'(E)|_{\mathrm{dom}(\Gamma)} = \theta(E)$;
  2. if $E_1, E_2 \in C$ coincide on $\mathcal{P}(UT')$ and if $\theta(E_1) = \theta(E_2)$, then $\theta'(E_1) = \theta'(E_2)$ (in the strong sense);
  3. $MEv' = \lambda E \in C \cdot$ if $E \in \mathrm{dom}(\theta')$ then $\theta'(E)_B$ else $\perp$.

Let $\Gamma_* = \Gamma' \cup \{A : \Sigma', \mathrm{id}_{\Sigma'} : A \to B\}$ and $\mathcal{A}_* = \mathcal{A} \cup \{A\}$. Observe that $\mathcal{A}_* \subseteq \mathrm{dom}(\Gamma_*) \setminus \mathrm{dom}(\Gamma_{gen})$, by (I.A), and that no generic unit name in UT'' is applied more than once. Let $C_s^* = C_s(\mathcal{A}_*, \Gamma_{gen}, \Gamma_*)$ and C^* be the context consisting of environments of the form $E^* = E \cup \{A \mapsto MEv'(E)\}$, where $E \in C$ and $MEv'(E) \neq \perp$, and let $\theta^*(E^*) = \theta'(E^*|_{\mathrm{dom}(E^*)\setminus\{A\}})$ for all $E^* \in C^*$. Then θ^* is a surjective function from C^* onto the set of all model families consistent with Γ', and such that:

a) for any $E^* \in C^*$, $\theta^*(E^*)$ extends the model family $\{E^*(U)\}_{U \in \mathcal{A}_*}$;
b) if E_1^* and E_2^* coincide on $\mathcal{A}_* \cup \overline{\mathcal{P}}(UT'')$, then $\theta^*(E_1^*) = \theta^*(E_2^*)$ (in the strong sense).

Thus, by (I.C), (I.E) and the induction hypothesis we may infer that $\mathcal{A}_*, \mathit{\Gamma}_{gen}, \mathit{\Gamma}_* \vdash \mathit{UT}'' \mathrel{\rhd\!\!\!\rhd} \mathit{\Gamma}'', E$ for some $\mathit{\Gamma}''$ and E. From this together with (II.A), (II.B), (I.A) we may conclude that for some $D \notin \mathrm{dom}(\mathit{\Gamma}'')$ we have $\mathcal{A}, \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT} \mathrel{\rhd\!\!\!\rhd} \mathit{\Gamma}''[D/A], E[D/A]$, which completes the proof. □

Proof (Theorem 5.5). Assume $\mathit{ASP} = \textbf{units}\ \mathit{UD}_1 \ldots \mathit{UD}_n\ \textbf{result}\ \mathit{UT}$ (the generic case is similar and we omit it) and that no generic unit in *ASP* is applied more than once.

It should be clear that all we need to prove is that if $\mathit{\Gamma}_{gen}$ and $\mathit{\Gamma}$ have disjoint domains, $\mathcal{A} = \mathrm{dom}(\mathit{\Gamma})$, $C_s = C_s(\mathcal{A}, \mathit{\Gamma}_{gen}, \mathit{\Gamma})$ and C contains all environments $E = F \cup M$, where F is a $\perp$-unit family consistent with $\mathit{\Gamma}_{gen}$ and M is a model family consistent with $\mathit{\Gamma}$, then $\mathcal{A}, \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT} \mathrel{\rhd\!\!\!\rhd} \mathit{\Gamma}_1, Z$ for some $\mathit{\Gamma}_1$ and Z if and only if $C_s \vdash \mathit{UT} \rhd \Sigma$ for some Σ and $C_s, C \vdash \mathit{UT} \Rightarrow_\perp \mathit{MEv}$ for some *MEv*. We may use Lemma 5.6, since $\mathcal{A} \subseteq \mathrm{dom}(\mathit{\Gamma}) \backslash \mathrm{dom}(\mathit{\Gamma}_{gen})$, no generic unit name in *UT* is applied more than once, and the function $\theta(F \cup M) = M$ satisfies conditions **a** and **b**. This completes the proof. □

5.3 The Proof Calculus

While specifications *SP* which appear in architectural specifications need not form an institution themselves, they are assumed to be built over some institution I; in the case of Casl, this is the institution of Casl logic. Because the institution I might not have weak amalgamation, we are forced to transfer part of the verification process to a specification formalism built over an institution J which has this property. Possible choices of J and of the specification formalism over J are discussed below.

We use the notation[2] $\mathcal{SP}$ for finite sets of specifications over a common signature Σ. If M is a Σ-model, then $M \models_\Sigma \mathcal{SP}$ means that $M \models_\Sigma \mathit{SP}$ for all $\mathit{SP} \in \mathcal{SP}$. If *SP* is a Σ-specification, then $\mathit{SP} \mathrel{\approx\!\!\!>}_{\Sigma} \mathcal{SP}$ means that for any Σ-model M, if $M \models_\Sigma \mathcal{SP}$ then $M \models_\Sigma \mathit{SP}$. The relation $\rightsquigarrow$ is the proof-theoretic counterpart of $\mathrel{\approx\!\!\!>}$. It is sound w.r.t. the relation $\mathrel{\approx\!\!\!>}$ if $\rightsquigarrow \subseteq \mathrel{\approx\!\!\!>}$, and it is complete if $\mathrel{\approx\!\!\!>} \subseteq \rightsquigarrow$.

Our aim is to create a calculus in which, using an institution J with a relation $\rightsquigarrow^J$ sound w.r.t. the relation $\mathrel{\approx\!\!\!>}^J$, one could derive statements of the form $\vdash \mathit{ASP} :: \mathit{USP}$ so that the following theorem holds:

Theorem 5.7. *Suppose that* $\vdash \mathit{ASP} \mathrel{\rhd\!\!\!\rhd} \Box$ *and that no generic unit declaration in ASP is inconsistent. If* $\rightsquigarrow^J$ *is complete w.r.t. the relation* $\mathrel{\approx\!\!\!>}^J$*, then*

$$\vdash \mathit{ASP} :: \mathit{USP}$$

if and only if

[2] Many concepts extensively used in this section have been defined in Chap. 4.

$$\vdash \mathit{ASP} \Rightarrow^{g} \mathit{UEv}$$

for some UEv such that for all $E \in \mathrm{dom}(\mathit{UEv})$*,*

$$\mathit{UEv}(E) \in [\![\mathit{USP}]\!].$$

The reason for assuming that no generic unit declaration is inconsistent should be clear: without it the problem is not recursively enumerable, since the problem of proving consistency is not recursively enumerable.

It is possible to have a calculus where the assumption on *ASP* having a denotation with respect to the extended static semantics can be dropped. It is at the same time possible to have a calculus such that the standard (applicative) model semantics can be used in the above theorem. Unfortunately, such calculi are quite complex – we refer the interested reader to [28] and [29].

5.3.1 Definition of the Proof Calculus

We assume that both in the specification formalism over I, as well as in that over J, for any signature morphism $\sigma : \Sigma \to \Delta$ and Σ-specification *SP*, there exists a *translation of SP along* σ, denoted $\sigma(\mathit{SP})$, i.e., a Δ-specification such that for any Δ-model M we have $M \in [\![\sigma(\mathit{SP})]\!]$ if and only if $M|_{\sigma} \in [\![\mathit{SP}]\!]$. We also assume the existence of basic specifications, that is, for any finite set of Σ-sentences there exists a Σ-specification *SP* such that $[\![\mathit{SP}]\!]$ is the class of models satisfying all the sentences from that set. For Casl structured specifications the translations required can be obtained by combining **with** and **and** constructs, the latter being needed if σ is not surjective enough, and finite sets of sentences are captured by Casl basic specifications – this relies on the fact that surjective signature morphisms, signature extensions and the sentences involved are expressible in the Casl syntax.

The contexts introduced below differ from those used in the extended static semantics only by additional specification components. Therefore we will freely apply concepts defined previously for 'bare' contexts to those defined below.

A *generic context* Γ_{gen} is a finite set of declarations $A :_{\Sigma \to \Sigma'} \mathit{SP} \to \mathit{SP}'$, where $\Sigma \to \Sigma'$ is the signature of the generic unit specification $\mathit{SP} \to \mathit{SP}'$.

A *context* Γ is a finite set of declarations of two forms:

- $A :_{\Sigma} \mathcal{SP}$, where $\mathcal{SP}$ is a finite set of specifications over Σ;
- $\sigma : A \to B$, where $A :_{\Sigma_A} \mathcal{SP}_A$ and $B :_{\Sigma_B} \mathcal{SP}_B$ in Γ and $\sigma : \Sigma_A \to \Sigma_B$ is a signature morphism.

A given unit name A may be declared at most once in a context Γ; again, the same applies to unit names in generic contexts Γ_{gen}. We say that $A : \Sigma$ in Γ to express the fact that $A :_{\Sigma} \mathcal{SP}$ in Γ for some $\mathcal{SP}$. Analogously we use the phrase $A : \Sigma \to \Sigma'$ in Γ_{gen}. If Γ_1 and Γ_2 are two contexts and for all

$A \in \mathrm{dom}(\mathit{\Gamma}_1) \cap \mathrm{dom}(\mathit{\Gamma}_2)$ we have $A : \mathit{\Sigma}$ in $\mathit{\Gamma}_1$ if and only if $A : \mathit{\Sigma}$ in $\mathit{\Gamma}_2$, then their sum $\mathit{\Gamma}_1 \cup \mathit{\Gamma}_2$ is a context as well.

Any context $\mathit{\Gamma}$ can be treated as a diagram in the signature category, whose nodes are additionally labeled by sets of specifications (though the signature morphisms in $\mathit{\Gamma}$ need not be specification morphisms). We will say that $\mathit{\Sigma}$, $\{\eta_U\}_{U \in \mathrm{dom}(\mathit{\Gamma})}$ is a *weakly amalgamable cocone* over $\mathit{\Gamma}$, if it is a weakly amalgamable cocone over the above-mentioned 'bare' diagram without the labeling[3].

Recall from Chap. 4, Remark 4.3, that by $R = (\mathit{\Phi}, \alpha, \beta)$ we denote a model-isomorphic simple theoroidal comorphism from the institution I of our interest to some institution J which admits weak amalgamation. We assume that for any $\mathit{\Sigma}$-specification SP in I, there exists a $\mathit{Sig}(\mathit{\Phi}(\mathit{\Sigma}))$-specification $R(\mathit{SP})$ in J such that for any model $M' \models^J \mathit{Ax}(\mathit{\Phi}(\mathit{\Sigma}))$ we have:

$$M' \models^J R(\mathit{SP}) \iff \beta(M') \models^I \mathit{SP}$$

In the presence of basic specifications this is equivalent to assuming that the comorphism R may be extended to a comorphism from an institution of specifications over I to an institution of specifications over J; hence, we call it the *comorphism condition*. Please note that specifications over I may be very different from specifications over J, e.g., the former could be flat specifications and the latter structured specifications. For a more practical example cf. the discussion below concerning the application of these ideas to CASL structured specifications and CASL logic. For any finite set of $\mathit{\Sigma}$-specifications $\mathcal{SP}$ by $\overline{R}(\mathcal{SP})$ we denote the set of specifications $R(\mathcal{SP})$ augmented by a basic specification containing the axioms $\mathit{Ax}(\mathit{\Phi}(\mathit{\Sigma}))$. For any context $\mathit{\Gamma}$ in I, by $R(\mathit{\Gamma})$ we denote the context in J obtained by mapping any declaration $A :_{\mathit{\Sigma}} \mathcal{SP}$ in $\mathit{\Gamma}$ to $A :_{\mathit{Sig}(\mathit{\Phi}(\mathit{\Sigma}))} \overline{R}(\mathcal{SP})$, and any declaration $\sigma : A \to B$ in $\mathit{\Gamma}$ to $\mathit{\Phi}(\sigma) : A \to B$ (where $\mathit{\Phi}(\sigma)$ is treated as a morphism in the signature category of J).

The problem of developing a proof calculus for architectural specifications is parametrized by the institution I and by a formalism of specifications over I. In order to solve this problem as announced in Theorem 5.7 additional parameters are needed: an institution J with weak amalgamation, a specification formalism over J with a relation $\rightsquigarrow^J$ sound w.r.t. $\overset{J}{\approx\!\!>}$, and a simple theoroidal model-isomorphic comorphism $R : I \to J$ such that for any specification SP over I a specification $R(\mathit{SP})$ over J satisfying the comorphism condition may be defined. Moreover, the specification formalisms must both have translations and basic specifications must exist in them.

If we apply the proof calculus to CASL, then the institution I is simply the institution of CASL logic, and the specification formalism over I is the formalism of CASL structured specifications[4]. Of course, the other parameters

[3] For a definition of weakly amalgamable cocone over a diagram see Chap. 4.

[4] Note however that, as announced in the introduction to this chapter, we disregard here the global environment and treat specifications as self-contained entities.

may be chosen in many ways; we will now describe one possible choice which makes use of the tools developed in the previous chapter.

As the comorphism $R : I \to J$ one may take either the embedding of (subsorted) CASL in many-sorted CASL, or its embedding in so-called Enriched CASL – for a discussion of these possibilities see Chap. 4, Remark 4.3. Specifications over J will be pairs $SP = (N, \mathcal{DG})$, where $\mathcal{DG}$ is a development graph over J and N a node in that graph. Then we define $sig[SP] = \Sigma^N$ and $[\![SP]\!] = \mathbf{Mod}_{\mathcal{DG}}(N)$. For any CASL specification SP over I (i.e., over the CASL logic), the specification $R(SP)$ over J may be obtained as follows: first, SP is translated to a pair consisting of a development graph $\mathcal{DG}$ over I and a node N in that graph, as described in Sect. 4.7; next, this development graph is translated via the comorphism R to a development graph $R(\mathcal{DG})$ over the institution J, as described in Sect. 4.3; then, the specification $R(SP)$ over J is simply the pair $(N, R(\mathcal{DG}))$.

Finally, the relation $(N_1, \mathcal{DG}_1) \leadsto^J (N_2, \mathcal{DG}_2)$ holds if and only if $\mathcal{DG}_1 \cup \mathcal{DG}_2 \vdash N_1 \overset{id}{=\!=\!\Rightarrow} N_2$ (we assume here that $\mathcal{DG}_1$ and $\mathcal{DG}_2$ are disjoint).

The disjointness assumption made above shows that the proposed translation would actually be quite ineffective in the presence of a global environment, losing much of the sharing information that could be kept in development graphs. Fortunately, in the proof calculus the non-signature part of specifications of the form $R(SP)$ is used only in proof obligations, which have the general form:

$$\eta(R(SP)) \leadsto^J \bigcup_{i=1\cdots n} \eta_i(\overline{R}(\mathcal{SP}_i)$$

In Remark 4.26 it has been shown how such proof obligations may be discharged while keeping the structure of specifications. When using these methods it is actually not necessary to define $R(SP)$ explicitly.

We now present the rules of a proof calculus which meets the requirements stated in Theorem 5.7.

$$\boxed{\vdash ASP :: USP}$$

$$\frac{\begin{array}{c}\vdash UD_1 :: \Gamma^1_{gen}, \Gamma^1 \\ \vdots \\ \vdash UD_n :: \Gamma^n_{gen}, \Gamma^n \\ \Gamma^1_{gen} \cup \cdots \cup \Gamma^n_{gen}, \Gamma^1 \cup \cdots \cup \Gamma^n \vdash UE :: USP\end{array}}{\vdash \mathbf{units}\ UD_1 \ldots UD_n\ \mathbf{result}\ UE :: USP}$$

$$\boxed{\vdash UD :: \Gamma_{gen}, \Gamma}$$

$$\frac{}{\vdash A : SP :: \emptyset, \{A :_{sig[SP]} \{SP\}\}}$$

$$\frac{}{\vdash A : SP \to SP' :: \{A :_{sig[SP] \to sig[SP']} SP \to SP'\}, \emptyset}$$

$$\boxed{\Gamma_{gen}, \Gamma \vdash UE :: USP}$$

The sets $\mathrm{dom}(\Gamma_{gen})$ and $\mathrm{dom}(\Gamma)$ should be disjoint.

$$\frac{\begin{array}{c} \Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A \\ \text{for all } U \in \mathrm{dom}(\Gamma'), \text{ we have } U :_{\Sigma_U} \mathcal{SP}_U \text{ in } \Gamma' \\ \Sigma, \{\eta_U\}_{U \in \mathrm{dom}(\Gamma')} \text{ is a weakly amalgamable cocone over } R(\Gamma') \\ \eta_A(R(SP)) \rightsquigarrow^J_\Sigma \bigcup_{U \in \mathrm{dom}(\Gamma')} \eta_U(\overline{R}(\mathcal{SP}_U)) \end{array}}{\Gamma_{gen}, \Gamma \vdash UT \text{ qua } UE :: SP}$$

$$\frac{\begin{array}{c} sig[SP] = sig[SP_1] = \Sigma \\ R(SP_1) \rightsquigarrow^J_{Sig(\Phi(\Sigma))} \overline{R}(\{SP\}) \\ R(SP) \rightsquigarrow^J_{Sig(\Phi(\Sigma))} \overline{R}(\{SP_1\}) \\ \Gamma_{gen}, \Gamma \cup \{A :_\Sigma \{SP\}\} \vdash UT :: \Gamma', B \\ B : sig[SP_2] \text{ in } \Gamma' \\ \text{for all } U \in \mathrm{dom}(\Gamma'), \text{ we have } U :_{\Sigma_U} \mathcal{SP}_U \text{ in } \Gamma' \\ \Sigma, \{\eta_U\}_{U \in \mathrm{dom}(\Gamma')} \text{ is a weakly amalgamable cocone over } R(\Gamma') \\ \eta_B(R(SP_2)) \rightsquigarrow^J_\Sigma \bigcup_{U \in \mathrm{dom}(\Gamma')} \eta_U(\overline{R}(\mathcal{SP}_U)) \end{array}}{\Gamma_{gen}, \Gamma \vdash \lambda A : SP \bullet UT :: SP_1 \to SP_2}$$

$$\boxed{\Gamma_{gen}, \Gamma \vdash UT :: \Gamma', A}$$

The context Γ is a subcontext of Γ'.

$$\frac{}{\mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash A :: \mathit{\Gamma}, A}$$

$$\frac{\begin{array}{c} A :_{\Sigma_f \to \Sigma_r} \mathit{SP}_f \to \mathit{SP}_r \text{ in } \mathit{\Gamma}_{gen} \\ \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT} :: \mathit{\Gamma}_a, A_a \\ \text{for all } U \in \mathrm{dom}(\mathit{\Gamma}_a)\text{, we have } U :_{\Sigma_U} \mathcal{SP}_U \text{ in } \mathit{\Gamma}_a \\ \Sigma, \{\eta_U\}_{U \in \mathrm{dom}(\mathit{\Gamma}_a)} \text{ is a weakly amalgamable cocone over } R(\mathit{\Gamma}_a) \\ \eta_{A_a}(R(\sigma(\mathit{SP}_f))) \rightsquigarrow^{J}_{\Sigma} \bigcup_{U \in \mathrm{dom}(\mathit{\Gamma}_a)} \eta_U(\overline{R}(\mathcal{SP}_U)) \\ \tau : \Sigma_a \to \Delta \text{ and } \sigma' : \Sigma_r \to \Delta \text{ form the selected pushout for } (\sigma, \iota_{\Sigma_f \subseteq \Sigma_r}) \\ A_f, A_r, B \notin \mathrm{dom}(\mathit{\Gamma}_a) \end{array}}{\begin{array}{l} \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash A\ [\ \mathit{UT}\ \textbf{fit}\ \sigma : \Sigma_f \to \Sigma_a\ ] :: \mathit{\Gamma}_a \cup \{A_f :_{\Sigma_f} \{\mathit{SP}_f\}, \sigma : A_f \to A_a, \\ \qquad A_r :_{\Sigma_r} \{\mathit{SP}_r\},\ \iota_{\Sigma_f \subseteq \Sigma_r} : A_f \to A_r,\ B :_{\Delta} \emptyset,\ \tau : A_a \to B,\ \sigma' : A_r \to B\}, B \end{array}}$$

$$\frac{\begin{array}{c} \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT}_1 :: \mathit{\Gamma}_1, A_1 \\ \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT}_2 :: \mathit{\Gamma}_2, A_2 \\ A_1 : \Sigma_1 \text{ in } \mathit{\Gamma}_1 \\ A_2 : \Sigma_2 \text{ in } \mathit{\Gamma}_2 \\ \mathrm{dom}(\mathit{\Gamma}_1) \cap \mathrm{dom}(\mathit{\Gamma}_2) = \mathrm{dom}(\mathit{\Gamma}) \\ B \notin \mathrm{dom}(\mathit{\Gamma}_1) \cup \mathrm{dom}(\mathit{\Gamma}_2) \end{array}}{\begin{array}{c} \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT}_1\ \textbf{and}\ \mathit{UT}_2 :: \mathit{\Gamma}_1 \cup \mathit{\Gamma}_2 \cup \\ \{B :_{\Sigma_1 \cup \Sigma_2} \emptyset,\ \iota_{\Sigma_1 \subseteq \Sigma_1 \cup \Sigma_2} : A_1 \to B,\ \iota_{\Sigma_2 \subseteq \Sigma_1 \cup \Sigma_2} : A_2 \to B\}, B \end{array}}$$

$$\frac{\begin{array}{c} \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT} :: \mathit{\Gamma}', A \\ B \notin \mathrm{dom}(\mathit{\Gamma}') \end{array}}{\mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT}\ \textbf{with}\ \sigma : \Sigma \to \Sigma' :: \mathit{\Gamma}' \cup \{B :_{\Sigma'} \emptyset, \sigma : A \to B\}, B}$$

$$\frac{\begin{array}{c} \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT} :: \mathit{\Gamma}', A \\ B \notin \mathrm{dom}(\mathit{\Gamma}') \end{array}}{\mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT}\ \textbf{reduction}\ \sigma : \Sigma \to \Sigma' :: \mathit{\Gamma}' \cup \{B :_{\Sigma} \emptyset, \sigma : B \to A\}, B}$$

$$\frac{\begin{array}{c} \mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT} :: \mathit{\Gamma}', B \\ B : \Sigma \text{ in } \mathit{\Gamma}' \\ A \notin \mathrm{dom}(\mathit{\Gamma}') \\ \mathit{\Gamma}_{gen}, \mathit{\Gamma} \cup \{A :_{\Sigma} \emptyset, \mathrm{id}_{\Sigma} : A \to B\} \vdash \mathit{UT}' :: \mathit{\Gamma}'', E \\ D \notin \mathrm{dom}(\mathit{\Gamma}'') \end{array}}{\mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \textbf{local}\ A = \mathit{UT}\ \textbf{within}\ \mathit{UT}' :: \mathit{\Gamma}''[D/A], E[D/A]}$$

The following lemma is analogous to Lemma 5.1:

Lemma 5.8. *Assume $A \notin \mathcal{A}$, $\mathcal{B}$ is a finite set of unit names, $C_s(\mathcal{A}, |\mathit{\Gamma}_{gen}|, |\mathit{\Gamma}|) \vdash \mathit{UT} \rhd \Sigma$ and $\mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash \mathit{UT} :: \mathit{\Gamma}', Z$. Then there exists $B \notin \mathcal{B}$ such that $\mathit{\Gamma}_{gen}, \mathit{\Gamma}[B/A] \vdash \mathit{UT} :: \mathit{\Gamma}'[B/A], Z[B/A]$.* □

5.3.2 Soundness and Completeness

Theorem 5.7 is a corollary of the following theorem:

Theorem 5.9. *Suppose that* $\vdash ASP \rhd\!\!\!> \Box$*, that no generic unit declaration in ASP is inconsistent and that no generic unit in ASP is applied more than once. If* $\rightsquigarrow^J$ *is complete w.r.t. the relation* $\overset{J}{\approx\!\!>}$*, then* $\vdash ASP :: USP$ *if and only if* $\vdash ASP \Rightarrow UEv$ *for some UEv such that for all* $E \in \mathrm{dom}(UEv)$, $UEv(E) \in [\![USP]\!]$.

Before we provide a proof of Theorem 5.9, we would like to comment on the above calculus. It is clear that it is really an algorithm presented in the form of a calculus. This algorithm generates proof obligations of the form $SP \rightsquigarrow^J \mathcal{SP}$, and all non-trivial decisions concerning proof-search have to be taken in the process of discharging those obligations. A calculus for discharging proof obligations of this form is described in the previous chapters, where they are translated into theorem links in development graphs (see Definition 4.22). A solution for the specific case of proof obligations generated by the architectural specification proof calculus is provided in Remark 4.26.

What remains to be done now is to prove Theorem 5.7.

For any generic context Γ_{gen} and context Γ, by $|\Gamma_{gen}|$ and $|\Gamma|$ we denote these contexts after removing all specifications and leaving the signatures only (i.e., we obtain contexts in the sense of Sect. 5.1).

Let Γ be a context. A model family M consistent with $|\Gamma|$ is *consistent with* Γ if $M_A \models_\Sigma \mathcal{SP}$ for $A :_\Sigma \mathcal{SP}$ in Γ. Similarly, let Γ_{gen} be a generic context. A unit family F consistent with $|\Gamma_{gen}|$ is *consistent with* Γ_{gen} if for any $U :_{\Sigma\rightarrow\Sigma'} SP \rightarrow SP'$ in Γ_{gen} we have $F_U \in [\![SP \rightarrow SP']\!]$.

The following lemma explains the meaning of the proof obligations 'generated' by the above calculus:

Lemma 5.10. *Let* Γ *be a context with* $U :_{\Sigma_U} \mathcal{SP}_U$ *for all* $U \in \mathrm{dom}(\Gamma)$*. Also, let* $A \in \mathrm{dom}(\Gamma)$ *and SP be a* Σ_A*-specification. Finally, assume that the sink* $\Sigma, \{\tau_U\}_{U\in\mathrm{dom}(\Gamma)}$ *is a weakly amalgamable cocone over* $R(\Gamma)$*. Then the refinement* $\eta_A(R(SP)) \overset{J}{\underset{\Sigma}{\approx\!\!>}} \bigcup_{U\in\mathrm{dom}(\Gamma)} \eta_U(\overline{R}(\mathcal{SP}_U))$ *holds iff for any model family M consistent with* Γ *we have* $M_A \models^I_{\Sigma_A} SP$*.*

Proof. Notice that β defines a bijection between model families consistent with Γ and model families consistent with $R(\Gamma)$. Moreover, this bijection preserves and reflects the satisfaction of specifications. Thus the right side is equivalent to saying that for any model family N consistent with $R(\Gamma)$ we have $N_A \models^J_{Sig(\Phi(\Sigma_A))} R(SP)$. This statement is equivalent to the left side, since $\Sigma, \{\eta_U\}_{U\in\mathrm{dom}(\Gamma)}$ is a weakly amalgamable cocone over $R(\Gamma)$. □

If E fits C_s, then define $E_\perp$ to be the environment taking any $U \in \mathrm{dom}(C_s)$ with $C_s(U) = \Sigma \rightarrow \Sigma'$ to the $\perp$-unit $\lambda M \in \mathbf{Mod}_\perp(\Sigma) \cdot$ if $M \in \mathrm{dom}(E(U))$ then $E(U)(M)$ else $\perp$, and any $U \in \mathrm{dom}(C_s)$ with $C_s(U) = \Sigma$ to $E(U)$.

Lemma 5.11. *Assume that $\mathcal{A} \subseteq \mathrm{dom}(\mathit{\Gamma}) \backslash \mathrm{dom}(\mathit{\Gamma}_{gen})$ and that no generic unit is applied more than once in UT. Define $C_s = C_s(\mathcal{A}, |\mathit{\Gamma}_{gen}|, |\mathit{\Gamma}|)$ and suppose that C fits C_s, that $\{E(U)\}_{U \in \mathrm{dom}(\mathit{\Gamma}_{gen})}$ is a unit family consistent with $\mathit{\Gamma}_{gen}$ for all $E \in C$, and that there exists a surjective function θ from C onto the set of all model families consistent with $\mathit{\Gamma}$, and such that:*

a) *for any $E \in C$, $\theta(E)|_{\mathcal{A}} = E|_{\mathcal{A}}$;*
b) *if E_1 and E_2 coincide on $\mathcal{A} \cup \overline{\mathcal{P}}(UT)$, then $\theta(E_1) = \theta(E_2)$ (in the strong sense).*

Assume also that we have a context $|C|$ which fits C_s and such that $E \in C$ implies $E_\perp \in |C|$, that $C_s \vdash UT \rhd \mathit{\Sigma}$, and that $C_s, |C| \vdash UT \Rightarrow_\perp MEv_\perp$.

If $\rightsquigarrow^J$ is complete w.r.t. the relation $\overset{J}{\approx\!\!\!\!\rightsquigarrow}$, then $\mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash UT :: \mathit{\Gamma}_1, Z$ for some $\mathit{\Gamma}_1$ and Z if and only if $C_s, C \vdash UT \Rightarrow MEv$ for some MEv.

Moreover, if both sides of the equivalence hold, then:

1. *$Z : \mathit{\Sigma}$ in $\mathit{\Gamma}_1$;*
2. *there exists a surjective total function θ_1 from C onto the set of all model families consistent with $\mathit{\Gamma}_1$, and such that:*
   a) for any $E \in C$, $\theta_1(E)|_{\mathrm{dom}(\mathit{\Gamma})} = \theta(E)$;
   b) if $E_1, E_2 \in C$ coincide on $\mathcal{P}(UT)$ and if $\theta(E_1) = \theta(E_2)$, then $\theta_1(E_1) = \theta_1(E_2)$;
   c) $MEv = \lambda E \in C \cdot \theta_1(E)_Z = \lambda E \in C \cdot MEv_\perp(E_\perp)$.

□

The proof of the above lemma is long and very similar to that of Lemma 5.6 and therefore we omit it. In the proof, Lemmas 5.8 and 5.10 are used.

Proof (Theorem 5.9). Assume that ASP has a denotation w.r.t. the extended static semantics, that no generic unit specification in it is inconsistent, and that no such unit is applied more than once. We will prove the theorem for the case $ASP =$ **units** $UD_1 \cdots UD_n$ **result** UT, the generic case being very similar.

Suppose $\vdash UD_i :: \mathit{\Gamma}^i_{gen}, \mathit{\Gamma}^i$ for $1 \leq i \leq n$, and let $\mathit{\Gamma}_{gen} = \mathit{\Gamma}^1_{gen} \cup \cdots \cup \mathit{\Gamma}^n_{gen}$, $\mathit{\Gamma} = \mathit{\Gamma}^1 \cup \cdots \cup \mathit{\Gamma}^n$, $\mathcal{A} = \mathrm{dom}(\mathit{\Gamma})$ and $\mathcal{P}$ be the set of generic unit names from $\mathrm{dom}(\mathit{\Gamma}_{gen})$ used in UT. Also, let $C_s = C_s(\mathcal{A}, |\mathit{\Gamma}_{gen}|, |\mathit{\Gamma}|)$ and let $|C|$ contain all environments of the form $F \cup M$, where F is a $\perp$-unit family $\perp$-consistent with $|\mathit{\Gamma}_{gen}|$ and M is a model family consistent with $|\mathit{\Gamma}|$. By Theorem 5.5 we then have $C_s \vdash UT \rhd \mathit{\Sigma}$ for some $\mathit{\Sigma}$ and $C_s, |C| \vdash UT \Rightarrow_\perp MEv_\perp$ for some $MEv_\perp$. Now let C contain all environments of the form $F \cup M$, where F is a unit family consistent with $\mathit{\Gamma}_{gen}$ and M is a model family consistent with $\mathit{\Gamma}$, and define $\theta(F \cup M) = M$. Clearly, $E \in C$ implies $E_\perp \in |C|$, θ is onto the set of all model families consistent with $\mathit{\Gamma}$, $\theta(F \cup M)$ extends M and does not depend on unit names not in $\mathcal{A}$.

By Lemma 5.11 we have that $\mathit{\Gamma}_{gen}, \mathit{\Gamma} \vdash UT :: \mathit{\Gamma}', A$ for some $\mathit{\Gamma}'$ and A iff $C_s, C \vdash UT \Rightarrow MEv$ for some MEv. Moreover, if both sides hold, then there exists a surjective total function θ_1 from C onto the set of all model families consistent with Γ', and such that $MEv = \lambda E \in C \cdot \theta_1(E)_A$.

If $\vdash ASP :: SP$, then the left side of the above equivalence holds, hence, so does the right one. Then, using one direction of Lemma 5.10, we see that indeed $MEv(E) \in [\![SP]\!]$ for all $E \in C$.

If, on the other hand, $\vdash ASP \Rightarrow MEv$, then the right side of the above equivalence holds, and $MEv(E) \in [\![SP]\!]$ for all $E \in C$. Then, using the other direction of Lemma 5.10, we see that $\vdash ASP :: SP$. □

6

Specification Library Calculus

The aim of the proof calculus for libraries is to capture the well-formedness of a library in terms of proof rules. While the static semantics of libraries already has the format of such rules, the model semantics has not – it is based on set-theoretic notions, and one would have to use to the rules of set theory to reason about it. We here sketch direct calculus rules instead.

The proof calculus for libraries is based on the proof calculi for basic, structured and architectural specifications developed in the preceding chapters.

A library is correct if:

- the static semantics of the library succeeds (Chap. III:6), where
- for each structured specification encountered in the static semantics of the library, the verification semantics (Sect. 4.7) for the structured specification succeeds, resulting in a development graph $(\mathcal{S}, Th)$, and moreover $\mathcal{S} \vdash Th$ according to the proof rules for development graphs (Sect. 4.4),
- for each architectural specification encountered in the static semantics of the library, the extended static semantics (Sect. III:5.6) for the architectural specifications succeeds, the induced amalgamability conditions can be discharged[1], and the induced proof obligations (which are between structured specifications) collected by the architectural proof calculus (Sect. 5.3) can be discharged using the calculus of development graphs (Sect. 4.4).

For tool purposes, it is interesting to compute a single development graph and set of proof obligations from a library, such that the library is correct if the static semantics succeeds and the proof obligations can be discharged with the calculus for development graphs given in Sect. 4.4. This goal can be achieved by introducing a uniform format for a verification semantics for `LIB-ITEM`s, extending the one for structured specifications given in Sect. 4.7. We sketch below how this could be done, using the material of the previous

[1] This can be done via enriched Casl or the so-called cell calculus, see Sect. III:5.6.

chapters. Based on this, the verification semantics for LIB-DEFNs is easy. The verification semantics collects all the proof obligations that arise in a library.

The verification semantics for libraries will be based on verification global environment:

An verification global environment

$\Gamma_s, (\mathcal{S}, Th) = (\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s), (\mathcal{S}, Th)$ consists of a development graph $(\mathcal{S}, Th)$ and finite functions from names to the *verification* denotations of generic specifications, views, architectural specifications and unit specifications (cf. Chap. III:6):

- $\mathcal{G}_s : \mathit{SpecName} \overset{\text{fin}}{\to} \mathit{VerGenSig}$
- $\mathcal{V}_s : \mathit{ViewName} \overset{\text{fin}}{\to} \mathit{VerViewSig}$
- $\mathcal{A}_s : \mathit{ArchSpecName} \overset{\text{fin}}{\to} \mathit{VerArchSig}$
- $\mathcal{T}_s : \mathit{UnitSpecName} \overset{\text{fin}}{\to} \mathit{VerUnitSig}$

The domains *VerGenSig* and *VerViewSig* have been defined in Sect. 4.7, as well as the context requirements on the corresponding parts of the global environment. We now introduce the semantic domains *VerArchSig* and *VerUnitSig*. They are basically obtained by taking their non-verification counterparts from Chap. III:5 and replacing signatures by development graph nodes. This means that all the introduced concepts have a meaning only relative to a development graph, and this is the development graph from the verification global environment. The requirements from Chap. III:5 are imposed here as well, with the signatures being those obtained from the nodes by looking up their associated signatures in the development graph.

$$\begin{array}{rl}
(N_1, \ldots, N_n) & \\
\text{or } \overline{N} \in & \mathit{VerCompSig} = \mathit{Node}^{+} \\
N_1, \ldots, N_n {\to} N & \\
\text{or } \overline{N} {\to} N \in & \mathit{VerParUnitSig} \subseteq \mathit{VerCompSig} \times \mathit{Node} \\
U\Sigma \in & \mathit{VerUnitSig} = \mathit{VerParUnitSig} \cup \mathit{Node} \\
(N^I, U\Sigma) \in & \mathit{VerImpUnitSig} \subseteq \mathit{Node} \times \mathit{VerParUnitSig} \\
C_s \in & \mathit{VerUnitCtx} = \mathit{UnitName} \overset{\text{fin}}{\to} (\mathit{VerImpUnitSig} \cup \mathit{Node}) \\
(C_s, U\Sigma) \text{ or } A\Sigma \in & \mathit{VerArchSig} = \mathit{VerUnitCtx} \times \mathit{VerUnitSig}
\end{array}$$

$$\boxed{\Gamma_s, (\mathcal{S}, Th) \vdash \texttt{LIB-ITEM} \ggg \Gamma'_s, (\mathcal{S}', Th')}$$

$\Gamma'_s, (\mathcal{S}', Th')$ is a verification global environment extending $\Gamma_s, (\mathcal{S}, Th)$.

The rules for SPEC-DEFN and VIEW-DEFN have been spelled out in Sect. 4.7. The rules from UNIT-SPEC-DEFN can be obtained by a straightforward adaption of the static semantic rules for unit specifications in Sect. III:5.4. The rules for ARCH-SPEC-DEFN are obtained from the rules of the extended static semantics for architectural specifications (Sect. III:5.6) and the architectural proof calculus (Sect. 5.3) in the following way. The rules follow those from the extended static semantics, but with verification contexts instead of diagrams, and with resulting specifications for unit expressions, as in the architectural proof calculus. Moreover, signatures have to be replaced by development graph nodes, as in the transition from the static semantics to the verification semantics for structured specifications. Any assumptions involving the $\rightsquigarrow$ relation between structured specifications occurring in the rules of Sect. 5.3 are replaced by theorem links between the corresponding development graph nodes.

$$\boxed{\vdash \texttt{LIB-DEFN} \ggg \Gamma_s, (\mathcal{S}, Th)}$$

$\Gamma_s, (\mathcal{S}, Th)$ is a verification global environment.

Rules very similar to those in the semantics of libraries (Chap. III:6).

Once this program is carried out, we arrive at the following

Theorem 6.1. *A library is well-formed according to the static and model semantics if the verification semantics succeeds and all the theorem links in the delivered development graph can be discharged. (The only if direction does not hold due to the various sources of incompleteness mentioned in Chap. 1.*

From structured calculus:

This verification semantics can be shifted to the level of CASL libraries in the same way as the ordinary static (and model) semantics. Just note that a new library starts with an empty global environment. The empty verification global environment consists of four empty maps and a development graph consisting just of one node (called $\emptyset$) with signature $\emptyset$ and a set of axioms $\emptyset$.

Theorem 6.2. *If*

$$\vdash \texttt{LIB-DEFN} \rhd (LN, \Gamma_s),$$

then there is some $\Gamma'_s, (\mathcal{S}, Th)$ with $strip(\Gamma'_s) = \Gamma_s$ and

$$\vdash \texttt{LIB-DEFN} \ggg (LN, \Gamma'_s, (\mathcal{S}, Th)),$$

and vice versa, if

$$\vdash \texttt{LIB-DEFN} \ggg (LN, \Gamma'_s, (\mathcal{S}, Th)),$$

then

$$\vdash \texttt{LIB-DEFN} \rhd (LN, strip(\Gamma'_s))$$

Moreover, in this case, the following are equivalent

1. *there is a model global environment Γ_m with $\vdash \texttt{LIB-DEFN} \Rightarrow \Gamma_m$.*
2. *$\mathcal{S} \models Th$.*

Furthermore, if these two equivalent conditions hold, then Γ'_s is compatible with Γ_m.

Part V

Casl Libraries

Markus Roggenbach
Till Mossakowski
Lutz Schröder

P.D. Mosses (Ed.): CASL Reference Manual, LNCS 2960, pp. 361–466, 2004.

1

Introduction

This part of the Casl reference manual describes a library of elementary specifications called the Basic Datatypes. This library has been developed with two main purposes in mind: on the one hand, it provides the user with a handy set of off-the-shelf specifications to be used as building blocks in the same way as library functions in a programming language, thus avoiding continuous reinvention of the wheel. On the other hand, it serves as a large reservoir of example specifications that illustrate both the use of Casl at the level of basic and structured specifications. The specification methodology behind the Basic Datatypes is described in [57].

The name Basic *Datatypes* is actually slightly misleading in that there are both monomorphic specifications of typical datatypes *and* loose specifications that express properties e.g. of an algebraic or order theoretic nature. The first type of specification includes simple datatypes like numbers and characters as well as structured datatypes (typically involving type parameters) such as lists, sets, arrays, or matrices. The second type of specification is oriented more closely towards traditional mathematical concepts; e.g. there are specifications of monoids and rings, as well as equivalence relations or partial orders. The library is structured partly along precisely these lines; an overview of the sublibraries is given in Section 1.1.

In the design of a library of basic specifications, there is a certain amount of tension between the contradicting goals of

- keeping specifications simple and readable also for novice users, and
- making them economical, concise, and amenable for tool support.

This concerns in particular the degree of structuring, with parametrized specifications being most prominent as on the one hand increasing elegance and reusability and on the other hand placing on the reader the burden of looking up imported specifications and keeping track of signature translations. With the exception of the library of numbers, the libraries exhibit a certain bias towards more extensive use of structuring operations. Several measures have been undertaken to enhance readability of the specifications, one of them being the facility to have the signatures for the specifications in a library explicitly listed by the Casl tools.

The specifications make use of a set of annotations concerning semantics and operator precedences; moreover, we use the Casl syntax for literals. The details of these annotations and syntax extensions are explained in Chap. II:5 of the Casl Language Syntax.

The material is organized as follows. After the above-mentioned descriptions of the component libraries (Section 1.1), the actual content of the libraries is presented in Chapters 2 through 11. Chap. 12 contains graphical representations of the dependencies between the specifications. Moreover, there is an index of all library and specification names at the end of the book.

Acknowledgement

We would like to thank the participants of various CoFI meetings for their valuable feedback during the development of these libraries. In particular, we wish to thank Hubert Baumeister, Ulrich Berger, Bernd Krieg-Brückner, Michel Bidoit, Hartmut Ehrig, Magne Haveraaen, Adis Hodzic, Hans-Jörg Kreowski, Christoph Lüth, Stephan Merz, Christoph Schmitz, Giuseppe Scollo, and John Tucker for refereeing the final draft of the libraries. Furthermore, special thanks to Klaus Lüttich for implementing the automatic translation from Casl to pretty printed LaTeX, as well as optimizing the dependency graphs. Of course, any ambiguities and errors that remain are solely our responsibility.

1.1 A Short Overview of the Specified Datatypes

The libraries of basic datatypes have been successfully parsed and statically checked with the Bremen Casl tool set (CATS), as well as with the Heterogeneous tool set (Hets). Both tools as well as an ASCII-format version of libraries of basic datatypes are available on the CD-ROM coming with this volume. The latest versions always can be obtained at

`http://www.cofi.info/Tools`

We recommend to use the Hets tool in order to obtain a graphical overview over the specifications in the libraries and also to inspect their signatures. A quick introduction to Hets can be found at the above URL and also in Chap. 10 of the Casl User Manual [5].

The collection of basic datatypes presented here consists of the following libraries:

- Numbers
- RelationsAndOrders
- Algebra_I
- SimpleDatatypes

- StructuredDatatypes
- Graphs
- Algebra_II
- LinearAlgebra_I
- LinearAlgebra_II
- MachineNumbers

each of which is described in detail in the following paragraphs. The graph of dependencies among the libraries is shown in Fig. 1.1.

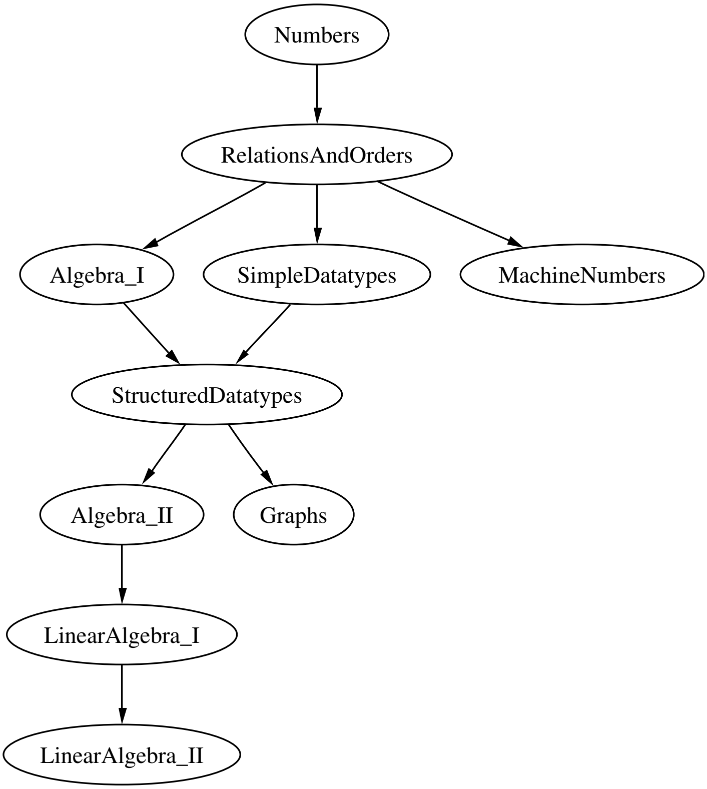

Fig. 1.1. Dependency graph of the libraries of basic datatypes.

1.2 The Library Basic/Numbers

This library, cf. Chap. 2, provides monomorphic specifications of natural numbers, integers and rational numbers, as well as rational numbers with syntactic constructs for decimal fractions.

To a certain extent, the real numbers can also be addressed in Casl, although a full specification is difficult due to the fact that completeness is a higher-order axiom. In [59], a weak theory of the real numbers is specified and compared to other approaches possible in a similar setting. But as none of these approaches leads to a 'basic off-the-shelf specification', we have refrained from including the real numbers in the library of Basic Datatypes.

In the specification Nat, the natural numbers are specified as a free type, thus ensuring that all natural numbers can be constructed from 0 and the successor operation *suc*, and that all terms formed from these two operations are distinct. Consequently, all predicates and operations over the sort Nat of natural numbers are defined by recursion over the two constructors.

In addition to the representation in terms of 0 and *suc*, the usual decimal representation of naturals is introduced in order to provide a more convenient syntax. To this end, the parsing annotation %**number** __@@__ is declared at the beginning of the library; this means that any sequence of digits is to be read as if the function __@@__ were placed in between. The semantics of the function __@@__ is then determined by the axiom %(*decimal_def*)%.

Note that the names for the *partial operations* subtraction __−?__ and division __/?__ include a question mark. This is to avoid overloading with the *total operations* __ − __ on integers and __/__ on rationals, which would lead to inconsistencies as both these specifications import the specification of natural numbers.

The introduction of the subsort Pos, consisting of the positive integers, gives rise to certain new operations, e.g.

$$\_\_ * \_\_ : Pos \times Pos \rightarrow Pos,$$

whose semantics is completely determined by overloading.

The specification Int of integers is built on top of the specification of naturals: integers are defined as equivalence classes of pairs of naturals written as differences, where %(*equality_Int*)% determines the equivalence relation on these pairs. The sort Nat is then declared to be a subsort of Int. Finally, the axiom %(*Nat2Int_embedding*)% characterizes the embedding of naturals into integers.

The definition of the predicates and operations on integers usually covers the whole domain of integers; hence, concerning consistency, one has to show that they do not contradict the axioms for naturals, to which the relevant operations are related by overloading.

Besides the division operator __/?__, the specification Int also provides the function pairs *div*/*mod* and *quot*/*rem*, respectively, as constructs for division:

- The functions *div* and *mod* are motivated by the residue class ring $\mathbb{Z}_m$, $m \in \mathbb{Z}\backslash\{0\}$, where the residue classes are represented by elements of $\{0, 1, \ldots, m-1\}$. In this notation, the operator *mod* computes the residue class $n\ mod\ m$ in $\mathbb{Z}_m$ of an element $n \in \mathbb{Z}$. The division operator *div* is related to *mod* by:

$$\forall n \in \mathbb{Z}, m \in \mathbb{Z}\backslash\{0\} :\ n = (n\ div\ m) * m + (n\ mod\ m)$$

This equation is solved by putting $n\ div\ m := \lfloor n/m \rfloor$, where $\lfloor\_\_\rfloor$ denotes the largest integer lower bound. Thus one obtains the following results:

$5\ div\ \ 3 = \ 1$	$5\ mod\ \ 3 = 2$
$-5\ div\ \ 3 = -2$	$-5\ mod\ \ 3 = 1$
$5\ div\ -3 = -1$	$5\ mod\ -3 = 2$
$-5\ div\ -3 = \ 2$	$-5\ mod\ -3 = 1$

- Another way to deal with division is to require of the remainder operator, now called *rem*, that

$$|n\ rem\ m| = |n|\ rem\ |m|$$

for all integers n, m (which doesn't hold for the operator *mod*). To this end, the representative has to be chosen depending on the sign of n: Choose the representative from the set $\{0, 1, \dots, m-1\}$ if $n \geq 0$, and from the set $\{0, -1, \dots, -(m-1)\}$ if $n < 0$. The division function *quot* is then determined by requiring

$$\forall n \in \mathbb{Z}, m \in \mathbb{Z}\backslash\{0\} :\ n = (n\ quot\ m) * m + (n\ rem\ m).$$

For this division operator, we have

$$|n\ quot\ m| = |n|\ quot\ |m|.$$

Some example values:

$5\ quot\ \ 3 = \ 1$	$5\ rem\ \ 3 = \ 2$
$-5\ quot\ \ 3 = -1$	$-5\ rem\ \ 3 = -2$
$5\ quot\ -3 = -1$	$5\ rem\ -3 = \ 2$
$-5\ quot\ -3 = \ 1$	$-5\ rem\ -3 = -2$

There is no need to define a new parsing annotation for decimal representation of integers, as the one for naturals carries over. Positive integers are first parsed as a natural and then subjected to the implicit embedding from naturals to integers. The same holds for negative integers, where just the unary minus has to be applied after the embedding.

The specification RAT of rational numbers follows the same scheme as the specification of integers discussed above. This time, the specification INT is imported. The rationals are then defined as equivalence classes of pairs consisting of an integer and a positive number written as quotients. %(*equality_Rat*)% determines the equivalence relation on these pairs. The sort *Int* is then declared to be a subsort of *Rat*. Finally, the axiom %(*Int2Rat_embedding*)%

characterizes the embedding. Note that thanks to Casl subsorting, the declaration of the operation

$$\_\_/\_\_ : \mathit{Rat} \times \mathit{Rat} \rightarrow ? \, \mathit{Rat}$$

allows writing rationals also as pairs x/y of arbitrary integers x and $y \neq 0$.

Again, the definition of the predicates and operations on rationals usually covers the whole domain of rationals, i.e. concerning consistency one has to show that they do not contradict to the axioms for naturals and integers, to which they are related by overloading.

The specification DecimalFraction extends the rationals by syntactic sugar that allows writing rationals in the form of decimal fractions like $11.02E{-}3$ or -0.2. To this end, the library Basic/Numbers includes the parsing annotation **%floating**. Here, the function $\_\_ ::: \_\_$ evaluates the decimal point, while the function $\_\_E\_\_$ is used for the exponentiation 'E'. The specification DecimalFraction then provides axioms for both of these. Note how the different parsing annotations cooperate with each other. The numbers to the left and right of the decimal point are parsed as naturals and then turned into a rational by evaluating the function replacing the decimal point. The function replacing the 'E' is applied to the result.

The main order-theoretic and algebraic properties of the numbers specified in this library are expressed in terms of Casl views. The library Basic/RelationsAndOrders, cf. Chap. 3, includes views that state that the specified naturals, integers, and rationals are totally ordered. In the library Basic/Algebra_I, cf. Chap. 4, views can be found expressing e.g. that the naturals with $\_\_ + \_\_$ and 0 or $\_\_ * \_\_$ and 1, respectively, satisfy the axioms of a commutative monoid, that the integers are an integral domain, and that the rationals form a field.

1.3 The Library Basic/RelationsAndOrders

This library, cf. Chap. 3, provides specifications for various types of relations. Among the specified structures are reflexive, symmetric, and transitive relations and equivalence relations, partial and total orders, as well as Boolean algebras. For some of these, the library offers extended versions.

Datatypes involving completeness properties, like directed complete partial orders or complete lattices, are omitted here. Their specification would require higher-order axioms that are not expressible in Casl. (It would, however, be possible to specify ω-complete partial orders.)

The specifications concerning the basic structures are naked textbook-style definitions. These are then extended by adding typical predicates and operations from the respective mathematical theories. Such an extended version ExtX takes the specification X as a parameter. In order to avoid excessive verbosity in cases where instantiations do not involve renaming, pre-instantiated parameterless specifications RichX are included; in particular, RichX specifies the same model class as ExtX. For instance, the specification TotalOrder adds an axiom stating comparability of all elements to the specification of partial orders. Its extended version ExtTotalOrder then provides (among others) the additional operations *min* and *max*, which are defined in terms of the $\_\_ \leq \_\_$ predicate stemming from the specification TotalOrder in the parameter. RichTotalOrder is identical to ExtTotalOrder, except that it does not have a parameter.

In the extended versions, concepts are often added independently of each other. This is reflected by the use of the Casl union operator '**and**' instead of the extension operator '**then**'. For example, adding the operators *inf* and *sup* to partial orders is independent of the definition of the predicate $\_\_ < \_\_$ on top of $\_\_ \leq \_\_$.

The library concludes with a collection of views. These state that the numbers specified in the library Basic/Numbers are totally ordered, and that a Boolean algebra carries also the structure of a partial order.

1.4 The Library Basic/Algebra_I

In this library, cf. Chap.4, specifications of basic algebraic structures are collected. These specifications, like those in Basic/RelationsAndOrders, are usually split into two parts, one that provides the necessary signature and axioms in as simple a way as possible, and a second, parametrized part labelled Ext... which contains derived operations and predicates. Typical examples are power operations with natural or integer exponents, the inverse operation for groups, and predicates concerning divisibility and invertibility in rings. Moreover, the extended specifications contain theorems, intended to be derivable from the axioms, in the shape of formulas annotated as implied. As in the library RelationsAndOrders, the extended specifications are pre-instantiated under the name Rich...; these specifications may be found near the end of the library. Any views into the extended specifications are repeated for the pre-instantiated forms, since views cannot be defined in terms of other named views in Casl.

The hierarchy of algebraic structures presented begins with monoids, continuing via groups and Abelian groups to rings, integral domains, and fields. At the end of the library, a number of views are given that subsume concepts from the numbers library under the appropriate algebraic concepts, e.g. a view IntegralDomain_in_Int which states that the integers with their usual addition and multiplication form an integral domain. By contrast, several views that express theorems about the algebraic concepts introduced are given directly after the concept they concern. E.g., there is a view PreOrder_in_ExtCRing stating that the elements of a commutative ring are pre-ordered by the divisibility relation, located adjacently to the (extended) specification of commutative rings.

A feature that requires a word of explanation is the fact that the specification of fields is actually split into *three* parts, namely, ConstructField, Field, and ExtField. The reason for this is that an extra sort of nonzero elements is needed to specify the multiplicative group structure of a field; since this sort is not regarded as a part of the basic signature of a field (this signature should be identical to that of a ring, i.e. consist of one sort and two unary and two binary operations), it is introduced in ConstructField, then hidden in Field, and finally reintroduced in ExtField by instantiating ExtCommutativeRing with ConstructField as argument.

The specifications ExtRing and ExtCommutativeRing contain, among other things, elements of divisibility theory in rings. In ExtRing, a unit predicate *isUnit* and an irreducibility predicate *isIrred* are introduced, along with subsorts *NonZero*[*Elem*], *RUnit*[*Elem*], and *Irred*[*Elem*] representing nonzero, unit, and irreducible elements. Recall that an element of a ring is a unit if it has a multiplicative inverse, and that a non-unit element is irreducible if it cannot be decomposed into two non-unit elements. The identity, multiplication, and additive inverse operations are given additional profiles stating that they restrict to unit elements; identity and multiplication indeed make the set of unit elements into a group, a fact which is expressed by the view Group_in_ExtRing. The specification ExtCommutativeRing additionally provides a nullary predicate *hasNoZeroDivisors*, detecting whether or not there are divisors of zero, and binary divisibility and associatedness relations; recall that two elements of a commutative ring are called associated if they differ only by an invertible factor. The fact that two elements are associated iff they are mutually divisible is stated as a theorem. Moreover, there are two views PreOrder_in_ExtCRing (see above) and EqRel_in_ExtCRing, with the latter stating that associatedness is an equivalence relation.

1.5 The Library Basic/SimpleDatatypes

This library, cf. Chap.5, provides unstructured datatypes like Booleans and characters.

The Booleans are specified in Boolean, are shown to be a Boolean algebra (view BooleanAlgebra_in_Boolean) and then enriched with the usual Boolean algebra operations (using the specification ExtBooleanAlgebra). The characters are specified in Char. They are defined to be the subset 0 . . . 255 of the natural numbers, and then constants for different representations (ASCII, decimal, octal, hexadecimal) are introduced.

1.6 The Library Basic/StructuredDatatypes

This library, cf. Chap. 6, provides specifications that formalize structuring concepts of data as used e.g. for the design of algorithms or within programming languages. Its main focus is data structures like (finite) sets, lists, strings, (finite) maps, (finite) bags, arrays, and various kinds of trees. But it also covers some elementary constructions like the encapsulation of data within a 'maybe'-type or arranging data as pairs. Common to all these concepts is that they are generic. Consequently, all specifications of this library are parametrized. Furthermore, in all specifications of this library the body of the specifications monomorphically extends the given parameters and imports.

Arbitrary sets, maps and bags, or streams are omitted. Their monomorphic specification would require higher order axioms not available in Casl. Non-monomorphic first-order specifications of these types are not included, as the answer to the question which operations to provide and which non-standard models to accept would depend too much on the particular context.

Finite sets, finite maps and finite bags are specified in terms of observers: given a generated sort, an operation or predicate is introduced in order to define equality on this sort. Concerning finite sets, equality on the sort *Set*[*Elem*] is characterized using the predicate $\_\_\epsilon\_\_$, see the specification GenerateSet. Finite maps, i.e. elements of the sort *Map*[*S*, *T*], are considered to be identical if their evaluation under the operation *eval* yields the same result, cf. the specification GenerateMap. In the specification GenerateBag, those elements of sort *Bag*[*Elem*] are identified that show the same frequency (observed by the operation *freq*) for all entries.

The specification PowerSet works solely with Casl subsorting and overloading; no defining axiom is needed. This is achieved by the subsort definitions for *PowerSet*[*X*] and *Elem*[*X*]. Besides determining the elements of the respective sorts, these definitions also induce the subsort relations $PowerSet[X] < Set[Elem]$ and $Elem[X] < Elem$. This type system ensures that the newly introduced predicates and operations are in overloading relation with the identically named predicates and operations of the specification Set[Elem], and hence are just restrictions of those.

Finite lists are specified in terms of a free datatype. In the specification GenerateList, lists are built up from the empty list by prefixing. The reverse construction, i.e. describing lists as a type that consists either of the empty list or a list followed by an element, is added in the specification List as an implied consequence. That is, the specification List makes both approaches and their corresponding induction principles available. The predicates and operators, however, are all defined using the first approach. The parsing annotation **%list** at the beginning of the library allows to write lists in the more convenient syntax '$[x_1, \ldots, x_n]$' besides '$x_1 :: \ldots :: x_n :: []$' as provided by the constructors.

The specification Array includes the condition $min \leq max$ as an axiom in its first parameter. This ensures a non-empty index set. Arrays are defined as finite maps from the sort *Index* to the sort *Elem*, where the typical array operations evaluation and assignment are introduced in terms of finite map operations. Finally, revealing the essential signature elements yields the desired datatype.

The library concludes with several specifications concerning trees. There are specifications covering binary trees (BinTree, BinTree2), trees with a possibly-different branching at each node (NTree, NTree2), and k-branching trees (KTree, KTree2). Each of these branching structures can be equipped with data in different ways: either all nodes of a tree carry data (as it is the case in BinTree, NTree, and KTree), or just the leaves of a tree have a data entry (as in BinTree2, NTree, and KTree). Note the slightly more complex type systems needed in the latter case.

In GenerateNTree and GenerateNTree2, it is necessary to specify both the datatype of trees as well as the datatype of lists modeling the branching together in *one* free type construct in order to avoid unintended models. To provide the usual operations on lists, the specification List is imported later in NTree and NTree2, resp. Note that NTree2 does not include empty branching, while in NTree an empty list of branches characterizes a leaf node. The k-branching trees are then introduced as subsorts of the NTrees.

The abstract properties of the specified concepts are expressed in terms of views, in this library mostly to be found directly after the specification. For example, finite sets carry the structure of a partial order, finite power sets are Boolean algebras, and lists form a monoid.

1.7 The Library Basic/Graphs

This library, cf. Chap. 7, provides a specification of directed graphs, as well as operations and predicates on graphs, like paths, transitive closure, connectedness, n-colorability and planarity. These capture standard notions from the literature, see e.g. [18].

The specification Graph constructs directed graphs inductively by successively adding nodes and edges to an empty graph, using a generated type and an explicit characterization of equality. Due to the inductive definition, only finite graphs are covered. However, the advantage of this approach is that graphs are first-class objects, i.e. members of algebras (rather than algebras themselves, as in other approaches).

The specification is parametrized over two sorts, *NodeId* and *EdgeId*. These provide the (typically infinite) vocabularies for node and edge identifiers. These must uniquely identify nodes and edges. Since we allow only finite graphs, in a given graph, only finitely many identifiers of the vocabularies are actually used. If multiple edges with the same label are needed, *EdgeId* should be chosen as isomorphic to a product (e.g. $\mathit{Label} \times \mathit{Int}$).

The operation *addNode* is total – if a node that is already present in the graph is added twice, nothing happens. By contrast, *addEdge* is partial: this is because an edge always has to be added together with its source and target node, and adding an edge twice (with possibly different source and target nodes) is prohibited.

The specification provides predicates to check whether a node or an edge is in the graph, as well as a predicate checking whether an edge goes between two particular nodes. The operations *source* and *target* are partial, because they act on the global vocabulary of edge identifiers, while a given graph usually contains only some of these. Note that *source* and *target* are undefined for the empty graph, and this undefinedness is inherited via the strong equations defining *source*, which yields that *source* and *target* are also undefined if the edge identifier given in the first argument is not present in the graph given by the second argument.

The specification RichGraph provides further operations (for removing nodes and edges) as well as a bunch of graph-theoretic predicates (*loopFree*, *simple*, *subgraphOf*, *complete*, *cliqueOf*, *maxCliqueOf*). Note that removing a node from a graph also entails removing all edges having this node as source or target node.

The specification GraphToSet provides a means to change the graph representation by mapping the inductively generated graphs to the more common mathematical definition: a graph consists of a set of nodes, a set of edges, and source and target functions going from edges to nodes.

A subsort of symmetric graphs is introduced in SymmetricGraph. Since these can be seen as a representation of undirected graphs, we also introduce restrictions of the graph operations to this subsort. The specification SymmetricClosure defines the symmetric closure of a graph.

In the specification Paths the paths in a graph as well as the transitive closure of a graph are specified. PathGraphs are defined as graphs over *lists* of edge identifiers. The transitive closure of a graph is then the minimal transitive super-PathGraph of that graph. A path in a graph is defined to be an edge in the transitive closure.

Further concepts are defined in a straightforward manner on top of these basic notions. Trees are acyclic graphs with a root node such that each node is reachable via a unique path from the root. Connectedness and acyclicity can be elegantly expressed using (symmetric) transitive closure. A spanning tree is a tree subgraph that has the same nodes as the graph.

Finally, some notions concerning undirected graphs (represented as symmetric graphs) are defined. A symmetric graph is said to have a cycle only if the cycle is non-trivial, i.e. does not exploit the fact that for each edge there also is an edge in the opposite direction. Symmetric trees are connected graphs that are acyclic in the symmetric sense.

The specification GraphColorability defines n-colorable and bipartite graphs.

Assuming a weight function on edge identifiers, the specification ShortestPaths provides a loose specification of shortest paths in a graph. For a source and a target node, the shortest path function is only defined if there is at least one path from the source to the target.

Since also homomorphisms between graphs over *different* node and edge vocabularies are interesting, the specification GraphHomomorphism is parametrized over two pairs of node and edge vocabulary, and for each pair, the specification Graph is instantiated (and the resulting sort *Graph* is renamed differently for the two instantiations).

Pre-homomorphisms just collect the data of graph homomorphisms, basically consisting of source and target graph and of the finite maps between nodes and edges. The subsort *Hom* consists of those pre-homomorphisms that actually satisfy the homomorphism condition.

A minor of a graph is something that can be homomorphically mapped to the transitive closure. This concept is formalized in the specification Minor.

The library continues with the specifications of specific graphs: K5 provides the complete graph over five nodes. Note that the K5 is captured by the constant $k5$ written lower case, since constants generally are written lower case in the library of Basic Datatypes (cf. [57]). K3_3 introduces the graph consisting of two copies of three nodes, such that two nodes are linked by an edge iff they stem from different copies. Then, Planar defines planar graphs using the Kuratowski characterization: K5 and K3_3 must not occur as minors.

Finally, the specification NonUniqueEdgesGraph provides graphs with edge labels that need not be unique (note that with ordinary graphs, each edge may be inserted only with one pair of source and target nodes). The trick is to turn edges labels into unique edges by coupling them with the source and target node.

1.8 The Library Basic/Algebra_II

This library, cf. Chap. 8, contains slightly more advanced algebraic concepts, in particular

- monoid and group actions on a space,
- ring theoretic notions such as euclidian and factorial rings,
- polynomials, and
- two views exhibiting the datatypes of lists and bags as free monoids and free commutative monoids, respectively.

At several points, use is made of structured datatypes. E.g., factorial rings require bags for the specification of factorizations (e.g., factorizing an integer amounts to stating how often it is divisible by any given prime), and polynomials are represented as lists of coefficients. For monoid actions, the use of parametrization has been restricted to the involved monoid, rather than parametrizing over the space as well, in order to keep the specifications readable.

Defining the degree function for polynomials requires an extension of the integers by $-\infty$ (since by the usual convention, $deg(0) = -\infty$); the corresponding specification IntInfinity is provided here as well. Polynomials (in one variable) can, of course, be specified very concisely as the free algebra on one generator using CASL's **free**-construct; this is stated in the library Basic/LinearAlgebra_II by means of a view.

For the same reason as for fields, the specification of factorial rings is split into three parts (cf. 1.4); the machinery required to arrive at the somewhat involved statement in CONSTRUCTFACTORIALRING that each element of a factorial ring has an essentially unique factorization into irreducible elements, where 'essentially unique' means unique up to associatedness of the factors, is temporarily discarded in FACTORIALRING. Several views are provided, stating e.g. that integers and polynomials, respectively, form euclidian rings and that euclidian rings are factorial.

In more detail, euclidian rings are defined as admitting division with remainder, where division strictly decreases a measure function *delta*. In the representation of polynomials as lists of coefficients, the head of the list represents the constant coefficient (i.e. that of X^0). In order to obtain a unique representation, lists that end with a 0 are excluded; e.g., 1 is represented by [1], and 0 is represented by []. This choice of representatives is reflected in a special constructor __ ::: __ which behaves like the usual list constructor except in cases where this would lead to a list with leading coefficient 0. Note that $a ::: p = a + p * X$, where a is an element of the underlying ring and p is a polynomial. Addition and multiplication of polynomials are defined by recursion over this special constructor. The view which identifies polynomial rings as being euclidian requires casting the (normally $\mathbb{Z}_\infty$-valued) degree function to natural number values in order to match the measure function *delta* in the specification of euclidian rings; the downcast is undefined for the zero polynomial, which is explicitly allowed for euclidian rings.

The extended specifications for monoid and group actions, respectively, mention a binary orbit relation *connected* on the underlying space, where x is connected to y iff it is taken to y by some element of the monoid. This relation is in general a pre-order, and an equivalence relation for group actions (the arising equivalence classes are usually called orbits), which facts are expressed by the views PREORDER_IN_EXTMONOIDACTION and EQREL_IN_EXTGROUPACTION, respectively.

1.9 The Library Basic/LinearAlgebra_I

This library, cf. Chap. 9, provides elementary concepts from linear algebra such as vector spaces and bases. Moreover, there are 'computational' specifications for tuples of vectors (i.e. finite powers of vector spaces), column vectors, and matrices, equipped with the usual operations such as scalar product, matrix multiplication, and determinant. These are related to the abstract notions of vector space etc. via suitable views.

Using predefined concepts from the algebra library, the definition of vector spaces can be kept very concise: a vector space is an action of the multiplicative monoid of a field on an Abelian group, subject to two distributivity axioms. The specification of a base of a vector space requires the introduction of a technical sort *BaseLC* for linear combinations of base elements. This sort is, as can by now be considered established practice, introduced in ConstructVSWithBase and hidden in VSWithBase. The advantage of having to specify only a base, but not a sort of linear combinations, is illustrated on several occasions where views are provided from VSWithBase to e.g. matrices or vectors.

More precisely, the concept of a linear combination is introduced in the auxiliary specification VectorSpaceLC; formally, a linear combination is a finite map from vectors to scalars including zero. An evaluation predicate *eval* for linear combinations and a test for zero linear combinations are defined recursively.

The specifications of tuples, vectors and matrices are comparatively lengthy due to the fact that concrete functions on them need to be defined recursively, occasionally using auxiliary functions that are later hidden. In fact, the number of auxiliary signature items in the specification of matrices is so large that they are more conveniently hidden via a local specification, rather than by an explicit hiding statement.

For example, n-tuples of vectors are defined as arrays, indexed from 1 to n; operations on them are defined in terms of the array access operation $\_\_!\_\_$ (cf. Section 1.6). There is a sum operation for vector tuples; the recursive definition of this function requires an auxiliary function *auxsum* which adds all elements up to a given index. Similarly, the definition of the scalar product $\langle\_\_||\_\_\rangle$ requires auxiliary functions *auxmult* and *auxprod*; the former is component-wise multiplication of vectors, and the second plays a role analogous to that of *auxsum* in the definition of a function *prod* that multiplies all scalars in a vector. Recall that the scalar product is defined by

$$\langle (x_1,\dots,x_n)||(y_1,\dots,y_n)\rangle = \sum_{i=1}^{n} x_i y_i.$$

The datatype of matrices is defined via tuples of vectors, i.e. in the end via two-dimensional arrays, with elements accessed by applying the array access function ! twice. A transpose operation and elementary matrices are defined via the access operation; recall that a matrix is elementary iff exactly one of its entries is 1 and all others are 0. The determinant of a matrix is defined by the Leibniz formula

$$\det(a_{ij}) = \sum \epsilon(\pi) a_{i\pi(i)},$$

where π ranges over all permutations of the set $\{1,\dots,n\}$ and $\epsilon(\pi)$ is the sign of π. This requires a separate specification of the n-th symmetric group that includes the sign function. A permutation is represented by the array containing its graph; the sign function is specified as the unique nontrivial homomorphism from the symmetric group into the multiplicative group $\{-1,1\}$. Moreover, the symmetric group is supplied with an enumeration function *perm* defined on the set $\{1,\dots,n!\}$.

At the end of the library, several views are provided that exhibit the sets of vectors and matrices as vector spaces equipped with standard bases. Similarly, there is a view stating that every field is a vector space over itself, with the multiplication of the field as scalar multiplication; this view is located earlier in the library, since it is needed in the specification ConstructVector. Moreover, there is an example for the use of views as 'higher order theorems': under the axiom of choice (which is assumed for the semantics of Casl, see Chap. III:4), every vector space has a base; this is expressed by means of the view VSWithBase_in_VectorSpace.

1.10 The Library Basic/LinearAlgebra_II

In this library, cf. Chap. 10, we present one advanced notion omitted from the elementary linear algebra library, namely that of algebras over a field, i.e. vector spaces equipped with a compatible ring structure. Notably, the extended specification of k-algebras contains an evaluation operation for polynomials over k, defined recursively using the special list constructor __ ::: __ for polynomials that avoids 0 as leading coefficient. Moreover, we have included two views stating that a vector space with a given base is free over that base, and that the polynomial ring in one variable over a field k is the free k-algebra over a one-element set. Two specifications are introduced expressly for this purpose, namely, FreeVectorSpace and FreeAlgebra. As indicated by the name, these specifications make use of Casl's structured **free**-construct; comparing them with the 'standard' ones gives a good feel for the expressive power of that construct.

1.11 The Library Basic/MachineNumbers

This library of machine numbers, cf. Chap. 11, contains specifications of those subtypes of the naturals and the integers that are used on actual machines.

The specifications CARDINAL and INTEGER provide subtypes of naturals and integers consisting of those numbers that have a binary representation within a given word length. Operations on these data types are partial restrictions of the usual operations on naturals and integers – they are undefined if the word length is exceeded.

The specification TwoComplement provides a 'cyclic' version of bounded integers that corresponds to the common two complement representation of integers used in many programming languages. Operations are total here – the successor of the maximal positive number fitting in the word length is the minimal negative number.

The Ext versions of the specifications add minimum and maximum operations by instantiating ExtTotalOrder.

2

Library Basic/Numbers

```
library BASIC/NUMBERS version 1.0

%authors(M. Roggenbach <csmarkus@swansea.ac.uk>, T. Mossakowski,
              L. Schröder)%
%date : 18 December 2003
%{ This library provides specifications of naturals, integers, and
   rationals. Concerning the rationals, the specification Rat includes
   the datatype proper, while the specification DecimalFraction adds the
   notions needed to represent rationals as decimal fractions. }%
%display(__<=__ %LATEX __≤__)%
%display(__>=__ %LATEX __≥__)%
%prec({__−?__, __−__, __+__} < {__∗__, __/?__, __/__,
      __div__, __mod__, __quot__, __rem__})%
%prec({__∗__, __/?__, __/__, __div__, __mod__, __quot__,
      __rem__} < {__ˆ__})%
%prec({−__} <> {__ˆ__})%
%prec({__E__} < {__:::__})%
%left_assoc(__+__, __∗__, __@@__)%
%number __@@__
%floating __:::__, __E__


spec NAT = %mono
    free type Nat ::= 0 | suc(pre:?Nat)
    preds __≤__, __≥__, __<__, __>__ : Nat × Nat;
          even, odd : Nat
    ops   __! : Nat → Nat;
          __+__, __∗__, __ˆ__, min, max, __−!__ :
             Nat × Nat → Nat;
          __−?__, __/?__, __div__, __mod__ :
             Nat × Nat →? Nat
```

%% Operations to represent natural numbers with digits:

ops $1: Nat = suc(0)$; %(1_def_Nat)%
$2: Nat = suc(1)$; %(2_def_Nat)%
$3: Nat = suc(2)$; %(3_def_Nat)%
$4: Nat = suc(3)$; %(4_def_Nat)%
$5: Nat = suc(4)$; %(5_def_Nat)%
$6: Nat = suc(5)$; %(6_def_Nat)%
$7: Nat = suc(6)$; %(7_def_Nat)%
$8: Nat = suc(7)$; %(8_def_Nat)%
$9: Nat = suc(8)$; %(9_def_Nat)%
$\_\_@@\_\_(m: Nat; n: Nat): Nat = m * suc(9) + n$
%(decimal_def)%

%% implied operation attributes :

ops $\_\_+\_\_ : Nat \times Nat \rightarrow Nat, comm, assoc, unit\ 0$; %**implied**
$\_\_*\_\_ : Nat \times Nat \rightarrow Nat, comm, assoc, unit\ 1$; %**implied**
$min : Nat \times Nat \rightarrow Nat, comm, assoc$; %**implied**
$max : Nat \times Nat \rightarrow Nat, comm, assoc, unit\ 0$ %**implied**

$\forall\ m, n, r, s, t: Nat$

%% axioms concerning predicates

- $0 \leq n$ %(leq_def1_Nat)%
- $\neg\ suc(n) \leq 0$ %(leq_def2_Nat)%
- $suc(m) \leq suc(n) \Leftrightarrow m \leq n$ %(leq_def3_Nat)%
- $m \geq n \Leftrightarrow n \leq m$ %(geq_def_Nat)%
- $m < n \Leftrightarrow m \leq n \wedge \neg\ m = n$ %(less_def_Nat)%
- $m > n \Leftrightarrow n < m$ %(greater_def_Nat)%
- $even(0)$ %(even_0_Nat)%
- $even(suc(m)) \Leftrightarrow odd(m)$ %(even_suc_Nat)%
- $odd(m) \Leftrightarrow \neg\ even(m)$ %(odd_def_Nat)%

%% axioms concerning operations

- $0\ ! = 1$ %(factorial_0)%
- $suc(n)\ ! = suc(n) * n\ !$ %(factorial_suc)%
- $0 + m = m$ %(add_0_Nat)%
- $suc(n) + m = suc(n + m)$ %(add_suc_Nat)%
- $0 * m = 0$ %(mult_0_Nat)%
- $suc(n) * m = n * m + m$ %(mult_suc_Nat)%
- $m\ \hat{}\ 0 = 1$ %(power_0_Nat)%
- $m\ \hat{}\ suc(n) = m * m\ \hat{}\ n$ %(power_suc_Nat)%
- $min(m, n) = m\ when\ m \leq n\ else\ n$ %(min_def_Nat)%
- $max(m, n) = n\ when\ m \leq n\ else\ m$ %(max_def_Nat)%
- $n -!\ m = 0\ if\ m > n$ %(subTotal_def1_Nat)%
- $n -!\ m = n -?\ m\ if\ m \leq n$ %(subTotal_def2_Nat)%
- $def\ m -?\ n \Leftrightarrow m \geq n$ %(sub_dom_Nat)% %**implied**
- $m -?\ n = r \Leftrightarrow m = r + n$ %(sub_def_Nat)%

```
     • def m /? n ⇔ ¬ n = 0 ∧ m mod n = 0
                                            %(divide_dom_Nat)% %implied
     • ¬ def m /? 0                                   %(divide_0_Nat)%
     • (m /? n = r ⇔ m = r ∗ n) if n > 0           %(divide_Pos_Nat)%
     • def m div n ⇔ ¬ n = 0                 %(div_dom_Nat)% %implied
     • m div n = r ⇔ ∃ s: Nat • m = n ∗ r + s ∧ s < n     %(div_Nat)%
     • def m mod n ⇔ ¬ n = 0                 %(mod_dom_Nat)% %implied
     • m mod n = s ⇔ ∃ r: Nat • m = n ∗ r + s ∧ s < n
                                                           %(mod_Nat)%

     %% important laws
     • (r + s) ∗ t = r ∗ t + s ∗ t              %(distr1_Nat)% %implied
     • t ∗ (r + s) = t ∗ r + t ∗ s              %(distr2_Nat)% %implied
then %mono
     sort  Pos = {p: Nat • p > 0}
     ops   1: Pos = suc(0);                          %(1_as_Pos_def)%
           __∗__ : Pos × Pos → Pos;
           __+__ : Pos × Nat → Pos;
           __+__ : Nat × Pos → Pos;
           suc : Nat → Pos
then %implies
     ∀ m, n, r, s: Nat
     • min(m, 0) = 0                                         %(min_0)%
     • m = (m div n) ∗ n + m mod n if ¬ n = 0         %(div_mod_Nat)%
     • m ˆ (r + s) = m ˆ r ∗ m ˆ s                        %(power_Nat)%
end

spec INT = %mono
     NAT
then %mono
     generated type Int ::= __−__(Nat; Nat)
     ∀ a, b, c, d: Nat
     • a − b = c − d ⇔ a + d = c + b                   %(equality_Int)%
     sort  Nat < Int
     ∀ a: Nat
     • a = a − 0                                  %(Nat2Int_embedding)%
then %def
     preds __≤__, __≥__, __<__, __>__ : Int × Int;
           even, odd : Int
     ops   −__, sign : Int → Int;
           abs : Int → Nat;
           __+__, __∗__, __−__, min, max : Int × Int → Int;
           __ˆ__ : Int × Nat → Int;
           __/?__, __div__, __quot__, __rem__ :
              Int × Int →? Int;
```

$\_\_mod\_\_ : Int \times Int \to ?\ Nat$

%% implied operation attributes :

ops $\_\_+\_\_ : Int \times Int \to Int,\ comm,\ assoc,\ unit\ 0;$ **%implied**

$\_\_*\_\_ : Int \times Int \to Int,\ comm,\ assoc,\ unit\ 1;$ **%implied**

$min,\ max : Int \times Int \to Int,\ comm,\ assoc$ **%implied**

$\forall\ m, n, r, s, t: Int;\ a, b, c, d: Nat$

%% axioms concerning predicates

- $\bullet\ m \leq n \Leftrightarrow (n - m \in Nat)$ %(leq_def_Int)%
- $\bullet\ m \geq n \Leftrightarrow n \leq m$ %(geq_def_Int)%
- $\bullet\ m < n \Leftrightarrow m \leq n \wedge \neg\ m = n$ %(less_def_Int)%
- $\bullet\ m > n \Leftrightarrow n < m$ %(greater_def_Int)%
- $\bullet\ even(m) \Leftrightarrow even(abs(m))$ %(even_def_Int)%
- $\bullet\ odd(m) \Leftrightarrow \neg\ even(m)$ %(odd_def_Int)%
- $\bullet\ odd(m) \Leftrightarrow odd(abs(m))$ %(odd_alt_Int)%

%% axioms concerning operations

- $\bullet\ -\ (a - b) = b - a$ %(neg_def_Int)%
- $\bullet\ sign(m) = 0\ when\ m = 0\ else\ 1\ when\ m > 0\ else\ -\ 1$ %(sign_def_Int)%
- $\bullet\ abs(m) = -\ m\ when\ m < 0\ else\ m$ %(abs_def_Int)%
- $\bullet\ (a - b) + (c - d) = (a + c) - (b + d)$ %(add_def_Int)%
- $\bullet\ (a - b) * (c - d) = (a * c + b * d) - (b * c + a * d)$ %(mult_def_Int)%
- $\bullet\ m - n = m + -\ n$ %(sub_def_Int)%
- $\bullet\ min(m, n) = m\ when\ m \leq n\ else\ n$ %(min_def_Int)%
- $\bullet\ max(m, n) = n\ when\ m \leq n\ else\ m$ %(max_def_Int)%
- $\bullet\ (-\ 1)\ \hat{}\ a = 1\ when\ even(a)\ else\ -\ 1$ %(power_neg1_Int)%
- $\bullet\ m\ \hat{}\ a = sign(m)\ \hat{}\ a * abs(m)\ \hat{}\ a\ if\ \neg\ m = -\ 1$ %(power_others_Int)%
- $\bullet\ def\ m\ /?\ n \Leftrightarrow m\ mod\ n = 0$ %(divide_dom2_Int)% **%implied**
- $\bullet\ m\ /?\ n = r \Leftrightarrow \neg\ n = 0 \wedge n * r = m$ %(divide_alt_Int)% **%implied**
- $\bullet\ m\ /?\ n = sign(m) * sign(n) * (abs(m)\ /?\ abs(n))$ %(divide_Int)%
- $\bullet\ def\ m\ div\ n \Leftrightarrow \neg\ n = 0$ %(div_dom_Int)% **%implied**
- $\bullet\ m\ div\ n = r \Leftrightarrow \exists\ a: Nat \bullet m = n * r + a \wedge a < abs(n)$ %(div_Int)%
- $\bullet\ def\ m\ quot\ n \Leftrightarrow \neg\ n = 0$ %(quot_dom_Int)% **%implied**

```
      • (m quot n = r ⇔
          ∃ s: Int • m = n ∗ r + s ∧ 0 ≥ s ∧ s > − abs(n)) if m < 0
                                                          %(quot_neg_Int)%
      • (m quot n = r ⇔
          ∃ s: Int • m = n ∗ r + s ∧ 0 ≤ s ∧ s < abs(n)) if m ≥ 0
                                                       %(quot_nonneg_Int)%
      • def m rem n ⇔ ¬ n = 0               %(rem_dom_Int)% %implied
      • (m rem n = s ⇔
          ∃ r: Int • m = n ∗ r + s ∧ 0 ≥ s ∧ s > − abs(n)) if m < 0
                                                          %(quot_rem_Int)%
      • (m rem n = s ⇔
          ∃ r: Int • m = n ∗ r + s ∧ 0 ≤ s ∧ s < abs(n)) if m ≥ 0
                                                        %(rem_nonneg_Int)%
      • def m mod n ⇔ ¬ n = 0               %(mod_dom_Int)% %implied
      • m mod n = a ⇔ ∃ r: Int • m = n ∗ r + a ∧ a < abs(n)
                                                              %(mod_Int)%

      %% important laws
      • (r + s) ∗ t = r ∗ t + s ∗ t           %(distr1_Int)% %implied
      • t ∗ (r + s) = t ∗ r + t ∗ s           %(distr2_Int)% %implied
then %implies
      ∀ m, n, r: Int; a, b: Nat
      • def a −? b ⇒ a −? b = a − b                %(Int_Nat_sub_compat)%
      • m = sign(m) ∗ abs(m)                            %(abs_decomp_Int)%
      • m mod n = m mod abs(n)                             %(mod_abs_Int)%
      • m = (m div n) ∗ n + m mod n if ¬ n = 0             %(div_mod_Int)%
      • abs(m quot n) = abs(m) quot abs(n)                %(quot_abs_Int)%
      • abs(m rem n) = abs(m) rem abs(n)                   %(rem_abs_Int)%
      • m = (m quot n) ∗ n + m rem n if ¬ n = 0           %(quot_rem_Int)%
      • m ˆ (a + b) = m ˆ a ∗ m ˆ b                           %(power_Int)%
end

spec RAT = %mono
      INT
then %mono
      generated type Rat ::= __/__(Int; Pos)
      ∀ i, j: Int; p, q: Pos
      • i / p = j / q ⇔ i ∗ q = j ∗ p                        %(equality_Rat)%
      sort   Int < Rat
      ∀ i: Int
      • i = i / 1                                     %(Int2Rat_embedding)%
then %def
      preds  __≤__, __<__, __≥__, __>__ : Rat × Rat
      ops    −__, abs : Rat → Rat;
             __+__, __−__, __∗__, min, max : Rat × Rat → Rat;
```

```
            __/__ : Rat × Rat →? Rat;
            __ˆ__ : Rat × Int → Rat
     %% implied operation attributes :
     ops    __+__ : Rat × Rat → Rat, comm, assoc, unit 0;    %implied
            __*__ : Rat × Rat → Rat, comm, assoc, unit 1;    %implied
            min, max : Rat × Rat → Rat, comm, assoc          %implied
     ∀ p, q: Pos; n: Nat; i, j: Int; x, y, z: Rat

     %% axioms concerning predicates
     • (i / p) ≤ (j / q) ⇔ (i ∗ q) ≤ (j ∗ p)                 %(leq_def_Rat)%
     • x ≥ y ⇔ y ≤ x                                         %(geq_def_Rat)%
     • x < y ⇔ x ≤ y ∧ ¬ x = y                               %(less_def_Rat)%
     • x > y ⇔ y < x                                      %(greater_def_Rat)%

     %% axioms concerning operations
     • − (i / p) = − i / p                                  %(minus_def_Rat)%
     • abs(i / p) = abs(i) / p                                %(abs_def_Rat)%
     • i / p + j / q = (i ∗ q + j ∗ p) / (p ∗ q)              %(add_def_Rat)%
     • x − y = x + − y                                        %(sub_def_Rat)%
     • (i / p) ∗ (j / q) = (i ∗ j) / (p ∗ q)                 %(mult_def_Rat)%
     • min(x, y) = x when x ≤ y else y                        %(min_def_Rat)%
     • max(x, y) = y when x ≤ y else x                        %(max_def_Rat)%
     • ¬ def x / 0                                        %(divide_def1_Rat)%
     • (x / y = z ⇔ x = z ∗ y) if ¬ y = 0                 %(divide_def2_Rat)%
     • x ˆ 0 = 1                                              %(power_0_Rat)%
     • x ˆ suc(n) = x ∗ x ˆ n                               %(power_suc_Rat)%
     • x ˆ (− p) = 1 / x ˆ p                                %(power_neg_Rat)%

     %% important laws
     • (x + y) ∗ z = x ∗ z + y ∗ z                 %(distr1_Rat)% %implied
     • z ∗ (x + y) = z ∗ x + z ∗ y                 %(distr2_Rat)% %implied
then %implies
     ∀ i, j: Int; p, q: Pos; x, y: Rat
     • i / p − j / q = (i ∗ q − j ∗ p) / (p ∗ q)               %(sub_rule_Rat)%
     • def x / y ⇔ ¬ y = 0                                  %(divide_dom_Rat)%
     • (i / p) / (j / q) = (i ∗ q) / (p ∗ j) if ¬ j = 0
                                                           %(divide_rule_Rat)%
     • x ˆ (i + j) = x ˆ i ∗ x ˆ j                              %(power_Rat)%
end

spec DecimalFraction = %mono
     Rat
then %def
     local
       op     tenPower : Nat → Nat
```

∀ *n*, *m*: *Nat*

%% tenPower(n):= min { 10^k | k in N \{0}, 10^k > n }:

• *tenPower*(*n*) = *10 when n* < *10 else 10* ∗ *tenPower*(*n div 10*)

%(tenPower_def)%

within

%% operations to represent a rational as a decimal fraction:

ops __:::__ : *Nat* × *Nat* → *Rat*;

__E__ : *Rat* × *Int* → *Rat*

∀ *r*: *Rat*; *n*, *m*: *Nat*; *p*: *Pos*; *i*: *Int*

%% represent the decimal fraction n.m as rational:

• *n* ::: *m* = *n* + *m* / *tenPower*(*m*) %(decfract_def)%

%% introduce an exponent:

• *r E i* = *r* ∗ *10* ^ *i* %(exponent_DecimalFraction)%

end

3

Library Basic/RelationsAndOrders

library Basic/RelationsAndOrders **version** 1.0

%authors(M. Roggenbach <csmarkus@swansea.ac.uk>, T. Mossakowski,
L. Schröder)**%**
%date : 18 December 2003
%{ This library provides

- specifications of binary relations of different sort,
- views stating that the numbers specified in the
Library Basic/Numbers are totally ordered, and
- a specification of Boolean Algebras.

Then, the different concepts specified are enriched with additional operations and predicates: In case of partial orders, the specification ExtPartialOrder provides the notions of inf, sup; the specification ExtTotalOrder adds the functions min and max to total orders; ExtBooleanAlgebra defines a complement operation as well as a less-or-equal relation for Boolean algebras.

Finally, the library provides non parametrized variants of these enriched specifications. **}%**

%display(__ ˜ __ %LATEX __$\sim$__**)%**

%display(__<=__ %LATEX __$\leq$__**)%**

%display(__>=__ %LATEX __$\geq$__**)%**

%display(__*cup*__ %LATEX __$\sqcup$__**)%**

%display(__*cap*__ %LATEX __$\sqcap$__**)%**

%display(*compl*__ %LATEX __$^{-1}$)**%**

%prec({__$\sqcup$__} < {__$\sqcap$__}**)%**

from Basic/Numbers **get** Nat, Int, Rat

```
spec Relation =
     sort  Elem
     pred  __∼__ : Elem × Elem
end

spec ReflexiveRelation =
     Relation
then
     ∀ x: Elem
     • x ∼ x                                          %(refl)%
end

spec IrreflexiveRelation =
     Relation
then
     ∀ x: Elem
     • ¬ x ∼ x                                        %(irrefl)%
end

spec SymmetricRelation =
     Relation
then
     ∀ x, y: Elem
     • x ∼ y if y ∼ x                                 %(sym)%
end

spec AsymmetricRelation =
     Relation
then
     ∀ x, y: Elem
     • ¬ x ∼ y if y ∼ x                               %(asym)%
end

spec AntisymmetricRelation =
     Relation


then
     ∀ x, y: Elem
     • x = y if x ∼ y ∧ y ∼ x                         %(antisym)%
end

spec TransitiveRelation =
     Relation
```

```
then
      ∀ x, y, z: Elem
      • x ∼ z if x ∼ y ∧ y ∼ z                                    %(trans)%
end

spec SimilarityRelation =
      ReflexiveRelation
and
      SymmetricRelation
end

spec PartialEquivalenceRelation =
      SymmetricRelation
and
      TransitiveRelation
end

spec EquivalenceRelation =
      ReflexiveRelation
and
      PartialEquivalenceRelation
end

spec PreOrder =
      {       ReflexiveRelation
       and
              TransitiveRelation
      }
      with pred __∼__ ↦ __≤__
end

spec StrictOrder =
      {       IrreflexiveRelation
       and
              TransitiveRelation
       then %implies
              AsymmetricRelation
      }
      with pred __∼__ ↦ __<__
end

spec PartialOrder =
      PreOrder
and
      AntisymmetricRelation with pred __∼__ ↦ __≤__
```

```
end

spec TotalOrder =
     PartialOrder
then
     ∀ x, y: Elem
     • x ≤ y ∨ y ≤ x                          %(dichotomy_TotalOrder)%
end

spec StrictTotalOrder =
     StrictOrder
then
     ∀ x, y: Elem
     • x < y ∨ y < x ∨ x = y           %(trichotomy_StrictTotalOrder)%
end

spec RightUniqueRelation =
     sorts  S, T
     pred   __R__ : S × T
     ∀ s: S; t1, t2: T
     • s R t1 ∧ s R t2 ⇒ t1 = t2
end

spec LeftTotalRelation =
     sorts  S, T
     pred   __R__ : S × T
     ∀ s: S
     • ∃ t: T • s R t
end

spec BooleanAlgebra =
     sort   Elem
     ops    0, 1 : Elem;
            __⊓__ : Elem × Elem → Elem, assoc, comm, unit 1;
            __⊔__ : Elem × Elem → Elem, assoc, comm, unit 0
     ∀ x, y, z: Elem
     • x ⊓ (x ⊔ y) = x                            %(absorption_def1)%
     • x ⊔ x ⊓ y = x                              %(absorption_def2)%
     • x ⊓ 0 = 0                                      %(zeroAndCap)%
     • x ⊔ 1 = 1                                      %(oneAndCup)%
     • x ⊓ (y ⊔ z) = x ⊓ y ⊔ x ⊓ z           %(distr1_BooleanAlgebra)%
     • x ⊔ y ⊓ z = (x ⊔ y) ⊓ (x ⊔ z)         %(distr2_BooleanAlgebra)%
     • ∃ x′: Elem • x ⊔ x′ = 1 ∧ x ⊓ x′ = 0
                                            %(inverse_BooleanAlgebra)%
then %implies
```

```
     ops    __⊔__, __⊓__ : Elem × Elem → Elem, idem
     ∀ x: Elem
     • ∃! x′: Elem • x ⊔ x′ = 1 ∧ x ⊓ x′ = 0
                                    %(uniqueComplement_BooleanAlgebra)%
end

spec ExtPartialOrder [PartialOrder] = %def
     preds __≤__, __<__, __≥__, __>__ : Elem × Elem
     ∀ x, y: Elem
     • x ≥ y ⇔ y ≤ x                          %(geq_def_ExtPartialOrder)%
     • x < y ⇔ x ≤ y ∧ ¬ x = y               %(less_def_ExtPartialOrder)%
     • x > y ⇔ y < x                       %(greater_def_ExtPartialOrder)%
and
     ops    inf, sup : Elem × Elem →? Elem, comm                  %implied
     ∀ x, y, z: Elem
     • inf(x, y) = z ⇔
       z ≤ x ∧ z ≤ y ∧ (∀ t: Elem • t ≤ x ∧ t ≤ y ⇒ t ≤ z)
                                               %(inf_def_ExtPartialOrder)%
     • sup(x, y) = z ⇔
       x ≤ z ∧ y ≤ z ∧ (∀ t: Elem • x ≤ t ∧ y ≤ t ⇒ z ≤ t)
                                               %(sup_def_ExtPartialOrder)%
end

spec ExtTotalOrder [TotalOrder] = %def
     ExtPartialOrder [PartialOrder]
and
     ops    min, max : Elem × Elem → Elem, comm, assoc           %implied
     ∀ x, y: Elem
     • min(x, y) = x when x ≤ y else y          %(min_def_ExtTotalOrder)%
     • max(x, y) = y when x ≤ y else x          %(max_def_ExtTotalOrder)%
then %implies
     ∀ x, y: Elem
     • min(x, y) = inf(x, y)                            %(min_inf_relation)%
     • max(x, y) = sup(x, y)                            %(max_sup_relation)%
end

spec ExtBooleanAlgebra [BooleanAlgebra] = %def
     preds __≤__, __<__, __≥__, __>__ : Elem × Elem
     ∀ x, y: Elem
     • x ≤ y ⇔ x ⊓ y = x                     %(leq_def_ExtBooleanAlgebra)%
     • x ≥ y ⇔ y ≤ x                         %(geq_def_ExtBooleanAlgebra)%
     • x < y ⇔ x ≤ y ∧ ¬ x = y              %(less_def_ExtBooleanAlgebra)%
     • x > y ⇔ y < x                     %(greater_def_ExtBooleanAlgebr)%
and
```

{ **op** $\_\_^{-1} : \mathit{Elem} \to \mathit{Elem}$
$\forall\, x, y{:}\ \mathit{Elem}$
$\bullet\ x^{\ -1} = y \Leftrightarrow x \sqcup y = \mathit{1} \wedge x \sqcap y = \mathit{0}$
%(compl_def_ExtBooleanAlgebra)%

then %implies
$\forall\, x, y{:}\ \mathit{Elem}$
$\bullet\ (x \sqcap y)^{\ -1} = x^{\ -1} \sqcup y^{\ -1}$ %(de_Morgan1)%
$\bullet\ (x \sqcup y)^{\ -1} = x^{\ -1} \sqcap y^{\ -1}$ %(de_Morgan2)%
$\bullet\ (x^{\ -1})^{\ -1} = x$ %(involution_compl_ExtBooleanAlgebra)%
}
end

spec RichPartialOrder =
ExtPartialOrder [PartialOrder]
end

spec RichTotalOrder =
ExtTotalOrder [TotalOrder]
end

spec RichBooleanAlgebra =
ExtBooleanAlgebra [BooleanAlgebra]
end

view TotalOrder_in_Nat : TotalOrder **to** Nat =
sort *Elem* ↦ *Nat*
end

view TotalOrder_in_Int : TotalOrder **to** Int =
sort *Elem* ↦ *Int*
end

view TotalOrder_in_Rat : TotalOrder **to** Rat =
sort *Elem* ↦ *Rat*
end

view PartialOrder_in_ExtBooleanAlgebra [BooleanAlgebra] :
PartialOrder **to** ExtBooleanAlgebra [BooleanAlgebra]
end

4

Library Basic/Algebra_I

```
library Basic/Algebra_I version 1.0

%authors : M. Roggenbach, T. Mossakowski, L. Schröder <lschrode@tzi.de>
%date : 21 May 2003
%prec({__*__} < {__^__})%
%prec({__+__, __−__} < {__/__, __*__})%
%left_assoc(__+__, __*__, __^__)%


from Basic/RelationsAndOrders get
        TotalOrder, ExtTotalOrder, RichTotalOrder,
        PreOrder, EquivalenceRelation

from Basic/Numbers get Nat, Int, Rat

spec Monoid =
        sort   Elem
        ops    e : Elem;
               __*__ : Elem × Elem → Elem, assoc, unit e
end

spec CommutativeMonoid =
        Monoid
then
        op     __*__ : Elem × Elem → Elem, comm
end

spec Group =
        Monoid
then
```

```
     ∀ x: Elem
     • ∃ x′: Elem • x′ ∗ x = e
end

spec AbelianGroup =
     Group
and
     CommutativeMonoid
end

spec Ring =
     AbelianGroup with sort Elem, ops __∗__ ↦ __+__, e ↦ 0
and
     Monoid with ops e, __∗__
then
     ∀ x, y, z: Elem
     • (x + y) ∗ z = x ∗ z + y ∗ z                    %(distr1_Ring)%
     • z ∗ (x + y) = z ∗ x + z ∗ y                    %(distr2_Ring)%
end

view AbelianGroup_in_Ring_add : AbelianGroup to Ring =
     ops e ↦ 0, __∗__ ↦ __+__
end

spec CommutativeRing =
     Ring with ops 0, __+__, e, __∗__
and
     CommutativeMonoid with ops e, __∗__
end

spec IntegralDomain =
     CommutativeRing
then
     ∀ x, y: Elem
     • x = 0 ∨ y = 0 if x ∗ y = 0                      %(noZeroDiv)%
     • ¬ e = 0                                        %(zeroNeqOne)%
end

spec ConstructField =
     CommutativeRing
then
     • ¬ e = 0
     sort   NonZeroElem = {x: Elem • ¬ x = 0}
and
     Group with sort Elem ↦ NonZeroElem, ops e, __∗__
```

```
end

%% an obvious view which helps to write the specification ExtField:
view AbelianGroup_in_ConstructField :
     AbelianGroup to ConstructField =
     sort Elem ↦ NonZeroElem
end

spec Field =
     ConstructField hide sort NonZeroElem
end

view IntegralDomain_in_Field : IntegralDomain to Field
end

spec ExtMonoid [Monoid] given Nat = %def
     op    __ ^__ : Elem × Nat → Elem
     ∀ x: Elem; n: Pos
     • x ^ 0 = e                                  %(pow_0_Monoid)%
     • x ^ suc(n) = x ∗ x ^ n                     %(pow_suc_Monoid)%
then %implies
     ∀ x: Elem; n, m: Nat
     • e ^ n = e                                  %(pow_unit_Monoid)%
     • x ^ (n + m) = x ^ n ∗ x ^ m                %(pow_add_Monoid)%
     • x ^ (n ∗ m) = (x ^ n) ^ m                  %(pow_mult_Monoid)%
end

spec ExtCommutativeMonoid [CommutativeMonoid]
     given Nat = %def
     ExtMonoid [Monoid]
then %implies
     ∀ x, y: Elem; n: Nat
     • x ^ n ∗ y ^ n = (x ∗ y) ^ n           %(pow_basemult_CMonoid)%
end

spec ExtGroup [Group] given Int = %def
     ExtMonoid [Monoid]
then
     ops   __ ^__ : Elem × Int → Elem;
           inv : Elem → Elem;
           __/__ : Elem × Elem → Elem
     ∀ x, y: Elem; p: Pos
     • inv(x) ∗ x = e                             %(inv_def_Group)%
     • x / y = x ∗ inv(y)                         %(div_def_Group)%
     • x ^ (− p) = inv(x ^ p)                     %(pow_neg_Group)%
```

```
then %implies
      ∀ x, y, z: Elem; n, m: Int
      • x * inv(x) = e                                %(rightInv_Group)%
      • x = y if z * x = z * y                        %(leftCancel_Group)%
      • x = y if x * z = y * z                        %(rightCancel_Group)%
      • inv(inv(x)) = x                               %(invInv_Group)%
      • inv(e) = e                                    %(invUnit_Group)%
      • inv(x * y) = inv(y) * inv(x)                  %(invMult_Group)%
      • e ^ n = e                                     %(pow_unit_Group)%
      • x ^ (n + m) = x ^ n * x ^ m                   %(pow_add_Group)%
      • x ^ (n * m) = (x ^ n) ^ m                     %(pow_mult_Group)%
end

spec ExtAbelianGroup [AbelianGroup] given Int = %def
      ExtGroup [AbelianGroup]
then %implies
      ∀ x, y: Elem; n: Int
      • x ^ n * y ^ n = (x * y) ^ n                   %(pow_basemult_AbGroup)%
end

spec ExtRing [Ring] given Int = %mono
      ExtAbelianGroup [view AbelianGroup_in_Ring_add]
      with ops inv ↦ −__, __/__ ↦ __−__, __^__ ↦ __times__
and
      ExtMonoid [Monoid] with op __^__
and
      preds isIrred, isUnit : Elem
      sorts  NonZero[Elem] = {x: Elem • ¬ x = 0};
             RUnit[Elem] = {x: Elem • isUnit(x)};
             Irred[Elem] = {x: Elem • isIrred(x)}
      ∀ x, y: Elem
      • isUnit(x) ⇔ ∃ y: Elem • x * y = e ∧ y * x = e
                                                      %(isUnit_def_Ring)%
      • isIrred(x) ⇔
        ¬ isUnit(x) ∧ (∀ y, z: Elem • isUnit(y) ∨ isUnit(z) if x = y * z)
                                                      %(isIrred_def_Ring)%
then %def
      ops    e : RUnit[Elem];
             −__ : RUnit[Elem] → RUnit[Elem];
             __*__ : RUnit[Elem] × RUnit[Elem] → RUnit[Elem]
end

view Group_in_ExtRing [Ring] given Int :
      Group to ExtRing [Ring] =
      sort Elem ↦ RUnit[Elem]
```

```
end

spec ExtCommutativeRing [CommutativeRing]
     given Int = %mono
     ExtRing [Ring]
then
     preds hasNoZeroDivisors : ();
           __divides__ : Elem × Elem;
           associated : Elem × Elem
     ∀ x, y: Elem
     • hasNoZeroDivisors ⇔ ∀ x, y: Elem • x = 0 ∨ y = 0 if x ∗ y = 0
                                                 %(hasNoZeroDivisors_def)%
     • x divides y ⇔ ∃ z: Elem • x ∗ z = y               %(divides_def)%
     • associated(x, y) ⇔ ∃ u: RUnit[Elem] • x = u ∗ y
                                                        %(associated_def)%
then %implies
     ∀ x, y: Elem
     • associated(x, y) ⇔ x divides y ∧ y divides x
end

view PreOrder_in_ExtCRing [CommutativeRing] given Int :
     PreOrder to ExtCommutativeRing [CommutativeRing] =
     pred __≤__ ↦ __divides__
end

view AbelianGroup_in_ExtCRing [CommutativeRing]
     given Int :
     AbelianGroup to
     ExtCommutativeRing [CommutativeRing] =
     sort Elem ↦ RUnit[Elem]
end

view EqRel_in_ExtCRing [CommutativeRing] given Int :
     EquivalenceRelation to
     ExtCommutativeRing [CommutativeRing] =
     pred __∼__ ↦ associated
end

spec ExtIntegralDomain [IntegralDomain]
     given Int = %mono
     ExtCommutativeRing [CommutativeRing]
then
     op    __∗__ : NonZero[Elem] × NonZero[Elem] → NonZero[Elem]
then %implies
     • hasNoZeroDivisors
```

```
end

spec ExtField [Field] given Int = %mono
      ExtRing [Ring]
then
      closed {ExtAbelianGroup
                  [view AbelianGroup_in_ConstructField]
               with sort NonZeroElem ↦ NonZero[Elem],
                     ops inv, __/__, __ˆ__
              }
then
      op      __/__ : Elem × Elem →? Elem
      ∀ x: Elem; n: NonZero[Elem]
      • 0 / n = 0                                    %(div_def1_Field)%
      • ¬ def x / 0                                  %(div_def2_Field)%
then %implies
      ∀ x, y: Elem
      • def x / y ⇔ ¬ y = 0                          %(div_dom_Field)%
end

spec RichMonoid =
      ExtMonoid [Monoid]
end

spec RichCommutativeMonoid =
      ExtCommutativeMonoid [CommutativeMonoid]
end

spec RichGroup =
      ExtGroup [Group]
end

spec RichAbelianGroup =
      ExtAbelianGroup [AbelianGroup]
end

spec RichRing =
      ExtRing [Ring]
end

view Group_in_RichRing : Group to RichRing =
      sort Elem ↦ RUnit[Elem]
end

spec RichCommutativeRing =
     ExtCommutativeRing [CommutativeRing]
end

view PreOrder_in_RichCRing :
     PreOrder to RichCommutativeRing =
     pred __≤__ ↦ __divides__
end

view EqRel_in_RichCRing :
     EquivalenceRelation to RichCommutativeRing =
     pred __∼__ ↦ associated
end

view AbelianGroup_in_RichCRing :
     AbelianGroup to RichCommutativeRing =
     sort Elem ↦ RUnit[Elem]
end

spec RichIntegralDomain =
     ExtIntegralDomain [IntegralDomain]
end

spec RichField =
     ExtField [Field]
end

view CommutativeMonoid_in_Nat_Add :
     CommutativeMonoid to Nat =
     sort Elem ↦ Nat, ops e ↦ 0, __*__ ↦ __+__
end

view CommutativeMonoid_in_Nat_Mult :
     CommutativeMonoid to Nat =
     sort Elem ↦ Nat, ops e ↦ 1, __*__ ↦ __*__
end

view CommutativeMonoid_in_Int_Mult :
     CommutativeMonoid to
     {     Int
      then
           op     1 : Int
     } =
     sort Elem ↦ Int, ops e ↦ 1, __*__ ↦ __*__
end
```

```
view AbelianGroup_in_Int_Add :
      AbelianGroup to
      {       Int
       then
              op      0 : Int
      } =
      sort Elem ↦ Int, ops __*__ ↦ __+__, e ↦ 0
end

view IntegralDomain_in_Int :
      IntegralDomain to
      {       Int
       then
              op      1 : Int
      } =
      sort Elem ↦ Int, op e ↦ 1
end

view Field_in_Rat :
      Field to
      {       Rat
       then
              op      1 : Rat
      } =
      sort Elem ↦ Rat, op e ↦ 1
end
```

5

Library Basic/SimpleDatatypes

```
library Basic/SimpleDatatypes version 1.0

%authors : T. Mossakowski <till@tzi.de>, M. Roggenbach, L. Schröder
%date : 18 June 2002
%{ This library provides unstructured datatypes like Booleans
   and characters.
   The Booleans are shown to be a Boolean algebra and then
   enriched with the usual Boolean algebra operations.
   The characters are defined to be the subset 0..255
   of the natural numbers, and then constants for different
   representations (ASCII, decimal, octal, hexadecimal)
   are introduced. }%
%prec({__Or__} < {__And__})%


from Basic/RelationsAndOrders get
       BooleanAlgebra, ExtBooleanAlgebra

from Basic/Numbers get Nat

spec Boolean = %mono
       free type
              Boolean ::= True | False
       %%capital True and False, since true and false are predefined
ops    Not__ : Boolean → Boolean;
       __And__, __Or__ : Boolean × Boolean → Boolean
∀ x, y: Boolean
• Not False = True                                        %(Not_False)%
• Not True = False                                        %(Not_True)%
• False And False = False                                 %(And_def1)%
• False And True = False                                  %(And_def2)%
• True And False = False                                         %(And_def3)%
• True And True = True                                           %(And_def4)%
• x Or y = Not (Not x And Not y)                                   %(Or_def)%
end

view BooleanAlgebra_in_Boolean :
     BooleanAlgebra to Boolean =
     sort Elem ↦ Boolean,
     ops 0 ↦ False, 1 ↦ True, __⊓__ ↦ __And__,
         __⊔__ ↦ __Or__
end

%{ Get derived operations for Boolean Algebras }%
spec RichBoolean = %mono
     ExtBooleanAlgebra [view BooleanAlgebra_in_Boolean]
     with pred __≤__, op __^{-1}
end

spec Char = %mono
     Nat
then %mono
%% characters are generated from natural numbers 0..255
     type  Char ::= chr(ord:Nat)?
     ∀ n: Nat; c: Char
     • def chr(n) ⇔ n ≤ 255                                      %(chr_dom)%
     • chr(ord(c)) = c                                   %(chr_ord_inverse)%
     %% definition of individual characters by decimal codes:
     ops   '\000': Char = chr(0);                              %(slash_000)%
           '\001': Char = chr(1);                              %(slash_001)%
           '\002': Char = chr(2);                              %(slash_002)%
           '\003': Char = chr(3);                              %(slash_003)%
           '\004': Char = chr(4);                              %(slash_004)%
           ...
           '\255': Char = chr(255)                             %(slash_255)%
     %% definition of the printable characters
     %% This relies on the character names
     ops   ' ': Char = '\032';                             %(printable_32)%
           '!': Char = '\033';                             %(printable_33)%
           '\"': Char = '\034';                            %(printable_34)%
           '#': Char = '\035';                             %(printable_35)%
           ...
           'ÿ': Char = '\255'                             %(printable_255)%
     preds isLetter(c: Char) ⇔
           (ord('A') ≤ ord(c) ∧ ord(c) ≤ ord('Z')) ∨
           (ord('a') ≤ ord(c) ∧ ord(c) ≤ ord('z'));
```

```
                                                              %(isLetter_def)%
       isDigit(c: Char) ⇔ ord('0') ≤ ord(c) ∧ ord(c) ≤ ord('9');
                                                               %(isDigit_def)%
       isPrintable(c: Char) ⇔
       (ord(' ') ≤ ord(c) ∧ ord(c) ≤ ord('˜')) ∨
       (ord(' ') ≤ ord(c) ∧ ord(c) ≤ ord('ÿ'))      %(isPrintable_def)%
%% definition of characters by octal codes, i.e. '\o ppp',
%% where p in {0,1,...,7} :
ops    '\o000': Char = '\000';                               %(slash_o000)%
       '\o001': Char = '\001';                               %(slash_o001)%
       '\o002': Char = '\002';                               %(slash_o002)%
       '\o003': Char = '\003';                               %(slash_o003)%
       '\o004': Char = '\004';                               %(slash_o004)%
       ...
       '\o377': Char = '\255'                                %(slash_o377)%
%% definition of characters by hexadecimal codes, i.e. '\xhh',
%% where h in {0,1,..., F} :
ops    '\x00': Char = '\000';                                 %(slash_x00)%
       '\x01': Char = '\001';                                 %(slash_x01)%
       '\x02': Char = '\002';                                 %(slash_x02)%
       '\x03': Char = '\003';                                 %(slash_x03)%
       '\x04': Char = '\004';                                 %(slash_x04)%
       ...
       '\xFF': Char = '\255'                                  %(slash_xFF)%
%% special characters:
ops    NUL: Char = '\000';                                      %(NUL_def)%
       SOH: Char = '\001';                                      %(SOH_def)%
       SYX: Char = '\002';                                      %(SYX_def)%
       ETX: Char = '\003';                                      %(ETX_def)%
       EOT: Char = '\004';                                      %(EOT_def)%
       ENQ: Char = '\005';                                      %(ENQ_def)%
       ACK: Char = '\006';                                      %(ACK_def)%
       BEL: Char = '\007';                                      %(BEL_def)%
       BS: Char = '\008';                                        %(BS_def)%
       HT: Char = '\009';                                        %(HT_def)%
       LF: Char = '\010';                                        %(LF_def)%
       VT: Char = '\011';                                        %(VT_def)%
       FF: Char = '\012';                                        %(FF_def)%
       CR: Char = '\013';                                        %(CR_def)%
       SO: Char = '\014';                                        %(SO_def)%
       SI: Char = '\015';                                        %(SI_def)%
       DLE: Char = '\016';                                      %(DLE_def)%
       DC1: Char = '\017';                                      %(DC1_def)%
       DC2: Char = '\018';                                      %(DC2_def)%
       DC3: Char = '\019';                                      %(DC3_def)%
```

```
          DC4: Char = '\020';                          %(DC4_def)%
          NAK: Char = '\021';                          %(NAK_def)%
          SYN: Char = '\022';                          %(SYN_def)%
          ETB: Char = '\023';                          %(ETB_def)%
          CAN: Char = '\024';                          %(CAN_def)%
          EM: Char = '\025';                            %(EM_def)%
          SUB: Char = '\026';                          %(SUB_def)%
          ESC: Char = '\027';                          %(ESC_def)%
          FS: Char = '\028';                            %(FS_def)%
          GS: Char = '\029';                            %(GS_def)%
          RS: Char = '\030';                            %(RS_def)%
          US: Char = '\031';                            %(US_def)%
          SP: Char = '\032';                            %(SP_def)%
          DEL: Char = '\127'                           %(DEL_def)%
    %% alternative names for special characters:
    ops   NL: Char = LF;                                %(NL_def)%
          NP: Char = FF                                 %(NP_def)%
    %% character constants:
    ops   '\n': Char = NL;                             %(slash_n)%
          '\t': Char = HT;                             %(slash_t)%
          '\v': Char = VT;                             %(slash_v)%
          '\b': Char = BS;                             %(slash_b)%
          '\r': Char = CR;                             %(slash_r)%
          '\f': Char = FF;                             %(slash_f)%
          '\a': Char = BEL;                            %(slash_a)%
          '\?': Char = '?'                         %(slash_quest)%
end
```

6

Library Basic/StructuredDatatypes

```
library Basic/StructuredDatatypes version 1.0

%authors(M. Roggenbach <csmarkus@swansea.ac.uk>, T. Mossakowski,
         L. Schröder)%
%date : 18 December 2003
%left_assoc(__+__, __−__, __++__)%
%left_assoc(__∪__, __∩__)%
%right_assoc(__::__)%
%list [__], [], __::__
%prec({__++__} < {__::__})%
%string emptyString, __: @ :__
%display(nothing %LATEX ⊥)%
%display(__eps__ %LATEX __ϵ__)%
%% note: \epsilon looks different from \in,
%% which is used as LaTeX display syntax
%% for the CASL membership predicate
%display(__isSubsetOf__ %LATEX __⊆__)%
%display(__intersection__ %LATEX __∩__)%
%display(__union__ %LATEX __∪__)%
%display(#__ %LATEX ♯__)%


from Basic/Numbers get Nat, Int

from Basic/RelationsAndOrders get
     PartialOrder, BooleanAlgebra

from Basic/Algebra_I get Monoid, CommutativeMonoid

from Basic/SimpleDatatypes get Char
```

```
spec MAYBE [sort S] = %mono
     free type Maybe[S] ::= ⊥ | sort S
end

spec PAIR [sort S] [sort T] = %mono
     free type Pair[S,T] ::= pair(first:S; second:T)
end

spec GENERATESET [sort Elem] = %mono
     generated type Set[Elem] ::= {} | __+__(Set[Elem]; Elem)
     pred  __ϵ__ : Elem × Set[Elem]
     ∀ x, y: Elem; M, N: Set[Elem]
     • ¬ x ϵ {}                                   %(elemOf_empty_Set)%
     • x ϵ (M + y) ⇔ x = y ∨ x ϵ M            %(elemOf_NonEmpty_Set)%
     • M = N ⇔ ∀ x: Elem • x ϵ M ⇔ x ϵ N            %(equality_Set)%
end

spec SET [sort Elem] given NAT = %mono
     GENERATESET [sort Elem]
then %def
     preds isNonEmpty : Set[Elem];
           __⊆__ : Set[Elem] × Set[Elem]
     ops   {__} : Elem → Set[Elem];
           ♯__ : Set[Elem] → Nat;
           __+__ : Elem × Set[Elem] → Set[Elem];
           __−__ : Set[Elem] × Elem → Set[Elem];
           __∩__, __∪__, __−__, __symDiff__ :
                  Set[Elem] × Set[Elem] → Set[Elem]
     %% implied operation attributes
     ops   __∪__ : Set[Elem] × Set[Elem] → Set[Elem],
              assoc, comm, idem, unit {};                        %implied
           __∩__ : Set[Elem] × Set[Elem] → Set[Elem],
              assoc, comm, idem                                  %implied
     ∀ x, y: Elem; M, N, O: Set[Elem]

     %% axioms concerning predicates
     • isNonEmpty(M) ⇔ ¬ M = {}                    %(isNonEmpty_def)%
     • M ⊆ N ⇔ ∀ x: Elem • x ϵ M ⇒ x ϵ N            %(isSubsetOf_def)%

     %% axioms concerning operations
     • { x } = {} + x                              %(singletonSet_def)%
     • ♯ {} = 0                                  %(numberOf_emptySet)%
     • ♯ (M + x) = ♯ M when x ϵ M else ♯ M + 1
                                              %(numberOf_NonEmptySet)%
     • x + M = M + x                                %(addElem_def_Set)%
```

• $\{\} - y = \{\}$ %(remElem_EmptySet)%
• $(M + x) - y = M - y$ *when* $x = y$ *else* $(M - y) + x$
%(remElem_NonEmptySet)%
• $M \cap \{\} = \{\}$ %(intersection_Emptyset)%
• $M \cap (N + x) = (M \cap N) + x$ *when* $x \epsilon M$ *else* $M \cap N$
%(intersection_NonEmptySet)%
• $M \cup \{\} = M$ %(union_EmptySet)%
• $M \cup (N + x) = M \cup N$ *when* $x \epsilon M$ *else* $(M \cup N) + x$
%(union_NonEmptySet)%
• $M - \{\} = M$ %(dif_Emptyset)%
• $M - (N + x) = M - N - x$ %(dif_Emptyset)%
• $M\ \mathit{symDiff}\ N = (M - N) \cup (N - M)$ %(symDiff_def)%

%% important laws
• $(M \cup N) \cap O = (M \cap O) \cup (N \cap O)$ %(distr1_Set)% **%implied**
• $O \cap (M \cup N) = (O \cap M) \cup (O \cap N)$ %(distr2_Set)% **%implied**
then %implies
$\forall\ x, y\colon \mathit{Elem}; M, N\colon \mathit{Set}[\mathit{Elem}]$
• $\sharp\ (M \cup N) = (\sharp\ M + \sharp\ N) -?\ \sharp\ (M \cap N)$ %(set_counting)%
• $\sharp\ M \leq \sharp\ (M \cup N)$ %(union_counting)%
• $M \subseteq (M \cup N)$ %(union_isSubsetOf)%
• $(M \cap N) \subseteq M$ %(intersection_isSubsetOf)%
end

view PartialOrder_in_Set [**sort** *Elem*] **given** Nat :
PartialOrder **to** Set [**sort** *Elem*] =
sort *Elem* ↦ *Set*[*Elem*], **pred** __≤__ ↦ __⊆__
end

spec PowerSet [Set [**sort** *Elem*]
then
op X : *Set*[*Elem*]]
given Nat = **%mono**
sorts $\mathit{PowerSet}[X] = \{Y\colon \mathit{Set}[\mathit{Elem}] \bullet Y \subseteq X\}$;
$\mathit{Elem}[X] = \{x\colon \mathit{Elem} \bullet x \epsilon X\}$
preds __ϵ__ : $\mathit{Elem}[X] \times \mathit{PowerSet}[X]$;
__⊆__ : $\mathit{PowerSet}[X] \times \mathit{PowerSet}[X]$;
isNonEmpty : $\mathit{PowerSet}[X]$
ops $\{\}, X : \mathit{PowerSet}[X]$;
♯__ : $\mathit{PowerSet}[X] \to \mathit{Nat}$;
__+__ : $\mathit{Elem}[X] \times \mathit{PowerSet}[X] \to \mathit{PowerSet}[X]$;
__−__ : $\mathit{PowerSet}[X] \times \mathit{Elem}[X] \to \mathit{PowerSet}[X]$;
__∩__, __∪__, __−__, __*symDiff*__ :
$\mathit{PowerSet}[X] \times \mathit{PowerSet}[X] \to \mathit{PowerSet}[X]$

```
    %% implied operation attributes
    ops     __∪__ : PowerSet[X] × PowerSet[X] → PowerSet[X],
                assoc, comm, idem, unit {};
                                                                    %implied
            __∩__ : PowerSet[X] × PowerSet[X] → PowerSet[X],
                assoc, comm, idem
                                                                    %implied
            %% important laws
    ∀ M, N, O: PowerSet[X]
    • (M ∪ N) ∩ O = (M ∩ O) ∪ (N ∩ O)
                                                %(distr1_PowerSet)% %implied
    • O ∩ (M ∪ N) = (O ∩ M) ∪ (O ∩ N)
                                                %(distr2_PowerSet)% %implied
end

view BooleanAlgebra_in_PowerSet [      Set [sort Elem]
                                  then
                                       op X : Set[Elem]]
    given Nat :
    BooleanAlgebra to
    PowerSet [      Set [sort Elem]
              then
                    op X : Set[Elem]] =
    sort Elem ↦ PowerSet[X],
    ops 0 ↦ {}, 1 ↦ X, __⊓__ ↦ __∩__, __⊔__ ↦ __∪__
end

spec GenerateList [sort Elem] = %mono
    free type List[Elem] ::= [] | __::__(first:?Elem; rest:?List[Elem])
end

spec List [sort Elem] given Nat = %mono
    GenerateList [sort Elem]
then %def
    preds isEmpty : List[Elem];
            __ϵ__ : Elem × List[Elem]
    ops     __+__ : List[Elem] × Elem → List[Elem];
            first, last : List[Elem] →? Elem;
            front, rest : List[Elem] →? List[Elem];
            ♯__ : List[Elem] → Nat;
            __++__ : List[Elem] × List[Elem] → List[Elem];
            reverse : List[Elem] → List[Elem];
            __!__ : List[Elem] × Nat →? Elem;
            take, drop : Nat × List[Elem] →? List[Elem];
            freq : List[Elem] × Elem → Nat
```

$\forall\ x, y\colon \mathit{Elem};\ n\colon \mathit{Nat};\ p\colon \mathit{Pos};\ L, K\colon \mathit{List}[\mathit{Elem}]$

%% axioms concerning predicates

• $\mathit{isEmpty}(L) \Leftrightarrow L = [\,]$ %(isEmpty_def)%

• $\neg\ x \mathrel{\epsilon} [\,]$ %(List_elemOf_nil)%

• $x \mathrel{\epsilon} (x :: L)$ %(List_elemOf_NeList1)%

• $(x \mathrel{\epsilon} (y :: L) \Leftrightarrow x \mathrel{\epsilon} L)\ \mathit{if}\ \neg\ x = y$ %(List_elemOf_NeList2)%

%% axioms concerning operations

• $L + x = L \mathbin{++} [\ x\ ]$ %(append_def)%

• $\mathit{def}\ \mathit{first}(L) \Leftrightarrow \neg\ \mathit{isEmpty}(L)$ %(first_dom)% **%implied**

• $\neg\ \mathit{def}\ \mathit{first}([\,])$ %(first_partial_nil)% **%implied**

• $\mathit{def}\ \mathit{last}(L) \Leftrightarrow \neg\ \mathit{isEmpty}(L)$ %(last_dom)% **%implied**

• $\neg\ \mathit{def}\ \mathit{last}([\,])$ %(last_nil)%

• $\mathit{last}(x :: L) = x\ \mathit{when}\ \mathit{isEmpty}(L)\ \mathit{else}\ \mathit{last}(L)$ %(last_NeList)%

• $\mathit{def}\ \mathit{front}(L) \Leftrightarrow \neg\ \mathit{isEmpty}(L)$ %(front_dom)% **%implied**

• $\neg\ \mathit{def}\ \mathit{front}([\,])$ %(front_nil)%

• $\mathit{front}(L + x) = L$ %(front_NeList)%

• $\mathit{def}\ \mathit{rest}(L) \Leftrightarrow \neg\ \mathit{isEmpty}(L)$ %(rest_dom)% **%implied**

• $\neg\ \mathit{def}\ \mathit{rest}([\,])$ %(rest_nil)% **%implied**

• $\sharp\ [\,] = \mathit{0}$ %(numberOf_nil_List)%

• $\sharp\ (x :: L) = \mathit{suc}(\sharp\ L)$ %(numberOf_NeList_List)%

• $[\,] \mathbin{++} K = K$ %(concat_nil_List)%

• $x :: L \mathbin{++} K = x :: (L \mathbin{++} K)$ %(concat_NeList_List)%

• $\mathit{reverse}([\,]) = [\,]$ %(reverse_nil)%

• $\mathit{reverse}(x :: L) = \mathit{reverse}(L) \mathbin{++} [\ x\ ]$ %(reverse_NeList)%

• $\mathit{def}\ L\ !\ n \Leftrightarrow \sharp\ L \geq n$ %(index_dom)% **%implied**

• $\neg\ \mathit{def}\ [\,]\ !\ n$ %(index_nil)%

• $(x :: L)\ !\ \mathit{0} = x$ %(index_0)%

• $(x :: L)\ !\ \mathit{suc}(p) = L\ !\ p$ %(index_suc)%

• $\mathit{def}\ \mathit{take}(n, L) \Leftrightarrow \sharp\ L \geq n$ %(take_dom)% **%implied**

• $\mathit{take}(n, L) = K \Leftrightarrow \exists\ S\colon \mathit{List}[\mathit{Elem}] \bullet K \mathbin{++} S = L \wedge \sharp\ K = n$
%(take_def)%

• $\mathit{def}\ \mathit{drop}(n, L) \Leftrightarrow \sharp\ L \geq n$ %(drop_dom)% **%implied**

• $\mathit{drop}(n, L) = K \Leftrightarrow \exists\ S\colon \mathit{List}[\mathit{Elem}] \bullet S \mathbin{++} K = L \wedge \sharp\ S = n$
%(drop_def)%

• $\mathit{freq}([\,], x) = \mathit{0}$ %(freq_nil)%

• $\mathit{freq}(x :: L, y) = \mathit{suc}(\mathit{freq}(L, y))\ \mathit{when}\ x = y\ \mathit{else}\ \mathit{freq}(L, y)$
%(freq_NeList)%

then %implies

free type $\mathit{List}[\mathit{Elem}] ::= [] \mid \_\_{+}\_\_(\mathit{front}{:}?\mathit{List}[\mathit{Elem}];\ \mathit{last}{:}?\mathit{Elem})$

$\forall\ L\colon \mathit{List}[\mathit{Elem}]$

• $\mathit{first}(L) :: \mathit{rest}(L) = L\ \mathit{if}\ \neg\ \mathit{isEmpty}(L)$ %(first_rest)%

• $\mathit{front}(L) + \mathit{last}(L) = L\ \mathit{if}\ \neg\ \mathit{isEmpty}(L)$ %(front_last)%

end

```
view MONOID_IN_LIST [sort Elem] given NAT :
     MONOID to LIST [sort Elem] =
     sort Elem ↦ List[Elem], ops e ↦ [], __*__ ↦ __++__
end

spec STRING = %mono
     LIST [CHAR fit sort Elem ↦ Char]
     with sort List[Char] ↦ String,
          ops [] ↦ emptyString, __::__ ↦ __: @ :__
end

spec GENERATEMAP [sort S] [sort T] = %mono
     generated type Map[S,T] ::= empty | __[__/__](Map[S,T]; T; S)
     op    lookup : S × Map[S,T] →? T
     ∀ M, N: Map[S,T]; s, s1, s2: S; t1, t2: T
     • ¬ def lookup(s, empty)                          %(lookup_empty_Map)%
     • lookup(s, M [ t1 / s1 ]) = t1 when s = s1 else lookup(s, M)
                                                     %(lookup_nonEmpty_Map)%
     • M = N ⇔ ∀ s: S • lookup(s, M) = lookup(s, N)
                                                            %(equality_Map)%
end

spec MAP [sort S] [sort T] given NAT = %mono
     GENERATEMAP [sort S] [sort T]
and
     SET [sort S]
and
     SET [sort T]
then %def
     free type Entry[S,T] ::= [__/__](target:T; source:S)
     preds isEmpty : Map[S,T];
           __ϵ__ : Entry[S,T] × Map[S,T];
           __::__− >__ : Map[S,T] × Set[S] × Set[T]
     ops   __+__, __−__ : Map[S,T] × Entry[S,T] → Map[S,T];
           __−__ : Map[S,T] × S → Map[S,T];
           __−__ : Map[S,T] × T → Map[S,T];
           dom : Map[S,T] → Set[S];
           range : Map[S,T] → Set[T];
           __∪__ : Map[S,T] × Map[S,T] →? Map[S,T]
     ∀ M, N, O: Map[S,T]; s, s1, s2: S; t, t1, t2: T; e: Entry[S,T];
     X: Set[S]; Y: Set[T]

     %% axioms concerning predicates
     • isEmpty(M) ⇔ M = empty                          %(isEmpty_def_Map)%
```

```
      • [ t / s ] ϵ M ⇔ lookup(s, M) = t              %(elemOf_def_Map)%
      • M :: X − > Y ⇔ dom(M) = X ∧ range(M) ⊆ Y
                                                       %(arrow_def_Map)%

      %% axioms concerning operations
      • M + [ t / s ] = M [ t / s ]                %(overwrite2_def_Map)%
      • empty − [ t / s ] = empty                   %(minus_empty_Map)%
      • M [ t / s ] − [ t1 / s1 ] =
        M − s when [ t / s ] = [ t1 / s1 ] else (M − [ t1 / s1 ]) + [ t / s ]
                                                %(minus_nonEmpty_Map)%
      • empty − s = empty                     %(minusSource_empty_Map)%
      • (M + e) − s =
        M − s when ∃ t: T • e = [ t / s ] else (M − s) + e
                                           %(minusSource_nonEmpty_Map)%
      • empty − t = empty                     %(minusTarget_empty_Map)%
      • (M + e) − t =
        M − source(e) − t when target(e) = t else (M − t) + e
                                           %(minusTarget_nonEmpty_Map)%
      • s ϵ dom(M) ⇔ def lookup(s, M)                  %(dom_def_Map)%
      • t ϵ range(M) ⇔ ∃ s: S • lookup(s, M) = t     %(range_def_Map)%
      • M ∪ N = O ⇔ ∀ e: Entry[S,T] • e ϵ O ⇔ e ϵ M ∨ e ϵ N
                                                       %(union_def_Map)%

      %% important laws
      • def lookup(s, M) ⇔ ∃ t: T • [ t / s ] ϵ M
                                          %(lookup_dom)% %implied
      • (M [ t1 / s ]) [ t2 / s ] = M [ t2 / s ]
                                       %(overwrite_Map)% %implied
      • (M [ t1 / s1 ]) [ t2 / s2 ] = (M [ t2 / s2 ]) [ t1 / s1 ] if ¬ s1 = s2
                                            %(comm_Map)% %implied
end

spec FINITE [sort Elem] =
      {       NAT
       then
              op     f : Nat →? Elem
              • ∀ x: Elem • ∃ n: Nat • f(n) = x           %(f_surjective)%
              • ∃ n: Nat • ∀ m: Nat • def f(m) ⇒ m < n    %(f_bounded)%
      }
      reveal Elem
end

spec TOTALMAP [FINITE [sort S]] [sort T] = %mono
      {       MAP [sort S] [sort T]
       then
```

```
          sort   TotalMap[S,T] = {M: Map[S,T]
                                  • ∀ x: S • def lookup(x, M)}
          ops    __[__/__] :
                     TotalMap[S,T] × T × S → TotalMap[S,T];
                 lookup : S × TotalMap[S,T] → T;
                 __+__ :
                     TotalMap[S,T] × Entry[S,T] → TotalMap[S,T];
                 range : TotalMap[S,T] → Set[T];
                 __∪__ :
                     TotalMap[S,T] × TotalMap[S,T] →? TotalMap[S,T]
          pred   __ϵ__ : Entry[S,T] × TotalMap[S,T]
     }
     hide Map[S,T]
end

spec GenerateBag [sort Elem] given Nat = %mono
     generated type Bag[Elem] ::= {} | __+__(Bag[Elem]; Elem)
     op     freq : Bag[Elem] × Elem → Nat
     ∀ x, y: Elem; M, N: Bag[Elem]
     • freq({}, y) = 0                              %(freq_empty_Bag)%
     • freq(M + x, y) = suc(freq(M, y)) when x = y else freq(M, y)
                                                 %(freq_nonEmpty_Bag)%
     • M = N ⇔ ∀ x: Elem • freq(M, x) = freq(N, x)   %(equality_Bag)%
end

spec Bag [sort Elem] given Nat = %mono
     GenerateBag [sort Elem]
then %def
     preds isEmpty : Bag[Elem];
           __ϵ__ : Elem × Bag[Elem];
           __⊆__ : Bag[Elem] × Bag[Elem]
     ops   __+__ : Elem × Bag[Elem] → Bag[Elem];
           __−__ : Bag[Elem] × Elem → Bag[Elem];
           __−__, __∪__, __∩__ :
               Bag[Elem] × Bag[Elem] → Bag[Elem];
           {__}(x: Elem): Bag[Elem] = {} + x   %(singleton_def_Bag)%
     %% implied operation attributes
     ops   __∪__ : Bag[Elem] × Bag[Elem] → Bag[Elem],
               assoc, comm, unit {};                          %implied
           __∩__ : Bag[Elem] × Bag[Elem] → Bag[Elem],
               assoc, comm, idem                              %implied
     ∀ n, m: Nat; x, y: Elem; M, N, O: Bag[Elem]

     %% axioms concerning predicates
     • isEmpty(M) ⇔ M = {}                          %(isEmpty_def_Bag)%
```

```
        • x ϵ M ⇔ freq(M, x) > 0                               %(elemOf_def_Bag)%
        • M ⊆ N ⇔ ∀ x: Elem • freq(M, x) ≤ freq(N, x)
                                                             %(isSubsetOf_def_Bag)%

        %% axioms concerning operations
        • x + M = M + x                                        %(addElem_def_Bag)%
        • M − x = N ⇔
          ∀ y: Elem
          • (freq(N, y) = freq(M, x) −! 1 if x = y) ∧
            (freq(N, y) = freq(M, y) if ¬ x = y)    %(removeElem_def_Bag)%
        • M − N = O ⇔
          ∀ x: Elem • freq(O, x) = freq(M, x) −! freq(N, x)
                                                             %(difference_def_Bag)%
        • M ∪ N = O ⇔ ∀ x: Elem • freq(O, x) = freq(M, x) + freq(N, x)
                                                                  %(union_def_Bag)%
        • M ∩ N = O ⇔
          ∀ x: Elem • freq(O, x) = min(freq(M, x), freq(N, x))
                                                           %(intersection_def_Bag)%
end

view CommutativeMonoid_in_Bag [sort Elem] given Nat :
        CommutativeMonoid to Bag [sort Elem] =
        sort Elem ↦ Bag[Elem], ops e ↦ {}, __*__ ↦ __∪__
end

view PartialOrder_in_Bag [sort Elem] given Nat :
        PartialOrder to Bag [sort Elem] =
        sort Elem ↦ Bag[Elem], pred __≤__ ↦ __⊆__
end

spec Array [ops min, max : Int
             • min ≤ max                         %(Cond_nonEmptyIndex)%]
            [sort Elem]
        given Int = %mono
        sort   Index = {i: Int • min ≤ i ∧ i ≤ max}
then %mono
        {       Map [sort Index] [sort Elem]
                with sort Map[Index,Elem] ↦ Array[Elem], op empty ↦ init
          then
                ops     __!__:=__ :
                             Array[Elem] × Index × Elem → Array[Elem];
                        __!__ : Array[Elem] × Index →? Elem
                ∀ A: Array[Elem]; i: Index; e: Elem
                • A ! i := e = A [ e / i ]                 %(assignment_def)%
                • A ! i = lookup(i, A)                         %(lookup_def)%
```

```
     }
     reveal sort Array[Elem], ops init, __!__, __!__:=__
then %implies
     ∀ A: Array[Elem]; i, j: Index; e, f: Elem
     • ¬ def init ! i                          %(lookup_domain1_Array)%
     • def (A ! i := e) ! i                    %(lookup_domain2_Array)%
     • (A ! i := e) ! j = e if i = j       %(lookup_assignment1_Array)%
     • (A ! i := e) ! j = A ! j if ¬ i = j
                                           %(lookup_assignment2_Array)%
end

spec GenerateBinTree [sort Elem] = %mono
     free type
          BinTree[Elem] ::= nil
                          | binTree(entry:?Elem; left:?BinTree[Elem];
                                    right:?BinTree[Elem])
end

spec BinTree [sort Elem] given Nat, Set [sort Elem] = %mono
     GenerateBinTree [sort Elem]
and
     Set [sort Elem]
then %def
     preds isEmpty, isLeaf : BinTree[Elem];
           isCompoundTree : BinTree[Elem];
           __ϵ__ : Elem × BinTree[Elem]
     ops   height : BinTree[Elem] → Nat;
           leaves : BinTree[Elem] → Set[Elem]
     ∀ x, y: Elem; T, T1, T2: BinTree[Elem]

     %% axioms concerning predicates
     • isEmpty(T) ⇔ T = nil                      %(isEmpty_def_BinTree)%
     • ¬ isLeaf(nil)                                %(isLeaf_nil_BinTree)%
     • isLeaf(binTree(x, T1, T2)) ⇔ T1 = nil ∧ T2 = nil
                                                       %(isLeaf_binTree)%
     • ¬ isCompoundTree(nil)              %(isCompoundTree_nil_BinTree)%
     • isCompoundTree(binTree(x, T1, T2)) ⇔
       ¬ isLeaf(binTree(x, T1, T2))
                                              %(isCompoundTree_binTree)%
     • ¬ x ϵ nil                                       %(eps_nil_BinTree)%
     • x ϵ binTree(y, T1, T2) ⇔ x = y ∨ x ϵ T1 ∨ x ϵ T1
                                                          %(eps_binTree)%

     %% axioms concerning operations
     • height(nil) = 0                              %(height_nil_BinTree)%
     • height(binTree(x, T1, T2)) = max(height(T1), height(T2)) + 1
                                                        %(height_BinTree)%
     • leaves(nil) = {}                                 %(leaves_nil_BinTree)%
     • leaves(binTree(x, T1, T2)) =
       { x } when isLeaf(binTree(x, T1, T2))
       else leaves(T1) ∪ leaves(T2)
                                                        %(leaves_BinTree)%
end

spec GenerateBinTree2 [sort Elem] = %mono
     free types NonEmptyBinTree2[Elem] ::=
                     leaf(entry:?Elem)
                   | binTree(left:?NonEmptyBinTree2[Elem];
                             right:?NonEmptyBinTree2[Elem]) ;
                BinTree2[Elem] ::= nil | sort NonEmptyBinTree2[Elem]
end

spec BinTree2 [sort Elem]
     given Nat, Set [sort Elem] = %mono
     GenerateBinTree2 [sort Elem]
and
     Set [sort Elem]
then %def
     preds isEmpty, isLeaf : BinTree2[Elem];
           isCompoundTree : BinTree2[Elem];
           __ϵ__ : Elem × BinTree2[Elem]
     ops   height : BinTree2[Elem] → Nat;
           leaves : BinTree2[Elem] → Set[Elem]
     ∀ x, y: Elem; T: BinTree2[Elem]; N1, N2: NonEmptyBinTree2[Elem]

     %% axioms concerning predicates
     • isEmpty(T) ⇔ T = nil                       %(isEmpty_def_BinTree2)%
     • isLeaf(leaf(x))                             %(isLeaf_leaf_BinTree2)%
     • ¬ isLeaf(binTree(N1, N2))                %(isLeaf_binTree_BinTree2)%
     • ¬ isLeaf(nil)                                %(isLeaf_nil_BinTree2)%
     • ¬ isCompoundTree(leaf(x))
                                           %(isCompoundTree_leaf_BinTree2)%
     • isCompoundTree(binTree(N1, N2))
                                        %(isCompoundTree_binTree_BinTree2)%
     • ¬ isCompoundTree(nil)               %(isCompoundTree_nil_BinTree2)%
     • x ϵ leaf(y) ⇔ x = y                          %(eps_leaf_BinTree2)%
     • x ϵ binTree(N1, N2) ⇔ x ϵ N1 ∨ x ϵ N2
                                                 %(eps_binTree_BinTree2)%
     • ¬ x ϵ nil                                     %(eps_nil_BinTree2)%
```

```
      %% axioms concerning operations
      • height(leaf(x)) = 1                        %(height_leaf_binTree2)%
      • height(binTree(N1, N2)) = max(height(N1), height(N2)) + 1
                                                %(height_binTree_BinTree2)%
      • height(nil) = 0                            %(height_nil_BinTree2)%
      • leaves(leaf(x)) = { x }                    %(leaves_leaf_BinTree2)%
      • leaves(binTree(N1, N2)) = leaves(N1) ∪ leaves(N2)
                                                %(leaves_binTree_BinTree2)%
      • leaves(nil) = {}                           %(leaves_nil_BinTree2)%
end

spec GenerateNTree [sort Elem] = %mono
      free types
            List[NTree[Elem]] ::= []
                                 | __::__(first:?NTree[Elem];
                                          rest:?List[NTree[Elem]]) ;
            NTree[Elem] ::= nil
                           | nTree(entry:?Elem;
                                   branches:?List[NTree[Elem]])
end

spec NTree [sort Elem] given Nat, Set [sort Elem] = %mono
      GenerateNTree [sort Elem]
and
      Set [sort Elem]
and
      List [sort NTree[Elem]]
then %def
      preds isEmpty, isLeaf : NTree[Elem];
            isCompoundTree : NTree[Elem];
            __ϵ__ : Elem × NTree[Elem];
            __is__branching : NTree[Elem] × Nat
      ops   height : NTree[Elem] → Nat;
            maxHeight : List[NTree[Elem]] → Nat;
            leaves : NTree[Elem] → Set[Elem];
            allLeaves : List[NTree[Elem]] → Set[Elem]
      ∀ x, y: Elem; T: NTree[Elem]; L: List[NTree[Elem]]; n: Nat

      %% axioms concerning predicates
      • isEmpty(T) ⇔ T = nil                       %(isEmpty_def_NTree)%
      • ¬ isLeaf(nil)                               %(isLeaf_nil_NTree)%
      • isLeaf(nTree(x, L)) ⇔ ∀ T: NTree[Elem] • T ϵ L ⇒ T = nil
                                                        %(isLeaf_nTree)%
      • ¬ isCompoundTree(nil)               %(isCompoundTree_nil_NTree)%
      • isCompoundTree(nTree(x, L)) ⇔ ¬ isLeaf(nTree(x, L))
```

```
                                                   %(isCompoundTree_nTree)%
     • ¬ x ϵ nil                                            %(eps_nil_NTree)%
     • x ϵ nTree(y, L) ⇔ x = y ∨ (∃ T: NTree[Elem] • T ϵ L ∧ x ϵ T)
                                                                %(eps_nTree)%
     • nil is n branching                            %(isKbranching_nil_NTree)%
     • nTree(x, L) is n branching ⇔
       L = [ ] ∨
       (♯ L = n ∧ (∀ T: NTree[Elem] • T ϵ L ⇒ T is n branching))
                                                   %(isKbranching_nTree_NTree)%

     %% axioms concerning operations
     • height(nil) = 0                                     %(height_nil_NTree)%
     • height(nTree(x, L)) = maxHeight(L) + 1                  %(height_nTree)%
     • maxHeight([ ]) = 0                                      %(maxHeight_nil)%
     • maxHeight(T :: L) = max(maxHeight(L), height(T))
                                                 %(maxHeight_nonEmptyList)%
     • leaves(nil) = {}                                    %(leaves_nil_NTree)%
     • leaves(nTree(x, L)) =
       { x } when isLeaf(nTree(x, L)) else allLeaves(L)  %(leaves_nTree)%
     • allLeaves([ ]) = {}                                     %(allLeaves_nil)%
     • allLeaves(T :: L) = allLeaves(L) ∪ leaves(T)
                                                  %(allLeaves_nonEmptyList)%
end

spec GenerateNTree2 [sort Elem] = %mono
     free types
          NonEmptyList[NonEmptyNTree2[Elem]] ::=
               __::__(first:NonEmptyNTree2[Elem];
                         rest:List[NonEmptyNTree2[Elem]]) ;
          List[NonEmptyNTree2[Elem]] ::=
               []
             | sort NonEmptyList[NonEmptyNTree2[Elem]] ;
          NonEmptyNTree2[Elem] ::=
               leaf(entry:?Elem)
             | nTree(branches:?NonEmptyList[NonEmptyNTree2[Elem]]) ;
          NTree2[Elem] ::= nil | sort NonEmptyNTree2[Elem]
end

spec NTree2 [sort Elem] given Nat, Set [sort Elem] = %mono
     GenerateNTree2 [sort Elem]
and
     Set [sort Elem]
and
     List [sort NonEmptyNTree2[Elem]]
then %def
```

preds $\mathit{isEmpty}, \mathit{isLeaf} : \mathit{NTree2}[\mathit{Elem}]$;
$\mathit{isCompoundTree} : \mathit{NTree2}[\mathit{Elem}]$;
$\_\_\epsilon\_\_ : \mathit{Elem} \times \mathit{NTree2}[\mathit{Elem}]$;
$\_\_\mathit{is}\_\_\mathit{branching} : \mathit{NTree2}[\mathit{Elem}] \times \mathit{Nat}$

ops $\mathit{branches} :$
$\mathit{NTree2}[\mathit{Elem}] \to \mathit{NonEmptyList}[\mathit{NonEmptyNTree2}[\mathit{Elem}]]$;
$h : \mathit{List}[\mathit{NonEmptyNTree2}[\mathit{Elem}]] \times \mathit{Nat} \to \mathit{Nat}$;
%% helps defining height
$\mathit{height} : \mathit{NTree2}[\mathit{Elem}] \to \mathit{Nat}$;
$l : \mathit{List}[\mathit{NonEmptyNTree2}[\mathit{Elem}]] \to \mathit{Set}[\mathit{Elem}]$;
%% helps defining leaves
$\mathit{leaves} : \mathit{NTree2}[\mathit{Elem}] \to \mathit{Set}[\mathit{Elem}]$

$\forall\ x, y\colon \mathit{Elem};\ n\colon \mathit{Nat};\ T\colon \mathit{NTree2}[\mathit{Elem}]$;
$\mathit{NT}\colon \mathit{NonEmptyNTree2}[\mathit{Elem}]$;
$L\colon \mathit{NonEmptyList}[\mathit{NonEmptyNTree2}[\mathit{Elem}]]$;
$K\colon \mathit{List}[\mathit{NonEmptyNTree2}[\mathit{Elem}]]$

%% axioms concerning predicates
- $\mathit{isEmpty}(T) \Leftrightarrow T = \mathit{nil}$ %(isEmpty_def_NTree2)%
- $\mathit{isLeaf}(\mathit{leaf}(x))$ %(isLeaf_leaf_NTree2)%
- $\neg\ \mathit{isLeaf}(\mathit{nTree}(L))$ %(isLeaf_nTree_NTree2)%
- $\neg\ \mathit{isLeaf}(\mathit{nil})$ %(isLeaf_nil_NTree2)%
- $\neg\ \mathit{isCompoundTree}(\mathit{leaf}(x))$ %(isCompoundTree_leaf_NTree2)%
- $\mathit{isCompoundTree}(\mathit{nTree}(L))$
  %(isCompoundTree_nTree_NTree2)%
- $\neg\ \mathit{isCompoundTree}(\mathit{nil})$ %(isCompoundTree_nil_NTree2)%
- $x \ \epsilon\ \mathit{leaf}(y) \Leftrightarrow x = y$ %(eps_leaf_NTree2)%
- $x \ \epsilon\ \mathit{nTree}(L) \Leftrightarrow \exists\ T\colon \mathit{NonEmptyNTree2}[\mathit{Elem}] \bullet T \ \epsilon\ L \wedge x \ \epsilon\ T$
  %(eps_nTree_NTree2)%
- $\neg\ x \ \epsilon\ \mathit{nil}$ %(eps_nil_NTree2)%
- $\mathit{leaf}(x)\ \mathit{is}\ n\ \mathit{branching}$ %(isKbranching_leaf_NTree2)%
- $\mathit{nTree}(L)\ \mathit{is}\ n\ \mathit{branching} \Leftrightarrow$
  $\sharp\ L = n \wedge$
  $(\forall\ T\colon \mathit{NonEmptyNTree2}[\mathit{Elem}] \bullet T \ \epsilon\ L \Rightarrow T\ \mathit{is}\ n\ \mathit{branching})$
  %(isKbranching_nTree_NTree2)%
- $\mathit{nil}\ \mathit{is}\ n\ \mathit{branching}$ %(isKbranching_nil_NTree2)%

%% axioms concerning operations
- $\neg\ \mathit{def}\ \mathit{branches}(\mathit{nil})$ %(branches_nil_nTree2)%
- $h([\,], n) = n$ %(h_nil_nTree2)%
- $h(\mathit{NT} :: K, n) = \mathit{max}(\mathit{height}(\mathit{NT}), h(K, n))$ %(n_cons_nTree2)%
- $\mathit{height}(\mathit{leaf}(x)) = \mathit{1}$ %(height_leaf_nTree2)%
- $\mathit{height}(\mathit{nTree}(L)) = h(L, \mathit{0}) + \mathit{1}$ %(height_nTree_nTree2)%
- $\mathit{height}(\mathit{nil}) = \mathit{0}$ %(height_nil_nTree2)%
- $l([\,]) = \{\}$ %(l_nil_nTree2)%

- $l(NT :: K) = leaves(NT) \cup l(K)$ %(l_cons_nTree2)%
- $leaves(leaf(x)) = \{\ x\ \}$ %(leaves_leaf_nTree2)%
- $leaves(nTree(L)) = l(L)$ %(leaves_nTree_nTree2)%
- $leaves(nil) = \{\}$ %(leaves_nil_nTree2)%

end

spec KTREE [**sort** *Elem*] [**op** $k : Nat$]
  given NAT, SET [**sort** *Elem*] = **%mono**
  NTREE [**sort** *Elem*]
then **%mono**
  sort $KTree[Elem] = \{T: NTree[Elem] \bullet T\ is\ k\ branching\}$
end

spec KTREE2 [**sort** *Elem*] [**op** $k : Nat$]
  given NAT, SET [**sort** *Elem*] = **%mono**
  NTREE2 [**sort** *Elem*]
then **%mono**
  sort $KTree2[Elem] = \{T: NTree2[Elem] \bullet T\ is\ k\ branching\}$
end

7

Library Basic/Graphs

```
library BASIC/GRAPHS version 1.0

%authors : T. Mossakowski <till@tzi.de>, M. Roggenbach, L. Schröder
%date : 18 December 2003
%% with contributions from Klaus Lüttich
%{ We construct directed graphs inductively by successively adding
   nodes and edges.
   Ids of nodes must be unique.
   Ids of edges between two given nodes must be unique as well.
   If you need multiple edges with the same label, take a sort
   isomorphic to a product (e.g. Label x Int) as EdgeId. }%
%display(__::__−− >__isIn__ %LATEX __::__⟶__ϵ__)%
%display(__isIn__ %LATEX __ϵ__)%


from BASIC/STRUCTUREDDATATYPES get SET, MAP, LIST

from BASIC/NUMBERS get NAT

spec GRAPH [sort NodeId] [sort EdgeId] =
     generated type
               Graph ::= emptyGraph
                       | addNode(NodeId; Graph)
                       | addEdge(NodeId; NodeId; EdgeId; Graph)?
     ops   source, target : EdgeId × Graph →? NodeId
     preds __ϵ__ : NodeId × Graph;
           __ϵ__ : EdgeId × Graph;
           __::__⟶__ϵ__ : EdgeId × NodeId × NodeId × Graph
     ∀ n, n1, s, t, s1, t1, s2, t2: NodeId; e, e1, e2: EdgeId; g, g′: Graph
     • def addEdge(s, t, e, g) ⇔ ¬ e ϵ g              %(dom_addEdge)%
     • ¬ n ϵ emptyGraph                            %(isIn_emptyGraph)%
```

$\bullet\ n \epsilon \mathit{addNode}(n1, g) \Leftrightarrow n = n1 \vee n \epsilon g$ %(isIn_addNode)%

$\bullet\ n \epsilon \mathit{addEdge}(s, t, e, g) \Leftrightarrow n = s \vee n = t \vee n \epsilon g$
%(isIn_addEdge)%

$\bullet\ \neg\ e \epsilon \mathit{emptyGraph}$ %(isIn_emptyGraph)%

$\bullet\ e \epsilon \mathit{addNode}(n, g) \Leftrightarrow e \epsilon g$ %(isIn_addNode)%

$\bullet\ e1 \epsilon \mathit{addEdge}(s2, t2, e2, g) \Leftrightarrow e1 = e2 \vee e1 \epsilon g$ %(isIn_addEdge)%

$\bullet\ \neg\ \mathit{def}\ \mathit{source}(e, \mathit{emptyGraph})$ %(source_empty)%

$\bullet\ \mathit{source}(e, \mathit{addNode}(n, g)) = \mathit{source}(e, g)$ %(source_addNode)%

$\bullet\ \mathit{source}(e1, \mathit{addEdge}(s, t, e2, g)) =$
$s\ \mathit{when}\ e1 = e2\ \mathit{else}\ \mathit{source}(e1, g)$ %(source_addEdge)%

$\bullet\ \neg\ \mathit{def}\ \mathit{target}(e, \mathit{emptyGraph})$ %(target_empty)%

$\bullet\ \mathit{target}(e, \mathit{addNode}(n, g)) = \mathit{target}(e, g)$ %(target_addNode)%

$\bullet\ \mathit{target}(e1, \mathit{addEdge}(s, t, e2, g)) =$
$t\ \mathit{when}\ e1 = e2\ \mathit{else}\ \mathit{target}(e1, g)$ %(target_addEdge)%

$\bullet\ e :: s \longrightarrow t \epsilon g \Leftrightarrow$
$e \epsilon g \wedge \mathit{source}(e, g) = s \wedge \mathit{target}(e, g) = t$ %(isIn_def)%

$\bullet\ g = g' \Leftrightarrow$
$(\forall\ n: \mathit{NodeId} \bullet n \epsilon g \Leftrightarrow n \epsilon g') \wedge$
$(\forall\ e: \mathit{EdgeId} \bullet e \epsilon g \Leftrightarrow e \epsilon g') \wedge$
$(\forall\ e: \mathit{EdgeId} \bullet \mathit{source}(e, g) = \mathit{source}(e, g')) \wedge$
$(\forall\ e: \mathit{EdgeId} \bullet \mathit{target}(e, g) = \mathit{target}(e, g'))$ %(extensionality)%

end

%% Some basic operations and predicates on graphs

spec RichGraph [**sort** *NodeId*] [**sort** *EdgeId*] =
Graph [**sort** *NodeId*] [**sort** *EdgeId*]

then %def

ops $\mathit{removeNode} : \mathit{NodeId} \times \mathit{Graph} \to \mathit{Graph};$
$\mathit{removeEdge} : \mathit{EdgeId} \times \mathit{Graph} \to \mathit{Graph}$

$\forall\ n, n1, n2: \mathit{NodeId};\ e, e1, e2: \mathit{EdgeId};\ g, g': \mathit{Graph}$

$\bullet\ \mathit{removeNode}(n, \mathit{emptyGraph}) = \mathit{emptyGraph}$
%(removeNode_emtpyGraph)%

$\bullet\ \mathit{removeNode}(n, \mathit{addNode}(n1, g)) =$
$\mathit{removeNode}(n, g)$
$\mathit{when}\ n = n1\ \mathit{else}\ \mathit{addNode}(n1, \mathit{removeNode}(n, g))$
%(removeNode_addNode)%

$\bullet\ \mathit{removeNode}(n, \mathit{addEdge}(n1, n2, e, g)) =$
$\mathit{removeNode}(n, g)$
$\mathit{when}\ n = n1 \vee n = n2$
$\mathit{else}\ \mathit{addEdge}(n1, n2, e, \mathit{removeNode}(n, g))$
%(removeNode_addEdge)%

$\bullet\ \mathit{removeEdge}(e, \mathit{emptyGraph}) = \mathit{emptyGraph}$
%(removeEdge_emtpyGraph)%

$\bullet\ \mathit{removeEdge}(e, \mathit{addNode}(n1, g)) = \mathit{addNode}(n1, \mathit{removeEdge}(e, g))$
%(removeEdge_addNode)%

```
        • removeEdge(e, addEdge(n1, n2, e1, g)) =
          removeEdge(e, g)
          when e = e1 else addEdge(n1, n2, e1, removeEdge(e, g))
                                                        %(removeEdge_addEdge)%
        pred  symmetric(g: Graph) ⇔
              ∀ n1, n2: NodeId; e: EdgeId
              • e :: n1 ⟶ n2 ϵ g ⇒
                (∃ e′: EdgeId • e′ :: n2 ⟶ n1 ϵ g)         %(symmetric_def)%
        pred  transitive(g: Graph) ⇔
              ∀ n1, n2, n3: NodeId; e1, e2: EdgeId
              • e1 :: n1 ⟶ n2 ϵ g ∧
                e2 :: n2 ⟶ n3 ϵ g ⇒
                (∃ e3: EdgeId • e2 :: n1 ⟶ n3 ϵ g)        %(transitive_def)%
        pred  loopFree(g: Graph) ⇔
              ¬ ∃ n: NodeId; e: EdgeId • e :: n ⟶ n ϵ g  %(loopFree_def)%
        pred  simple(g: Graph) ⇔
              ∀ e1, e2: EdgeId; s, t: NodeId
              • e1 :: s ⟶ t ϵ g ∧ e2 :: s ⟶ t ϵ g ⇒
                e1 = e2                                      %(simple_def)%
        pred  __subgraphOf__(g1, g2: Graph) ⇔
              (∀ n: NodeId • n ϵ g1 ⇒ n ϵ g2) ∧
              (∀ n1, n2: NodeId; e: EdgeId
               • e :: n1 ⟶ n2 ϵ g1 ⇒
                 e :: n1 ⟶ n2 ϵ g2)                      %(subgraphOf_def)%
        pred  complete(g: Graph) ⇔
              ∀ n1, n2: NodeId
              • n1 ϵ g ∧ n2 ϵ g ⇒ (∃ e: EdgeId • e :: n1 ⟶ n2 ϵ g)
                                                            %(complete_def)%
        pred  __cliqueOf__(g1, g2: Graph) ⇔
              g1 subgraphOf g2 ∧ complete(g1)               %(cliqueOf_def)%
        pred  __maxCliqueOf__(g1, g2: Graph) ⇔
              g1 cliqueOf g2 ∧
              (∀ g3: Graph
               • g1 subgraphOf g3 ∧ g3 cliqueOf g2 ⇒ g1 = g3)
                                                        %(max_cliqueOf_def)%
end

%% Mapping our representation to a set-based one
spec GraphToSet [sort NodeId] [sort EdgeId] =
        Graph [sort NodeId] [sort EdgeId]
and
%% The following also imports Set[EdgeId] and Set[NodeId]
        Map [sort EdgeId] [sort NodeId]
then %def
        ops   nodeSet : Graph → Set[NodeId];
```

```
            sourceMap, targetMap : Graph → Map[EdgeId,NodeId]
      ∀ g: Graph; n: NodeId; e: EdgeId
      • n ϵ nodeSet(g) ⇔ n ϵ g                              %(nodeSet_def)%
      • [ n / e ] ϵ sourceMap(g) ⇔ source(e, g) = n      %(sourceMap_def)%
      • [ n / e ] ϵ targetMap(g) ⇔ target(e, g) = n      %(targetMap_def)%
then %def
      ops   edgeSet : Graph → Set[EdgeId];
            successors, predecessors : NodeId × Graph → Set[NodeId];
            inEdges, outEdges : NodeId × Graph → Set[EdgeId];
            inDegree, outDegree : NodeId × Graph → Nat
      ∀ n, n1, n2: NodeId; e: EdgeId; g: Graph
      • e ϵ edgeSet(g) ⇔ e ϵ g                              %(edgeSet_def)%
      • n2 ϵ successors(n1, g) ⇔
        ∃ e: EdgeId • e :: n1 ⟶ n2 ϵ g                   %(successors_def)%
      • n1 ϵ predecessors(n2, g) ⇔
        ∃ e: EdgeId • e :: n1 ⟶ n2 ϵ g                 %(predecessors_def)%
      • e ϵ inEdges(n1, g) ⇔
        ∃ n2: NodeId • e :: n2 ⟶ n1 ϵ g                     %(inEdges_def)%
      • e ϵ outEdges(n1, g) ⇔
        ∃ n2: NodeId • e :: n1 ⟶ n2 ϵ g                    %(outEdges_def)%
      • inDegree(n, g) = ♯ inEdges(n, g)                    %(inDegree_def)%
      • outDegree(n, g) = ♯ outEdges(n, g)                %(outDegree_def)%
end

%% The subsort of symmetric (= undirected) graphs
spec SymmetricGraph [sort NodeId] [sort EdgeId] =
      Graph [sort NodeId] [sort EdgeId]
then
      sort  SymmetricGraph = {g: Graph
                                 • ∀ s, t: NodeId; e: EdgeId
                                   • e :: s ⟶ t ϵ g ⇔
                                     e :: t ⟶ s ϵ g}
                                                     %(SymmetricGraph_def)%
      type  SymmetricGraph ::= emptyGraph
                             | addNode(node:?NodeId;
                                       graph:?SymmetricGraph)
                             | addEdgeSym(source, target:?NodeId;
                                          edge:?EdgeId;
                                          graph:?SymmetricGraph)
      preds __ϵ__ : NodeId × SymmetricGraph;
            __::__⟶__ϵ__ : EdgeId × NodeId × NodeId ×
            SymmetricGraph
      ∀ s, t: NodeId; e: EdgeId; g: SymmetricGraph
      • addEdgeSym(s, t, e, g) =
        addEdge(s, t, e, addEdge(t, s, e, g)) as SymmetricGraph
                                                          %(addEdge_def)%
      %% the other operations and predicates are determined
      %% by the overloading relations
end

spec SymmetricClosure [sort NodeId] [sort EdgeId] =
      RichGraph [sort NodeId] [sort EdgeId]
then
      op      sc : Graph → Graph
      ∀ n, n1, n2: NodeId; e: EdgeId; g: Graph
      • n ϵ sc(g) ⇔ n ϵ g                                   %(sc_def_1)%
      • e :: n1 ⟶ n2 ϵ sc(g) ⇔
        e :: n1 ⟶ n2 ϵ g ∨ e :: n2 ⟶ n1 ϵ g
                                                            %(sc_def_2)%
then %implies
      ∀ g: Graph
      • symmetric(sc(g))                                 %(symmetric_sc)%
end

%% Various things defined in terms of paths and transitive closure
spec Paths [sort NodeId] [sort EdgeId] =
      RichGraph [sort NodeId] [sort EdgeId]
and
      List [sort EdgeId]
and
      SymmetricClosure [sort NodeId] [sort List[EdgeId]]
      with sort Graph ↦ PathGraph
then
      ops     source, target : List[EdgeId] × Graph →? NodeId;
            tc, stc : Graph → PathGraph
      preds   __pathOf__ : List[EdgeId] × Graph;
            __pathSubgraphOf__ : Graph × PathGraph;
            pathTransitive : PathGraph
      ∀ n, n1, n2: NodeId; e: EdgeId; p: List[EdgeId]; g: Graph;
      g′: PathGraph
      • source(p, g) = source(first(p), g)
      • target(p, g) = target(last(p), g)
      • g pathSubgraphOf g′ ⇔
        (∀ n: NodeId • n ϵ g ⇔ n ϵ g′) ∧
        (∀ n1, n2: NodeId; e: EdgeId
         • e :: n1 ⟶ n2 ϵ g ⇔ [ e ] :: n1 ⟶ n2 ϵ g′)
                                                   %(pathSubgraphOf_def)%
```

```
• pathTransitive(g′) ⇔
  ∀ n1, n2, n3: NodeId; e1, e2: List[EdgeId]
  • e1 :: n1 ⟶ n2 ϵ g′ ∧
    e2 :: n2 ⟶ n3 ϵ g′ ⇒
    (e1 ++ e2) :: n1 ⟶ n3 ϵ g′                %(pathTransitive_def)%
• pathTransitive(tc(g))                                   %(tc_def1)%
• g pathSubgraphOf tc(g)                                  %(tc_def2)%
• g pathSubgraphOf g′ ∧ pathTransitive(g′) ⇒ tc(g) subgraphOf g′
                                                          %(tc_def3)%
• p pathOf g ⇔ p ϵ tc(g)                              %(pathOf_def)%
• stc(g) = sc(tc(g))                                      %(stc_def)%
%% Connectedness and acyclity can be elegantly expressed
%% using (symmetric) transitive closure
pred  connected(g: Graph) ⇔ complete(stc(g))     %(connected_def)%
pred  stronglyConnected(g: Graph) ⇔ complete(tc(g))
                                          %(stronglyconnected_def)%
pred  acyclic(g: Graph) ⇔ loopFree(tc(g))            %(acyclic_def)%
pred  hasCycle(g: Graph) ⇔ ¬ acyclic(g)            %(hasCycle_def)%
%% A tree is an acyclic graph with a root node, such that each
%% other node is reachable via a unique path from the root
pred  tree(g: Graph) ⇔
        acyclic(g) ∧
        (∃ root: NodeId
         • root ϵ g ∧
           (∀ n: NodeId
            • n ϵ g ∧ ¬ n = root ⇒
              (∃! p: List[EdgeId]
               • p pathOf g ∧ source(p, g) = root ∧ target(p, g) = n)))
                                                         %(tree_def1)%
%% A spanning tree is a tree subgraph that has the same nodes
pred  __spanningTreeOf__(t, g: Graph) ⇔
        tree(t) ∧ t subgraphOf g ∧ (∀ n: NodeId • n ϵ g ⇒ n ϵ t)
                                                %(spanning_tree_def)%
%% a symmetric graph is hasCycle iff there is a path that is a cycle
%% and that does not contain a two-loop
pred  symmetricHasCycle(g: Graph) ⇔
        symmetric(g) ∧
        (∃ p: List[EdgeId]; e: EdgeId
         • p pathOf g ∧
           freq(p, e) = 2 ∧
           ¬ ∃ e1, e2: EdgeId; n1, n2: NodeId
             • e1 ϵ p ∧
               e2 ϵ p ∧
               e1 :: n1 ⟶ n2 ϵ g ∧
               e2 :: n2 ⟶ n1 ϵ g)          %(symmetricHasCycle_def)%
```

```
      pred   symmetricAcyclic(g: Graph) ⇔
             symmetric(g) ∧ ¬ symmetricHasCycle(g)
                                                %(symmetricAcyclic_def)%
      pred   symmetricTree(g: Graph) ⇔
             symmetricAcyclic(g) ∧ connected(g)     %(symmetricTree_def)%
then %implies
%% finite trees are finite acyclic connected graphs with no
%% occurrences of –> * <–
      pred   tree(g: Graph) ⇔
             acyclic(g) ∧
             connected(g) ∧
             ¬ ∃ e1, e2: EdgeId; n1, n2, n3: NodeId
               • e1 :: n1 ⟶ n2 ϵ g ∧
                 e2 :: n3 ⟶ n2 ϵ g
                                                        %(tree_def2)%
end

spec GraphColorability [sort NodeId] [sort EdgeId]
      given Nat =
      GraphToSet [sort NodeId] [sort EdgeId]
and
      Map [sort NodeId]  [sort Nat]
then
      pred   __is__colorable(g: Graph; n: Nat) ⇔
             ∃ f: Map[NodeId,Nat]
             • dom(f) = nodeSet(g) ∧
               (∀ m: Nat • m ϵ range(f) ⇒ m < n) ∧
               (∀ e: EdgeId; s, t: NodeId
                • e :: s ⟶ t ϵ g ⇒ ¬ lookup(s, f) = lookup(t, f))
                                                      %(colorable_def)%
      pred   bipartite(g: Graph) ⇔ g is 2 colorable       %(bipartite_def)%
end

%% Shortest paths in graphs having weights for their edges
%% Note that the weight function is given globally
%% If edge Ids should be e.g. integers, then EdgeId should
%% consists of pairs (id,weight) of integers and naturals
spec ShortestPaths [sort NodeId]
                   [sort EdgeId
                    op weight : EdgeId → Nat]
      given Nat =
      Paths [sort NodeId] [sort EdgeId]
then
      ops    distance : List[EdgeId] → Nat;
             shortestPath : NodeId × NodeId × Graph →? List[EdgeId]
```

```
     ∀ s, t: NodeId; e: EdgeId; p: List[EdgeId]; g: Graph
     • distance([ ]) = 0                                      %(distance_nil)%
     • distance(e :: p) = weight(e) + distance(p)            %(distance_cons)%
     • def shortestPath(s, t, g) ⇔
       ∃ p: List[EdgeId]
       • p pathOf g ∧ source(p, g) = s ∧ target(p, g) = t
                                                          %(shortestPath_dom)%
     • def shortestPath(s, t, g) ∧
       p pathOf g ∧
       source(p, g) = s ∧
       target(p, g) = t ⇒
       distance(shortestPath(s, t, g)) ≤ distance(p)   %(shortestPath_def)%
end

spec GraphHomomorphism [sort N1] [sort E1] [sort N2] [sort E2] =
     Graph [sort N1] [sort E1] with Graph ↦ Graph1
and
     Graph [sort N2] [sort E2] with Graph ↦ Graph2
and
     Map [sort N1] [sort N2]
and
     Map [sort E1] [sort E2]
then
     free type
          PreHom ::= preHom(source:Graph1; target:Graph2;
                        nodeMap:Map[N1,N2]; edgeMap:Map[E1,E2])
     sort   Hom = {h: PreHom
                        • ∀ n, n′: N1; e: E1
                        • e :: n ⟶ n′ ϵ source(h) ⇒
                          lookup(e, edgeMap(h)) :: lookup(n, nodeMap(h))
                          ⟶ lookup(n′, nodeMap(h)) ϵ target(h)}
                                                                 %(Hom_def)%
end

%% A minor of a graph is something that can be homomorphically
%% mapped to the transitive closure
spec Minor [sort N1] [sort E1] [sort N2] [sort E2] =
     Paths [sort N2] [sort E2] with Graph ↦ Graph2
and
     GraphHomomorphism [sort N1] [sort E1] [sort N2]
                                  [sort List[E2]]
     with Graph2 ↦ PathGraph
then
```

```
    pred  __minorOf__(g1: Graph1; g2: Graph2) ⇔
          ∃ h: Hom • source(h) = g1 ∧ target(h) = stc(g2)
                                                      %(minorOf_def)%
end

%% The complete graph over five nodes
spec K5 =
    free types
        Five ::= 1 | 2 | 3 | 4 | 5 ;
        FivePair ::= pair(Five; Five)
then
    RichGraph [sort Five] [sort FivePair]
then
    op    k5 : Graph
    ∀ n, n1, n2: Five
    • simple(k5)                                       %(k5_simple)%
    • n ϵ k5                                           %(k5_def_1)%
    • pair(n1, n2) :: n1 ⟶ n2 ϵ k5                     %(k5_def_2)%
end

%% The graph consisting of two copies of three nodes,
%% such that two nodes are linked by an edge iff they stem
%% from different copies
spec K3_3 =
    free type Three ::= 1 | 2 | 3
    free type Three2 ::= left(Three) | right(Three)
    free type ThreePair ::= pair(Three; Three)
then
    RichGraph [sort Three2] [sort ThreePair]
then
    op    k3_3 : Graph
    ∀ n, n1, n2: Three
    • simple(k3_3)
    • left(n) ϵ k3_3                                   %(k3_3_def_1)%
    • right(n) ϵ k3_3                                  %(k3_3_def_2)%
    • pair(n1, n2) :: left(n1) ⟶ right(n2) ϵ k3_3
                                                       %(k3_3_def_3)%
end

%% planar graphs defined using the Kuratowski characterization:
%% K5 and K3_3 must not occur as minors
spec Planar [sort NodeId] [sort EdgeId] =
    K5 with Graph ↦ Graph5
and
    K3_3 with Graph ↦ Graph3_3
```

```
and
      Minor [sort Five] [sort FivePair] [sort NodeId] [sort EdgeId]
      with Graph1 ↦ Graph5, Graph2 ↦ Graph
and
      Minor [sort Three2] [sort ThreePair] [sort NodeId] [sort EdgeId]
      with Graph1 ↦ Graph3_3, Graph2 ↦ Graph
then
      pred  planar(g: Graph) ⇔ ¬ k5 minorOf g ∧ ¬ k3_3 minorOf g
                                                              %(planar_def)%
end

%% Graphs with edge labels that need not be unique
%% The trick is to make them unique by adding the source and target node
spec NonUniqueEdgesGraph [sort NodeId] [sort EdgeLabel] =
      free type EdgeId ::= EI(NodeId; EdgeLabel; NodeId)
then
      RichGraph [sort NodeId] [sort EdgeId]
then
      ops    addEdge :
                       NodeId × NodeId × EdgeLabel × Graph → Graph;
             removeEdgeLabel : EdgeLabel × Graph → Graph
      preds  __ϵ__ : EdgeLabel × Graph;
             __::__⟶__ϵ__ :
                EdgeLabel × NodeId × NodeId × Graph
      ∀ n, n1, n2: NodeId; el, el′: EdgeLabel; g: Graph
      • addEdge(n1, n2, el, g) = addEdge(n1, n2, EI(n1, el, n2), g)
                                                          %(addEdgeNU_def)%
      • el′ ϵ removeEdgeLabel(el, g) ⇔ el′ ϵ g ∧ ¬ el = el′
                                                     %(removeEdgeLabel_def1)%
      • n ϵ removeEdgeLabel(el, g) ⇔ n ϵ g       %(removeEdgeLabel_def2)%
      • el ϵ g ⇔ ∃ n1, n2: NodeId • EI(n1, el, n2) ϵ g     %(isInNU1_def)%
      • el :: n1 ⟶ n2 ϵ g ⇔ EI(n1, el, n2) ϵ g              %(isInNU2_def)%
end
```

8

Library Basic/Algebra_II

```
library Basic/Algebra_II version 1.0

%authors : L. Schröder <lschrode@tzi.de>, M. Roggenbach, T. Mossakowski
%date : 21 May 2003
%prec({__∗__} < {__ˆ__})%
%prec({__+__, __−__} < {__/__, __∗__})%
%left_assoc(__+__, __∗__, __ˆ__)%


from Basic/RelationsAndOrders get
      PreOrder, TotalOrder, EquivalenceRelation

from Basic/Algebra_I get
      Monoid, CommutativeMonoid, Group, CommutativeRing,
      IntegralDomain, RichIntegralDomain, Field, ExtMonoid,
      ExtGroup, ExtCommutativeRing

from Basic/Numbers get Nat, Int

from Basic/StructuredDatatypes get List, Bag

spec EuclidianRing =
      IntegralDomain
and
      Nat reveal pred __<__
then
      op    delta : Elem →? Nat
      ∀ a, b: Elem
      • def delta(a) if ¬ a = 0                    %(delta_dom_ER)%
```

```
     • (∃ q, r: Elem • a = q ∗ b + r ∧ (r = 0 ∨ delta(r) < delta(b)))
       if ¬ b = 0                                            %(div_ER)%
end

spec ConstructFactorialRing =
     RichIntegralDomain
     with sorts RUnit[Elem], Irred[Elem], pred associated
then %mono
     Bag [sort Irred[Elem]]
     with sort Bag[Irred[Elem]] ↦ Factors[Elem]
then %def
     pred  equivalent : Factors[Elem] × Factors[Elem]
     op    prod : Factors[Elem] → Elem
     ∀ i, j: Irred[Elem]; S, T: Factors[Elem]
     • prod({}) = e                               %(prod_empty_CFR)%
     • prod(S + i) = prod(S) ∗ i                   %(prod_plus_CFR)%
     • equivalent(S, T) ⇔
       (S = {} ∧ T = {}) ∨
       (∃ s, t: Irred[Elem]
        • s ϵ S ∧ t ϵ T ∧ associated(s, t) ∧ equivalent(S − s, T − t))
                                               %(equivalent_def_CFR)%
then
     ∀ x: Elem; S, T: Factors[Elem]
     • ∃ V: Factors[Elem] • x = prod(V)            %(existsFact_CFR)%
     • associated(prod(S), prod(T)) ⇒ equivalent(S, T)
                                                   %(uniqueFact_CFR)%
end

spec FactorialRing =
     ConstructFactorialRing
     reveal sort Elem,
              ops __+__ : Elem × Elem → Elem,
                  __∗__ : Elem × Elem → Elem, 0 : Elem, e : Elem
end

spec IntInfinity =
     Int
then
     {      free type IntInfty ::= sort Int | infty | negInfty
            ops    __+__, __∗__ : IntInfty × IntInfty →? IntInfty,
                      comm;
                   __−__ : IntInfty × IntInfty →? IntInfty;
                   −__ : IntInfty → IntInfty
            preds __<__, __≤__ : IntInfty × IntInfty
      then
```

```
            ∀ n: Int; m, k: IntInfty
            • − infty = negInfty                          %(neg_def1_II)%
            • − negInfty = infty                          %(neg_def2_II)%
            • negInfty ≤ m                                %(leq_def1_II)%
            • m ≤ infty                                   %(leq_def2_II)%
            • m ≤ negInfty ⇒ m = negInfty                 %(leq_def3_II)%
            • infty ≤ m ⇒ infty = m                       %(leq_def4_II)%
            • m < k ⇔ m ≤ k ∧ ¬ m = k                      %(less_def_II)%
            • infty + n = infty                           %(add_def1_II)%
            • infty + infty = infty                       %(add_def2_II)%
            • ¬ def infty + negInfty                      %(add_def3_II)%
            • negInfty + k = − (infty + − k)              %(add_def4_II)%
            • 0 < m ⇒ infty ∗ m = infty                   %(mult_def1_II)%
            • ¬ def infty ∗ 0                             %(mult_def2_II)%
            • − m ∗ k = − (m ∗ k)                         %(mult_def3_II)%
            • m − k = m + − k                              %(sub_def_II)%
    }
    hide negInfty
end

view TotalOrder_in_IntInfinity :
    TotalOrder to IntInfinity =
    sort Elem ↦ IntInfty
end

spec ConstructPolynomial [CommutativeRing]
    given {     Int
            then
                op      0 : Int
          } = %mono
    IntInfinity
then
    local List [sort Elem] within
      {       %% [a_0,...,a_n] is a_n * x^n + ... + a_0
              sort    Poly[Elem] = {l: List[Elem] • ¬ last(l) = 0}
        then
              sort    Elem < Poly[Elem]
              ops     X : Poly[Elem];
                      degree : Poly[Elem] → IntInfty;
                      __:::__ : Elem × Poly[Elem] → Poly[Elem];
                      __+__, __*__ :
                              Poly[Elem] × Poly[Elem] → Poly[Elem]
              ∀ a, b: Elem; p, q: Poly[Elem]
              • X = [ 0, e ]                              %(X_def_Poly)%
              • a = [ ] when a = 0 else [ a ]           %(emb_def_Poly)%
```

```
                • a ::: p = a when p = 0 else a :: p        %(cons_def_Poly)%
                • degree(p) = − infty when p = 0 else pre(♯ p)
                                                             %(degree_def_Poly)%
                • p + 0 = p                                   %(add_zero1_Poly)%
                • 0 + p = 0                                   %(add_zero2_Poly)%
                • (a ::: p) + (b ::: q) = (a + b) ::: (p + q)
                                                              %(add_cons_Poly)%
                • p ∗ 0 = 0                                  %(mult_zero1_Poly)%
                • 0 ∗ p = 0                                  %(mult_zero2_Poly)%
                • (a ::: p) ∗ (b ::: q) =
                  ((a ∗ b) ::: (b ∗ p + a ∗ q)) + (0 ::: (0 ::: (p ∗ q)))
                                                              %(mult_cons_Poly)%
          }
end

view CRing_in_CPolynomial [CommutativeRing] given Int :
      CommutativeRing to
      ConstructPolynomial [CommutativeRing] =
      sort Elem ↦ Poly[Elem]
end

spec Polynomial [CommutativeRing] given Int =
      ExtCommutativeRing [view CRing_in_CPolynomial
                                                    [CommutativeRing]]
then %implies
      ∀ p, q: Poly[Elem]
      • degree(p) ≤ degree(q) ⇒ degree(p + q) ≤ degree(q)
                                                             %(degree_add_Poly)%
      • degree(p ∗ q) ≤ (degree(p) + degree(q))       %(degree_mult1_Poly)%
      • hasNoZeroDivisors ⇒ degree(p ∗ q) = degree(p) + degree(q)
                                                            %(degree_mult2_Poly)%
end

view EuclidianRing_in_Polynomial [Field] given Int :
      EuclidianRing to
      {       Polynomial [Field]
       then
              op      natDegree : Poly[Elem] →? Nat
              ∀ p: Poly[Elem]
              • natDegree(p) = degree(p) as Nat            %(natDegree_def)%
      } =
      sort Elem ↦ Poly[Elem], op delta ↦ natDegree
end

spec MonoidAction [Monoid] =
      sort  Space
      op    __∗__ : Elem × Space → Space
      ∀ x: Space; a, b: Elem
      • e ∗ x = x                                          %(unit_MAction)%
      • a ∗ b ∗ x = a ∗ (b ∗ x)                           %(assoc_MAction)%
end

spec GroupAction [Group] =
      MonoidAction [Group]
end

spec ExtEuclidianRing [EuclidianRing] given Int = %mono
      RichIntegralDomain
end

spec ExtFactorialRing [FactorialRing] given Int =
      RichIntegralDomain
and
      ConstructFactorialRing
end

spec ExtMonoidAction [MonoidAction [Monoid]]
      given Nat = %def
      ExtMonoid [Monoid]
then
      pred  connected : Space × Space
      ∀ x, y: Space
      • connected(x, y) ⇔ ∃ a: Elem • a ∗ x = y
                                                %(connected_def_EMAction)%
end

view PreOrder_in_ExtMonoidAction [MonoidAction [Monoid]]
      given Nat :
      PreOrder to ExtMonoidAction [MonoidAction [Monoid]] =
      sort Elem ↦ Space, pred __≤__ ↦ connected
end

spec ExtGroupAction [GroupAction [Group]]
      given Int = %def
      ExtMonoidAction [GroupAction [Group]]
and
      ExtGroup [Group]
then %implies
      ∀ a, b: Elem; x, y: Space
      • x = y if a ∗ x = a ∗ y                              %(inj_EGAction)%
```

```
    • ∃ z: Space • a ∗ z = x                          %(surj_EGAction)%
end

view EqRel_in_ExtGroupAction [GroupAction [Group]]
    given Int :
    EquivalenceRelation to
    ExtGroupAction [GroupAction [Group]] =
    sort Elem ↦ Space, pred __∼__ ↦ connected
end

spec RichMonoidAction [Monoid] =
    ExtMonoidAction [MonoidAction [Monoid]]
end

view PreOrder_in_RichMonoidAction [Monoid] :
    PreOrder to RichMonoidAction [Monoid] =
    sort Elem ↦ Space, pred __≤__ ↦ connected
end

spec RichGroupAction [Group] =
    ExtGroupAction [GroupAction [Group]]
end

view EqRel_in_RichGroupAction [Group] :
    EquivalenceRelation to RichGroupAction [Group] =
    sort Elem ↦ Space, pred __∼__ ↦ connected
end

spec RichEuclidianRing =
    ExtEuclidianRing [EuclidianRing]
end

spec RichFactorialRing =
    ExtFactorialRing [FactorialRing]
end

view FactorialRing_in_ExtEuclRing [EuclidianRing]
    given Int :
    FactorialRing to ExtEuclidianRing [EuclidianRing]
end

view FactorialRing_in_RichEuclidianRing :
    FactorialRing to RichEuclidianRing
end
```

```
view EuclidianRing_in_Int :
     EuclidianRing to
     {     Int
      then
           op     1 : Int
     } =
     sort Elem ↦ Int, ops delta ↦ abs, e ↦ 1
end

view FreeMonoid_in_List [sort Elem] given Nat :
     {     sort   Generators
      then
           free {     Monoid
                 then
                      op     inject : Generators → Elem
                }
     } to
     {     List [sort Elem]
      then
           op     singleton(x: Elem): List[Elem] = [ x ]
     } =
     sorts Elem ↦ List[Elem], Generators ↦ Elem,
     ops e ↦ [], __*__ ↦ __++__, inject ↦ singleton
end

view FreeCommutativeMonoid_in_Bag [sort Elem]
     given Nat :
     {     sort   Generators
      then
           free {     CommutativeMonoid
                 then
                      op     inject : Generators → Elem
                }
     } to
     Bag [sort Elem] =
     sorts Elem ↦ Bag[Elem], Generators ↦ Elem,
     ops e ↦ {}, __*__ ↦ __∪__, inject ↦ {__}
end
```

9

Library Basic/LinearAlgebra_I

```
library Basic/LinearAlgebra_I version 1.0

%authors : L. Schröder <lschrode@tzi.de>, M. Roggenbach, T. Mossakowski
%date : 9 January 2004
%prec({__*__} < {__^__})%
%prec({__+__, __−__} < {__/__, __*__})%
%left_assoc(__+__, __*__, __^__)%


from Basic/Algebra_I get
       AbelianGroup, ExtAbelianGroup, Monoid, Group, Field,
       ExtField, RichField

from Basic/Algebra_II get
       MonoidAction, RichMonoidAction, GroupAction,
       ExtGroupAction, EuclidianRing_in_Int

from Basic/Numbers get Nat, Int

from Basic/Algebra_II get Polynomial

from Basic/StructuredDatatypes get Array, Map

spec VectorSpace [Field] =
       MonoidAction [Monoid
                             with ops e, __*__ : Elem × Elem → Elem]
       with sort Space, op __*__ : Elem × Space → Space
then
       closed {AbelianGroup
                 with sort Elem ↦ Space, ops e ↦ 0, __*__ ↦ __+__
```

```
              }
then
      ∀ x, y: Space; a, b: Elem
      • (a + b) ∗ x = a ∗ x + b ∗ x                       %(distr1_VS)%
      • a ∗ (x + y) = a ∗ x + a ∗ y                       %(distr2_VS)%
end

view AbelianGroup_in_VectorSpace [Field] :
      AbelianGroup to VectorSpace [Field] =
      sort Elem ↦ Space, ops e ↦ 0, __∗__ ↦ __+__
end

view GroupAction_in_VectorSpace [Field] :
      GroupAction [Group] to
      {       VectorSpace [RichField
                              reveal sorts Elem, NonZero[Elem],
                                    ops e, 0, __+__, __∗__]
       then
              op      __∗__ : NonZero[Elem] × Space → Space
      } =
      sort Elem ↦ NonZero[Elem]
end

view VectorSpace_in_Field [Field] :
      VectorSpace [Field] to Field =
      sort Space ↦ Elem,
      op __∗__ : Elem × Space → Space ↦
         __∗__ : Elem × Elem → Elem
end

spec VectorSpaceLC [VectorSpace [Field]] = %mono
      Map [sort Space] [sort Elem]
      with sort Map[Space,Elem] ↦ LC[Space,Elem]
      hide sorts Set[Space], Set[Elem], NonEmptySet[Space],
                 NonEmptySet[Elem]
then
      op     eval : LC[Space,Elem] → Space
      pred   isZero : LC[Space,Elem]
      ∀ x: Space; r: Elem; l: LC[Space,Elem]
      • eval(empty) = 0                              %(eval_empty_EVS)%
      • lookup(x, l) = r ⇒ eval(l) = r ∗ x + eval(l − [ r / x ])
                                                       %(eval_add_EVS)%
      • isZero(l) ⇔ ∀ y: Space • lookup(y, l) = 0 if def lookup(y, l)
                                                      %(isZero_def_EVS)%
end
```

```
spec ConstructVSWithBase [Field] [sort Base]
     given Int = %mono
     VectorSpaceLC [VectorSpace [Field]]
then
     sort  Base < Space
then
     Map [sort Base] [sort Elem]
     with sort Map[Base,Elem] ↦ LC[Base,Elem]
     hide sorts Set[Base], Set[Elem], NonEmptySet[Base],
           NonEmptySet[Elem]
then
     sort  LC[Base,Elem] < LC[Space,Elem]
     ∀ l: LC[Base,Elem]
     • ∀ y: Space • ∃ k: LC[Base,Elem] • y = eval(k)
                                                    %(generating_CVSB)%
     • eval(l) = 0 ⇒ isZero(l)                    %(independent_CVSB)%
end

spec VSWithBase [Field] [sort Base] =
     ConstructVSWithBase [Field] [sort Base]
     reveal sorts Space, Elem, Base,
               ops __+__ : Space × Space → Space, 0 : Space,
                   __*__ : Elem × Space → Space,
                   __*__ : Elem × Elem → Elem,
                   __+__ : Elem × Elem → Elem, 0 : Elem, e : Elem
end

spec ExtVectorSpace [VectorSpace [Field]]
     given Int = %mono
     RichField
and
     ExtAbelianGroup [view AbelianGroup_in_VectorSpace
                                                   [Field]]
     with ops inv ↦ −__, __^__ ↦ __times__,
              __/__ ↦ __−__
and
     RichMonoidAction [Monoid]
and
     ExtGroupAction [view GroupAction_in_VectorSpace
                                       [Field]]
and
     VectorSpaceLC [VectorSpace [Field]]
end
```

```
spec ExtVSWithBase [VSWithBase [Field] [sort Base]]
     given Int = %mono
     ExtVectorSpace [VSWithBase [Field] [sort Base]]
and
     ConstructVSWithBase [Field] [sort Base]
     with sort LC[Base,Elem]
then %implies
     ∀ l, k: LC[Base,Elem]
     • eval(l) = eval(k) ⇒ l = k                    %(uniqueRepres_EVSB)%
then %def
     op    coefficients : Space → LC[Base,Elem]
     ∀ x: Space
     • eval(coefficients(x)) = x                   %(coefficients_def_EVSB)%
then %implies
     ∀ l: LC[Base,Elem]
     • coefficients(eval(l)) = l                       %(recoverCoeff_EVSB)%
end

spec VectorTuple [VectorSpace [Field]] [op n : Pos]
     given Int = %mono
     {      Array [ops 1, n : Int fit ops min : Int ↦ 1, max : Int ↦ n]
                  [sort Space]
            hide sorts Set[Space], Set[Index], NonEmptySet[Space],
                       NonEmptySet[Index]
            with sorts Index ↦ Index[n], Array[Space]
       then %def
            sort   Index[n] < Nat
       then
            sort   Tuple[Space,n] = {x: Array[Space]
                                     • ∀ i: Index[n] • def x ! i}
            ops    __!__ : Tuple[Space,n] × Index[n] → Space;
                   0 : Tuple[Space,n];
                   __*__ : Elem × Tuple[Space,n] → Tuple[Space,n];
                   __+__ :
                      Tuple[Space,n] × Tuple[Space,n] → Tuple[Space,n];
                   auxsum : Tuple[Space,n] × Index[n] → Space;
                   sum : Tuple[Space,n] → Space
            ∀ r: Elem; x, y: Tuple[Space,n]; i: Index[n]
            • 0 ! i = 0                                     %(0_def_Tuple)%
            • (r * x) ! i = r * (x ! i)                  %(mult_def_Tuple)%
            • (x + y) ! i = (x ! i) + (y ! i)              %(add_def_Tuple)%
            • auxsum(x, 1 as Index[n]) = x ! 1 as Index[n]
                                                         %(auxsum_1_Tuple)%
            • auxsum(x, suc(i) as Index[n]) =
              auxsum(x, i) + (x ! suc(i) as Index[n])
```

```
                                                  %(auxsum_suc_Tuple)%
          • sum(x) = auxsum(x, n as Index[n])      %(sum_def_Tuple)%
      }
      hide op auxsum
end

spec ConstructVector [Field] [op n : Pos]
      given Int = %mono
      VectorTuple [view VectorSpace_in_Field [Field]]
                        [op n : Pos]
      with sorts Tuple[Elem,n] ↦ Vector[Elem,n], Index[n],
            ops 0, __*__, __+__, sum
      with op __!__ : Vector[Elem,n] × Index[n] → Elem
then
      ops   <__||__> : Vector[Elem,n] × Vector[Elem,n] → Elem;
            prod : Vector[Elem,n] → Elem;
            unitVector : Index[n] → Vector[Elem,n];
            auxmult : Vector[Elem,n] × Vector[Elem,n] → Vector[Elem,n];
            auxprod : Vector[Elem,n] × Index[n] → Elem
      pred  orthogonal : Vector[Elem,n] × Vector[Elem,n]
      ∀ x, y: Vector[Elem,n]; i, j: Index[n]
      • auxmult(x, y) ! i = (x ! i) * (y ! i)        %(auxmult_def_CVector)%
      • < x || y > = sum(auxmult(x, y))                 %(scp_def_CVector)%
      • auxprod(x, 1 as Index[n]) = x ! 1 as Index[n]
                                                    %(auxprod_1_CVector)%
      • auxprod(x, suc(i) as Index[n]) =
        auxprod(x, i) * (x ! suc(i) as Index[n])
                                                  %(auxprod_suc_CVector)%
      • prod(x) = auxprod(x, n as Index[n])          %(prod_def_CVector)%
      • orthogonal(x, y) ⇔ < x || y > = 0      %(orthogonal_def_CVector)%
      • unitVector(i) ! j = e when i = j else 0         %(unitVector_def)%
      sort  UnitVector[Elem,n] = {x: Vector[Elem,n]
                                      • ∃ i: Index[n] • x = unitVector(i)}
      hide ops auxmult, auxprod
then %implies
      ∀ x, y: Vector[Elem,n]
      • < x || y > = < y || x >                        %(scpComm_CVector)%
      • < x || x > = 0 ⇒ x = 0                          %(scpPos_CVector)%
end

spec Vector [Field] [op n : Pos] given Int =
      ConstructVector [Field] [op n : Pos]
      reveal sorts Vector[Elem,n], UnitVector[Elem,n],
              ops __+__ :
                    Vector[Elem,n] × Vector[Elem,n] → Vector[Elem,n],
                   __*__ : Elem × Vector[Elem,n] → Vector[Elem,n],
                   0 : Vector[Elem,n]
end

view VECTORSPACE_IN_VECTOR [FIELD] [op n : Pos] given NAT :
     VECTORSPACE [FIELD] to VECTOR [FIELD] [op n : Pos] =
     sort Space ↦ Vector[Elem,n]
end

spec SYMMETRICGROUP [op n : Pos] given INT = %mono
     sort  Index[n] = {i: Pos • i ≤ n}
then
     ARRAY [ops 1, n : Int fit ops min : Int ↦ 1, max : Int ↦ n]
           [sort Index[n]]
     with sorts Array[Index[n]], Index ↦ Index[n]
then
     sort  Perm[n] = {p: Array[Index[n]]
                     • ∀ i: Index[n] • ∃ j: Index[n] • p ! j = i}
     ops   id : Perm[n];
           __comp__ : Perm[n] × Perm[n] → Perm[n];
           sign : Perm[n] → Int;
           nFac: Nat = n !                        %(nFac_def_SymGroup)%
     ∀ p, q: Perm[n]; i: Index[n]
     • id ! i = i                                  %(id_def_SymGroup)%
     • (p comp q) ! i = p ! (q ! i)              %(comp_def_SymGroup)%
     • sign(p comp q) = sign(p) ∗ sign(q)
                                           %(signHomomorphic_SymGroup)%
     • abs(sign(p)) = 1                          %(signRange_SymGroup)%
     • ∃ r: Perm[n] • sign(r) = − 1               %(signSurj_SymGroup)%
then %cons
     sort  PermIndex[n] = {i: Pos • i ≤ nFac}
     op    perm : PermIndex[n] → Perm[n]
     ∀ p: Perm[n]
     • ∃ i: PermIndex[n] • perm(i) = p           %(permSurj_SymGroup)%
end

view GROUP_IN_SYMMETRICGROUP [op n : Pos] given NAT :
     GROUP to SYMMETRICGROUP [op n : Pos] =
     sort Elem ↦ Perm[n], ops __∗__ ↦ __comp__, e ↦ id
end
```

```
spec Matrix [Field] [op n : Pos] given Int = %mono
     VectorTuple [view VectorSpace_in_Vector
                                          [Field] [op n : Pos]]
                      [op n : Pos]
     with sorts Index[n], Tuple[Vector[Elem,n],n] ↦ Matrix[Elem,n]
and
     ConstructVector [Field] [op n : Pos]
and
     ExtField [Field]
then
     ops   transpose : Matrix[Elem,n] → Matrix[Elem,n];
           1 : Matrix[Elem,n];
           elementary : Index[n] × Index[n] → Matrix[Elem,n];
           __*__ : Matrix[Elem,n] × Vector[Elem,n] → Vector[Elem,n];
           __*__ : Matrix[Elem,n] × Matrix[Elem,n] → Matrix[Elem,n];
           det : Matrix[Elem,n] → Elem
     ∀ a, b: Matrix[Elem,n]; x: Vector[Elem,n]; i, j, k: Index[n]
     • (transpose(a) ! i) ! j = (a ! j) ! i          %(transpose_def_Matrix)%
     • (1 ! i) ! j = e when i = j else 0                      %(1_def_Matrix)%
     • elementary(i, j) ! k = unitVector(j) when i = k else 0
                                                    %(elementary_def_Matrix)%
     • (a * x) ! i = < transpose(a) ! i || x >        %(scalmult_def_Matrix)%
     • (a * b) ! i = a * (b ! i)                           %(mult_def_Matrix)%
     sort  ElementaryMatrix[Elem,n] = {x: Matrix[Elem,n]
                                       • ∃ i, j: Index[n]
                                         • x = elementary(i, j)}
then
     local
             SymmetricGroup [op n : Pos]
             with sorts Perm[n], PermIndex[n], ops sign, perm, nFac
       then
             closed ConstructVector [Field]
                                          [op nFac : Pos]
             with sorts Vector[Elem,nFac],
                        Index[nFac] ↦ PermIndex[n]
       then
             ops   summands : Matrix[Elem,n] → Vector[Elem,nFac];
                   factors :
                      Matrix[Elem,n] × PermIndex[n] → Vector[Elem,n]
     within
       ∀ a: Matrix[Elem,n]; i: Index[n]; j: PermIndex[n]
       • factors(a, j) ! i = (a ! i) ! (perm(j) ! i)   %(factors_def_Matrix)%
       • summands(a) ! j = prod(factors(a, j)) times sign(perm(j))
                                                      %(summands_def_Matrix)%
       • det(a) = sum(summands(a))                                %(Leibnitz)%
```

```
then %implies
     ∀ a, b: Matrix[Elem,n]
     • det(0) = 0                                                  %(det0)%
     • ¬ det(a) = 0 ⇔ ∀ x: Vector[Elem,n] • x = 0 if  a ∗ x = 0
                                                            %(detVanishes)%
     • det(1) = e                                                  %(det1)%
     • det(a ∗ b) = det(a) ∗ det(b)                             %(detMult)%
end

spec RichVectorSpace =
     ExtVectorSpace [VectorSpace [Field]]
end

spec RichVSWithBase [Field] [sort Base] =
     ExtVSWithBase [VSWithBase [Field] [sort Base]]
end

view VectorSpace_in_VectorTuple [VectorSpace [Field]]
                                             [op n : Pos]
     given Int :
     VectorSpace [Field] to
     VectorTuple [VectorSpace [Field]] [op n : Pos] =
     sort Space ↦ Tuple[Space,n]
end

view VSWithBase_in_Field [     Field
                           then
                                op a : Elem
                                • ¬ a = 0 ] :
     VSWithBase [Field] [sort Base] to
     {     Field
      then
           sort   Singleton[a] = {x: Elem • x = a}
     } =
     sorts Space ↦ Elem, Base ↦ Singleton[a],
     op __∗__ : Elem × Space → Space ↦
         __∗__ : Elem × Elem → Elem
end

view VSWithBase_in_Vector [Field] [op n : Pos] given Nat :
     VSWithBase [Field] [sort Base] to
     Vector [Field] [op n : Pos] =
     sorts Space ↦ Vector[Elem,n], Base ↦ UnitVector[Elem,n]
end
```

```
view VSWithBase_in_Matrix [Field] [op n : Pos] given Nat :
        VSWithBase [Field] [sort Base] to Matrix [Field] [op n : Pos] =
        sorts Space ↦ Matrix[Elem,n], Base ↦ ElementaryMatrix[Elem,n]
end

%% The following view expresses that every vector space has a base.
%% This holds because the CASL semantics assumes the axiom of choice.
view VSWithBase_in_VectorSpace [Field] given Int :
        {VSWithBase [Field] [sort Base] hide sort Base} to
        VectorSpace [Field]
end
```

10

Library Basic/LinearAlgebra_II

```
library Basic/LinearAlgebra_II version 1.0

%authors : L. Schröder <lschrode@tzi.de>, M. Roggenbach, T. Mossakowski
%date : 9 January 2004
%prec({__*__} < {__^__})%
%prec({__+__, __−__} < {__/__, __*__})%
%left_assoc(__+__, __*__, __^__)%


from Basic/Numbers get Nat, Int

from Basic/Algebra_I get Field, RichField, Ring, ExtRing

from Basic/Algebra_II get Polynomial

from Basic/LinearAlgebra_I get
      VectorSpace, VSWithBase, ExtVectorSpace, Matrix

spec FreeVectorSpace [Field] [sort Base] = %mono
      free {        VectorSpace [Field]
            then
                    op      inject : Base → Space
           }
end

spec Algebra [Field] =
      VectorSpace [Field]
and
      closed {Ring
                with sort Elem ↦ Space, ops __+__, __*__, 0, e}
```

```
then
     sort   Elem < Space
     ∀ r: Elem; x, y: Space
     • r * x * y = r * (x * y)                    %(leftLinear_Algebra)%
     • x * (r * y) = r * (x * y)                  %(rightLinear_Algebra)%
end

spec FreeAlgebra [Field] =
     free {      Algebra [Field]
          then
                 op     X : Space
          }
end

spec ExtFreeVectorSpace [FreeVectorSpace [Field]
                                                  [sort Base]]
     given Int =
     ExtVectorSpace [FreeVectorSpace [Field] [sort Base]]
end

spec ExtAlgebra [Algebra [Field]] given Int = %mono
     RichField
and
     ExtVectorSpace [VectorSpace [Field]]
and
     ExtRing [Algebra [Field] fit sort Elem ↦ Space]
and
     Polynomial [Field]
then
     op     eval : Poly[Elem] × Space → Space
     ∀ a: Elem; p: Poly[Elem]; x: Space
     • eval(0, x) = 0                             %(eval_0_EAlgebra)%
     • eval(a ::: p, x) = a + eval(p, x) * x      %(eval_cons_EAlgebra)%
end

spec ExtFreeAlgebra [Field] given Int =
     ExtAlgebra [FreeAlgebra [Field]]
end

view Algebra_in_Matrix [Field] [op n : Pos] given Nat :
     Algebra [Field] to Matrix [Field] [op n : Pos] =
     sort Space ↦ Matrix[Elem,n], op e ↦ 1
end

spec RichAlgebra [Field] =
      ExtAlgebra [Algebra [Field]]
end

spec RichFreeVectorSpace [Field] [sort Base] =
      ExtFreeVectorSpace [FreeVectorSpace [Field] [sort Base]]
end

spec RichFreeAlgebra [Field] =
      ExtFreeAlgebra [Field]
end

%% The following view expresses that a vector space is free over
%% any of its bases
view FreeVectorSpace_in_VSWithBase [Field] given Int :
      FreeVectorSpace [Field] [sort Base] to
      {       VSWithBase [Field] [sort Base]
       then
              op     inject : Base → Space
              ∀ x: Base
              • inject(x) = x                                  %(inject_def_VSB)%
      }
end

view FreeAlgebra_in_Polynomial [Field] given Int :
      FreeAlgebra [Field] to Polynomial [Field] =
      sort Space ↦ Poly[Elem]
end
```

11

Library Basic/MachineNumbers

library BASIC/MACHINENUMBERS **version** 1.0

%authors : T. Mossakowski <till@tzi.de>, M. Roggenbach, L. Schröder
%date : 25 June 2003
%{ This library contains specifications of those subtypes
of the naturals and the integers that are used on actual
machines.
The specifications CARDINAL and INTEGER provide subtypes
of Nat and Int consisting of those numbers that have
a binary representation within a given word length.
Operations on these data types are partial restrictions
of the usual operations on Nat and Int - they are undefined
if the word length is exceeded.
The specification TwoComplement provides a “cyclic”
version of bounded integers that corresponds to the
common two complement representation of integers
used in many programming languages.
Operations are total here - the successor of the
maximal positive number fitting in the word length is
the minimal negative number.
The Ext versions of the specifications add min and max
operations (inherited from TotalOrder). **}%**

from BASIC/RELATIONSANDORDERS **get**
TOTALORDER, EXTTOTALORDER

from BASIC/NUMBERS **get** NAT, INT

spec CARDINAL [**op** *WordLength* : *Nat*] **given** NAT = **%mono**
NAT

```
then %mono
%% Define CARDINAL to be isomorphic to the subset
%% 0.. 2ˆWordLength-1 of Nat
%% using a partial constructor natToCard
      type  CARDINAL ::= natToCard(cardToNat:Nat)?
      ∀ x: Nat; c: CARDINAL
      • def natToCard(x) ⇔ x ≤ (2 ˆ WordLength −? 1)
                                                   %(natToCard_dom)%
      • natToCard(cardToNat(c)) = c                %(natToCard_def)%
then %def
%% The predicates and operations are just inherited from Nat,
%% but operations may become partial, since natToCard is partial
      pred  __≤__ : CARDINAL × CARDINAL
      ∀ x, y: CARDINAL
      • x ≤ y ⇔ cardToNat(x) ≤ cardToNat(y)        %(leq_CARDINAL)%
then %def
      ops   maxCardinal : Nat;
            0, 1, maxCardinal : CARDINAL;
            __+__, __−__, __∗__, __div__, __mod__ :
                     CARDINAL × CARDINAL →? CARDINAL
      • maxCardinal = 2 ˆ WordLength −? 1          %(maxCardinal_Nat)%
      • maxCardinal = natToCard(maxCardinal)
                                                   %(maxCardinal_CARDINAL)%
      ∀ x, y: CARDINAL
      • natToCard(0) = 0                           %(def_0_CARDINAL)%
      • natToCard(1) = 1                           %(def_1_CARDINAL)%
      • x + y = natToCard(cardToNat(x) + cardToNat(y))
                                                   %(add_CARDINAL)%
      • x − y = natToCard(cardToNat(x) −? cardToNat(y))
                                                   %(sub_CARDINAL)%
      • x ∗ y = natToCard(cardToNat(x) ∗ cardToNat(y))
                                                   %(mult_CARDINAL)%
      • x div y = natToCard(cardToNat(x) div cardToNat(y))
                                                   %(div_CARDINAL)%
      • x mod y = natToCard(cardToNat(x) mod cardToNat(y))
                                                   %(mod_CARDINAL)%
then %implies
      ops   __+__ : CARDINAL × CARDINAL →? CARDINAL,
               assoc, comm, unit 0;
            __∗__ : CARDINAL × CARDINAL →? CARDINAL,
               assoc, comm, unit 1
      ∀ x, y: CARDINAL
      • def x + y ⇔ (cardToNat(x) + cardToNat(y)) ≤ maxCardinal
                                                   %(add_CARDINAL_dom)%
      • def x − y ⇔ y ≤ x                          %(sub_CARDINAL_dom)%
```

• $def\ x * y \Leftrightarrow (cardToNat(x) * cardToNat(y)) \leq maxCardinal$
%(mult_CARDINAL_dom)%
• $def\ x\ div\ y \Leftrightarrow \neg\ y = 0$ %(div_CARDINAL_dom)%
• $def\ x\ mod\ y \Leftrightarrow \neg\ y = 0$ %(mod_CARDINAL_dom)%
end

spec INTEGER [**op** *WordLength* : *Nat*] **given** NAT = **%mono**
INT
then %mono
%% Define INTEGER to be isomorphic to the subset
%% -2^(WordLength-1)..2^(WordLength-1)-1 of Int
%% using a partial constructor intToInteger
type $INTEGER ::= intToInteger(integerToInt{:}Int)?$
$\forall\ x{:}\ Int;\ i{:}\ INTEGER$
• $def\ intToInteger(x) \Leftrightarrow$
$-\ (2\ \hat{}\ (WordLength\ -?\ 1)) \leq x$
$\wedge\ x \leq (2\ \hat{}\ (WordLength\ -?\ 1) - 1)$
%(intToInteger_dom)%
• $intToInteger(integerToInt(i)) = i$ %(intToInteger_def)%
then %def
%% The predicates and operations are just inherited from Int,
%% but operations may become partial, since intToInteger is partial
pred $\_\_\leq\_\_ : INTEGER \times INTEGER$
$\forall\ x,\ y{:}\ INTEGER$
• $x \leq y \Leftrightarrow integerToInt(x) \leq integerToInt(y)$ %(leq_INTEGER)%
then %def
ops $maxInteger,\ minInteger : Int;$
$0,\ 1,\ maxInteger,\ minInteger : INTEGER;$
$-\_\_,\ abs : INTEGER \rightarrow?\ INTEGER;$
$\_\_+\_\_,\ \_\_-\_\_,\ \_\_*\_\_,\ \_\_/\_\_,\ \_\_div\_\_,\ \_\_mod\_\_,$
$\_\_quot\_\_,\ \_\_rem\_\_ :$
$INTEGER \times INTEGER \rightarrow?\ INTEGER$
• $maxInteger = 2\ \hat{}\ (WordLength\ -?\ 1) - 1$ %(maxInteger_Int)%
• $minInteger = -\ (2\ \hat{}\ (WordLength\ -?\ 1))$ %(minInteger_Int)%
• $maxInteger = intToInteger(maxInteger)$
%(maxInteger_INTEGER)%
• $minInteger = intToInteger(minInteger)$
%(minInteger_INTEGER)%
$\forall\ x,\ y{:}\ INTEGER$
• $intToInteger(0) = 0$ %(def_0_INTEGER)%
• $intToInteger(1) = 1$ %(def_1_INTEGER)%
• $-\ x = intToInteger(-\ integerToInt(x))$ %(minus_INTEGER)%
• $abs(x) = intToInteger(abs(integerToInt(x)))$ %(abs_INTEGER)%
• $x + y = intToInteger(integerToInt(x) + integerToInt(y))$
%(add_INTEGER)%

• $x - y = \mathit{intToInteger}(\mathit{integerToInt}(x) - \mathit{integerToInt}(y))$
%(sub_INTEGER)%

• $x * y = \mathit{intToInteger}(\mathit{integerToInt}(x) * \mathit{integerToInt}(y))$
%(mult_INTEGER)%

• $x \mathbin{/} y = \mathit{intToInteger}(\mathit{integerToInt}(x) \mathbin{/?} \mathit{integerToInt}(y))$
%(divide_INTEGER)%

• $x \mathbin{\mathit{div}} y = \mathit{intToInteger}(\mathit{integerToInt}(x) \mathbin{\mathit{div}} \mathit{integerToInt}(y))$
%(div_INTEGER)%

• $x \mathbin{\mathit{mod}} y = \mathit{intToInteger}(\mathit{integerToInt}(x) \mathbin{\mathit{mod}} \mathit{integerToInt}(y))$
%(mod_INTEGER)%

• $x \mathbin{\mathit{quot}} y = \mathit{intToInteger}(\mathit{integerToInt}(x) \mathbin{\mathit{quot}} \mathit{integerToInt}(y))$
%(quot_INTEGER)%

• $x \mathbin{\mathit{rem}} y = \mathit{intToInteger}(\mathit{integerToInt}(x) \mathbin{\mathit{rem}} \mathit{integerToInt}(y))$
%(rem_INTEGER)%

then %implies

ops __+__ : $\mathit{INTEGER} \times \mathit{INTEGER} \to ?\ \mathit{INTEGER}$,
assoc, *comm*, *unit 0*;
__*__ : $\mathit{INTEGER} \times \mathit{INTEGER} \to ?\ \mathit{INTEGER}$,
assoc, *comm*, *unit 1*

$\forall\ x, y$: *INTEGER*

• $\mathit{def}\ - x \Leftrightarrow (\mathit{minInteger} + 1) \leq \mathit{integerToInt}(x)$
%(minus_INTEGER_dom)%

• $\mathit{def}\ \mathit{abs}(x) \Leftrightarrow (\mathit{minInteger} + 1) \leq \mathit{integerToInt}(x)$
%(abs_INTEGER_dom)%

• $\mathit{def}\ x + y \Leftrightarrow$
$\mathit{minInteger} \leq (\mathit{integerToInt}(x) + \mathit{integerToInt}(y)) \wedge$
$(\mathit{integerToInt}(x) + \mathit{integerToInt}(y)) \leq \mathit{maxInteger}$
%(add_INTEGER_dom)%

• $\mathit{def}\ x - y \Leftrightarrow$
$\mathit{minInteger} \leq (\mathit{integerToInt}(x) - \mathit{integerToInt}(y)) \wedge$
$(\mathit{integerToInt}(x) - \mathit{integerToInt}(y)) \leq \mathit{maxInteger}$
%(sub_INTEGER_dom)%

• $\mathit{def}\ x * y \Leftrightarrow$
$\mathit{minInteger} \leq (\mathit{integerToInt}(x) * \mathit{integerToInt}(y)) \wedge$
$(\mathit{integerToInt}(x) * \mathit{integerToInt}(y)) \leq \mathit{maxInteger}$
%(mult_INTEGER_dom)%

• $\mathit{def}\ x \mathbin{/} y \Leftrightarrow \mathit{def}\ \mathit{intToInteger}(\mathit{integerToInt}(x) \mathbin{/?} \mathit{integerToInt}(y))$
%(divide_INTEGER_dom)%

• $\mathit{def}\ x \mathbin{\mathit{div}} y \Leftrightarrow \neg\ y = 0$ %(div_INTEGER_dom)%

• $\mathit{def}\ x \mathbin{\mathit{mod}} y \Leftrightarrow \neg\ y = 0$ %(mod_INTEGER_dom)%

• $\mathit{def}\ x \mathbin{\mathit{quot}} y \Leftrightarrow \neg\ y = 0$ %(quot_INTEGER_dom)%

• $\mathit{def}\ x \mathbin{\mathit{rem}} y \Leftrightarrow \neg\ y = 0$ %(rem_INTEGER_dom)%

end

spec TwoComplement [**op** *WordLength* : *Nat*]

```
        given NAT = %mono
        INT
then %mono
%% Define TwoComplement to be isomorphic to the subset
%% -2^(WordLength-1)..2^(WordLength-1)-1 of Int
%% using a total constructor intToTC
%% The constructor can be total because integers are
%% taken modulo 2^WordLength
        generated type TwoComplement ::= intToTC(Int)
        ops    maxInteger, minInteger : Int;
               TCtoInt : TwoComplement → Int
        • maxInteger = 2 ^ (WordLength −? 1) − 1        %(maxInteger_Int)%
        • minInteger = − (2 ^ (WordLength −? 1))        %(minInteger_Int)%
        ∀ x, y: Int; i: TwoComplement
        • intToTC(x) = intToTC(x + 2 ^ WordLength)        %(cycle_max)%
        • intToTC(x) = intToTC(y) ⇒ x − y mod 2 ^ WordLength = 0
                                                          %(cycle_min)%
        • TCtoInt(intToTC(x)) = x if minInteger ≤ x ∧ x ≤ maxInteger
then %def
%% The predicates and operations are just inherited from Int.
%% Operations remain total, since intToTC is total
        pred   __≤__ : TwoComplement × TwoComplement
        ∀ x, y: TwoComplement
        • x ≤ y ⇔ TCtoInt(x) ≤ TCtoInt(y)          %(leq_TwoComplement)%
then %def
        ops    0, 1, maxInteger, minInteger : TwoComplement;
               −__, abs : TwoComplement → TwoComplement;
               __+__, __−__, __∗__, __/__, __div__, __mod__,
               __quot__, __rem__ :
                  TwoComplement × TwoComplement → TwoComplement
        • maxInteger = intToTC(maxInteger)
                                          %(maxInteger_TwoComplement)%
        • minInteger = intToTC(minInteger)
                                          %(minInteger_TwoComplement)%
        ∀ x, y: TwoComplement
        • intToTC(0) = 0                          %(def_0_TwoComplement)%
        • intToTC(1) = 1                          %(def_1_TwoComplement)%
        • − x = intToTC(− TCtoInt(x))             %(minus_TwoComplement)%
        • abs(x) = intToTC(abs(TCtoInt(x)))         %(abs_TwoComplement)%
        • x + y = intToTC(TCtoInt(x) + TCtoInt(y))
                                                    %(add_TwoComplement)%
        • x − y = intToTC(TCtoInt(x) − TCtoInt(y))
                                                    %(sub_TwoComplement)%
        • x ∗ y = intToTC(TCtoInt(x) ∗ TCtoInt(y))
                                                   %(mult_TwoComplement)%
```

```
     • x / y = intToTC(TCtoInt(x) /? TCtoInt(y))
                                           %(divide_TwoComplement)%
     • x div y = intToTC(TCtoInt(x) div TCtoInt(y))
                                              %(div_TwoComplement)%
     • x mod y = intToTC(TCtoInt(x) mod TCtoInt(y))
                                              %(mod_TwoComplement)%
     • x quot y = intToTC(TCtoInt(x) quot TCtoInt(y))
                                             %(quot_TwoComplement)%
     • x rem y = intToTC(TCtoInt(x) rem TCtoInt(y))
                                              %(rem_TwoComplement)%
end

view TotalOrder_in_CARDINAL [op WordLength : Nat]
     given Nat :
     TotalOrder to CARDINAL [op WordLength : Nat] =
     sort Elem ↦ CARDINAL
end

view TotalOrder_in_INTEGER [op WordLength : Nat]
     given Nat :
     TotalOrder to INTEGER [op WordLength : Nat] =
     sort Elem ↦ INTEGER
end

view TotalOrder_in_TwoComplement [op WordLength : Nat]
     given Nat :
     TotalOrder to TwoComplement [op WordLength : Nat] =
     sort Elem ↦ TwoComplement
end

spec ExtCARDINAL [op WordLength : Nat] given Nat =
     ExtTotalOrder [view TotalOrder_in_CARDINAL
                                   [op WordLength : Nat]]
end

spec ExtINTEGER [op WordLength : Nat] given Nat =
     ExtTotalOrder [view TotalOrder_in_INTEGER
                                   [op WordLength : Nat]]
end

spec ExtTwoComplement [op WordLength : Nat] given Nat =
     ExtTotalOrder [view TotalOrder_in_TwoComplement
                                   [op WordLength : Nat]]
end
```

12

Dependency Graphs of the Libraries

This chapter contains the dependency graphs for the Basic Libraries. Elliptic nodes in the graphs usually denote named specifications from the library (some of them, labeled with N1, N2, etc., also denote anonymous specifications, e.g. occurring as targets of views). Square nodes denote specifications that are imported from other libraries. Normal solid edges denote references to other specifications, whereas dotted edges denote references occurring in a formal parameter or import. Thick solid edges denote views.

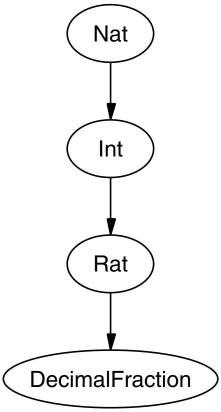

Fig. 12.1. Dependency graph for Basic/Numbers

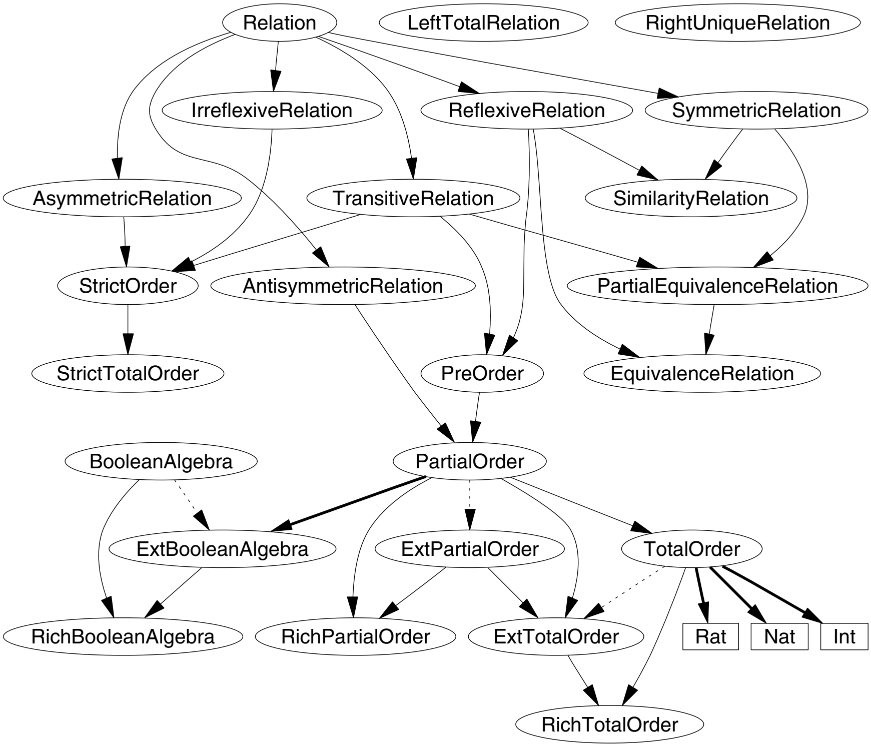

Fig. 12.2. Dependency graph for Basic/RelationsAndOrders

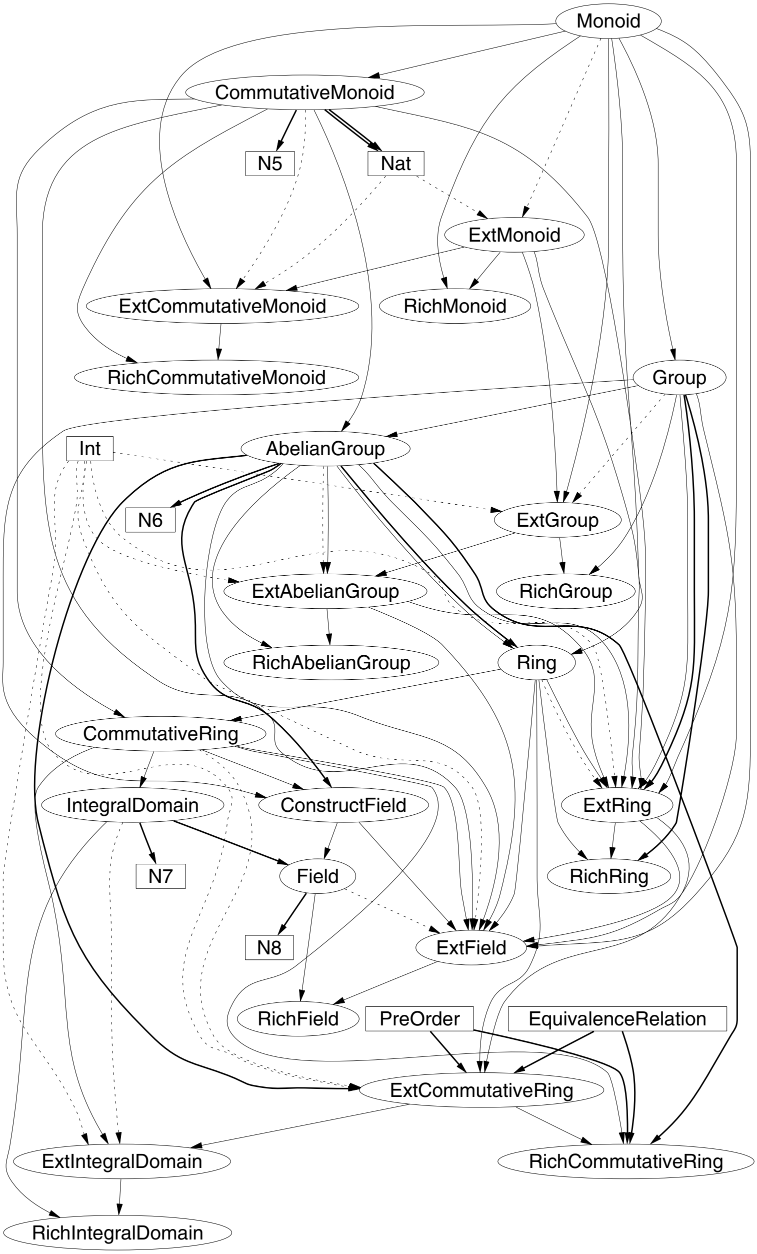

Fig. 12.3. Dependency graph for Basic/Algebra_I

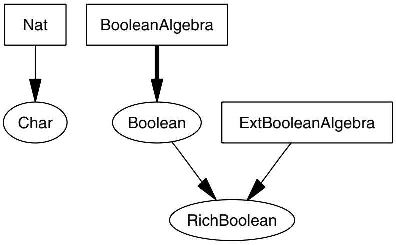

Fig. 12.4. Dependency graph for Basic/SimpleDatatypes

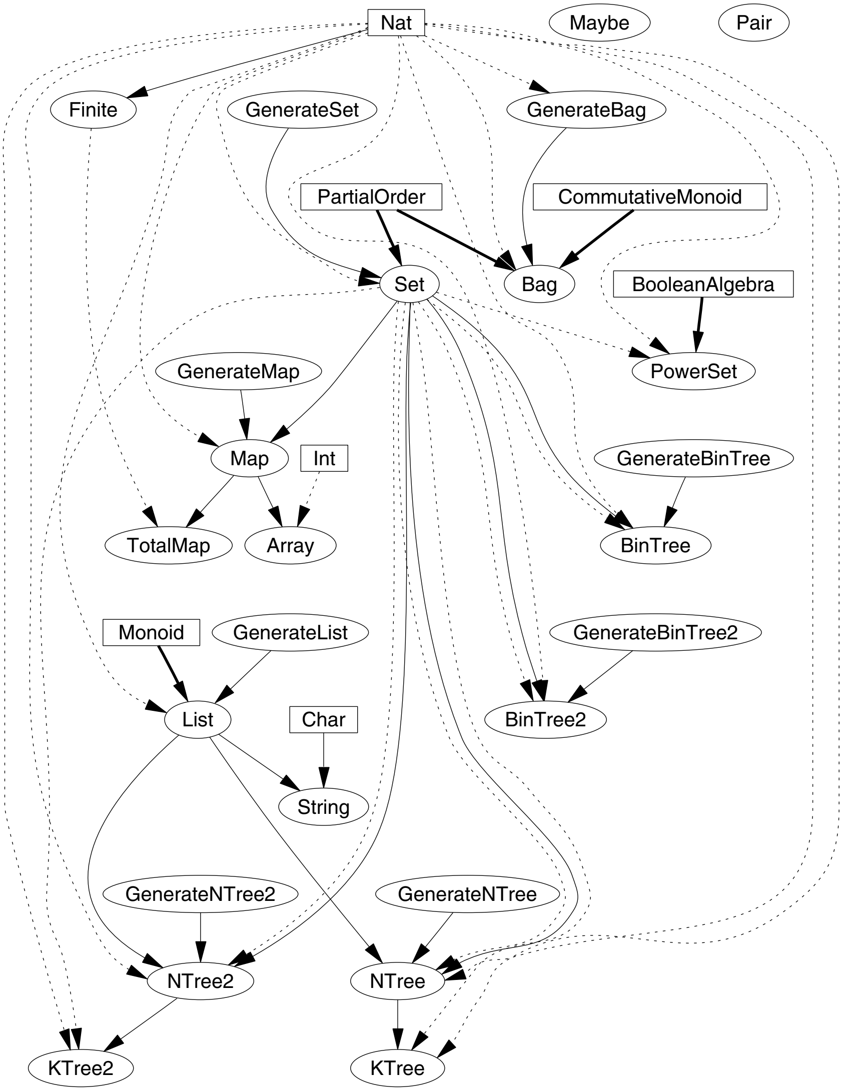

Fig. 12.5. Dependency graph for Basic/StructuredDatatypes

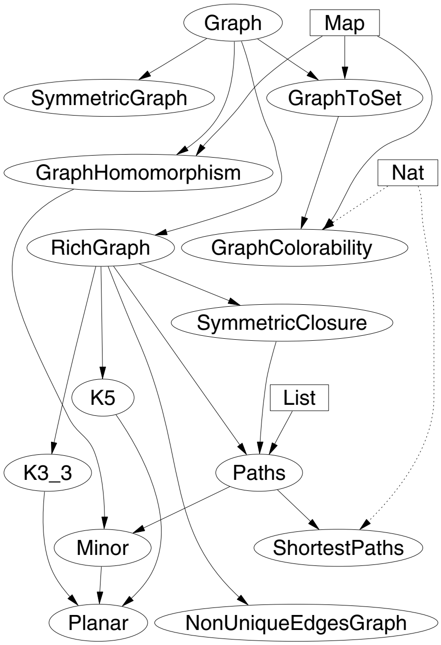

Fig. 12.6. Dependency graph for Basic/Graphs

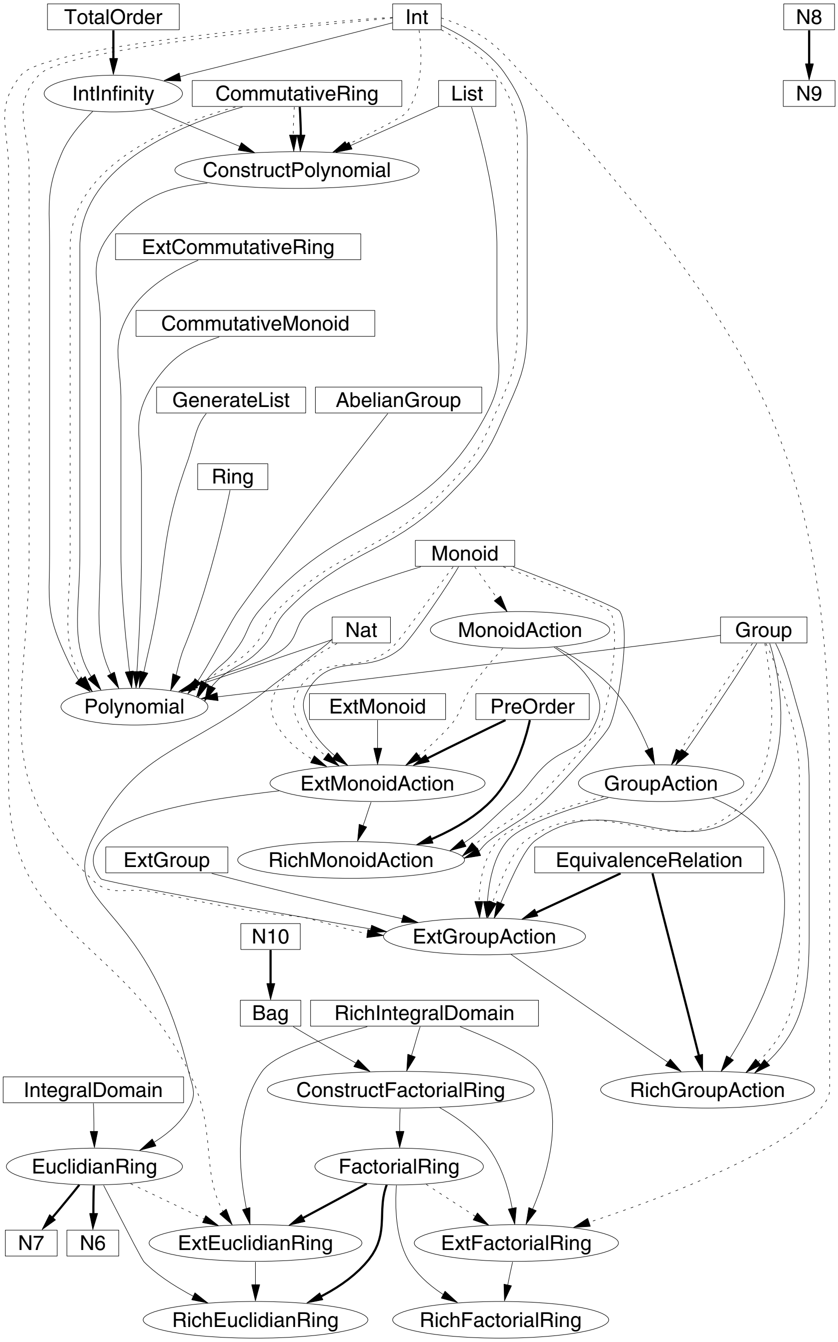

Fig. 12.7. Dependency graph for Basic/Algebra_II

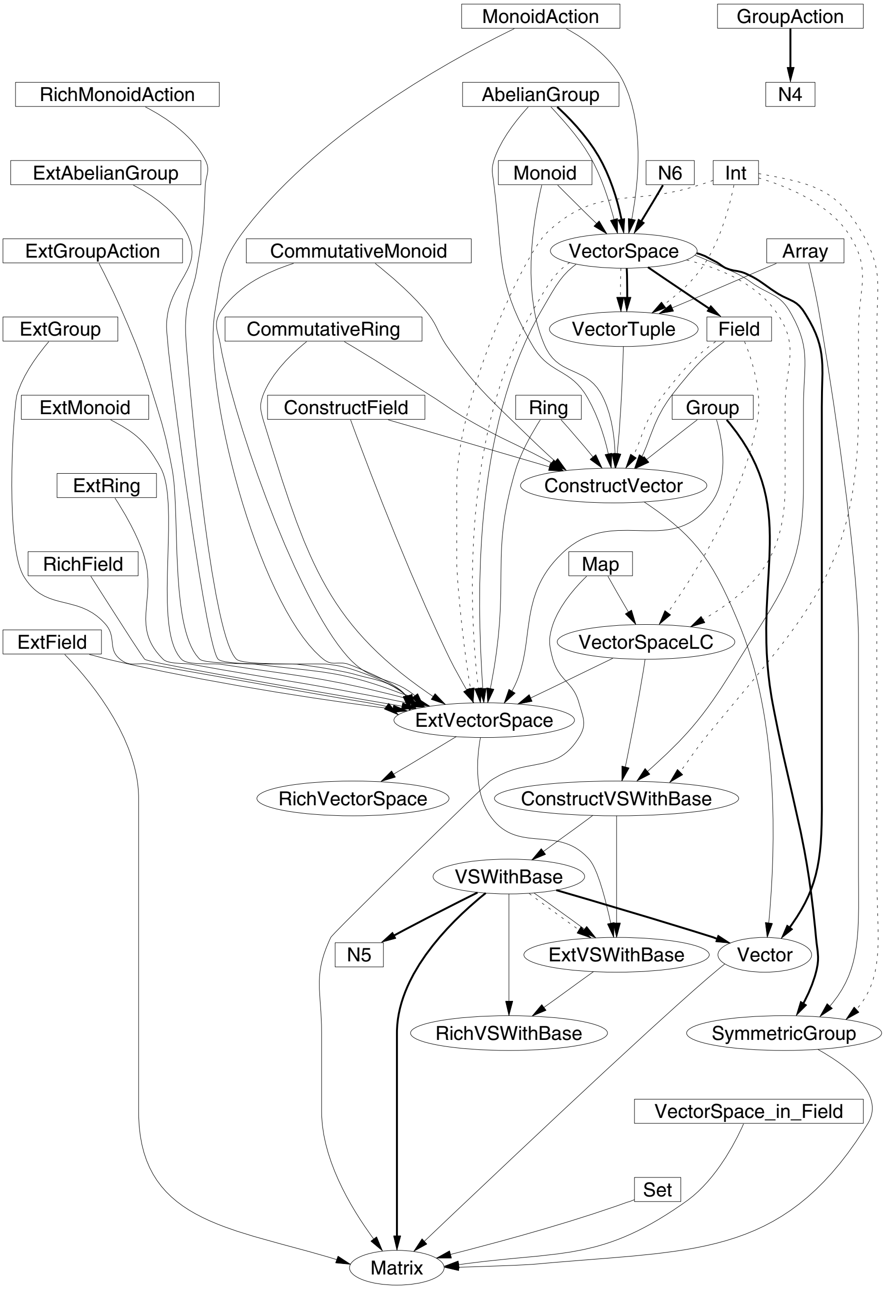

Fig. 12.8. Dependency graph for Basic/LinearAlgebra_I

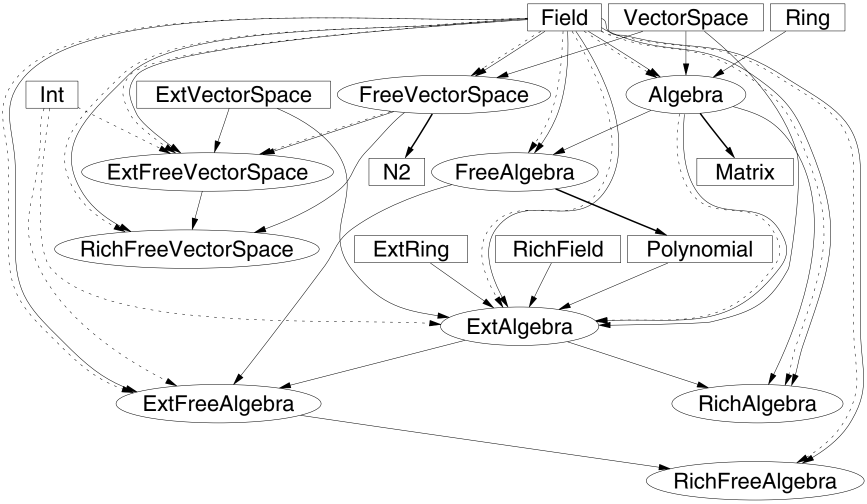

Fig. 12.9. Dependency graph for Basic/LinearAlgebra_II

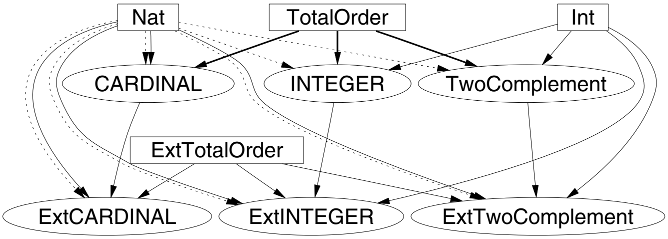

Fig. 12.10. Dependency graph for Basic/MachineNumbers

Appendices

P.D. Mosses (Ed.): CASL Reference Manual, LNCS 2960, pp. 467–528, 2004.

Annotated Bibliography

`Ancona:2000:ECL`.

Davide Ancona, Maura Cerioli, and Elena Zucca. Extending CASL by late binding. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 53–72. Springer, 2000.

> *Proposes an extension of CASL with methods, which are special functions s.t. overloading resolution for them is delayed to evaluation time and is not required to be conservative.*

`Aspinall:2002:FSC`.

David Aspinall and Donald Sannella. From specifications to code in CASL. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, Proceedings*, LNCS Vol. 2422, pages 1–14. Springer, 2002.

> *Discusses the relationship between CASL and programming languages.*

`Astesiano:1998:UHM`.

Egidio Astesiano and Gianna Reggio. UML as heterogeneous multiview notation: Strategies for a formal foundation. In L. Andrade, A. Moreira, A. Deshpande, and S. Kent, editors, *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?* ACM Press, 1998.

> *The paper presents some initial ideas about the formalization of the UML.*

`Astesiano:1999:ASC`.

Egidio Astesiano, Manfred Broy, and Gianna Reggio. Algebraic specification of concurrent systems. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, chapter 13. Springer, 1999.

> *Presents a survey of the algebraic methods for the specification of concurrent systems, using a common simple example, and classifying them in four kinds.*

`Astesiano:2000:PDC`.

Egidio Astesiano, Maura Cerioli, and Gianna Reggio. Plugging data constructs into paradigm-specific languages: Towards an application to UML. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, Proceedings*, LNCS Vol. 1816, pages 273–292. Springer, 2000.

Presents an approach for the composition of languages, in particular a data description language and a paradigm-specific language, exemplified by sketching how to combine UML and a data language.

`Astesiano:2001:LTL`.

Egidio Astesiano and Gianna Reggio. Labelled Transition Logic: An outline. *Acta Informatica*, 37(11–12), 2001.

Outlines a logical (algebraic) method for the specification of reactive/distributed systems both at the requirement and at the design level, providing references for detailed presentations of single aspects and applications.

`Astesiano:2002:CASL`.

Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. Casl: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.

Gives an overview of the Casl design, indicating major issues, and explaining main concepts and constructs. Compares Casl to some other major algebraic specification languages.

`Autexier:2000:TEF`.

Serge Autexier, Dieter Hutter, Heiko Mantel, and Axel Schairer. Towards an evolutionary formal software-development using Casl. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 73–88. Springer, 2000.

Defines a translation of a subset of CASL into the notion of development graphs, in order to maintain evolving Casl specifications.

`Autexier:2002:DGM`.

Serge Autexier, Dieter Hutter, Till Mossakowski, and Axel Schairer. The development graph manager Maya (system description). In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, Proceedings*, LNCS Vol. 2422, pages 495–502. Springer, 2002.

Explains the Maya system, which maintains structured specifications and their proofs with the help of development graphs.

`Autexier:2002:IHD`.

Serge Autexier and Till Mossakowski. Integrating Hol-Casl into the development graph manager Maya. In A. Armando, editor, *Frontiers of Combining Systems, 4th International Workshop, FroCoS 2002, Santa Margherita Ligure, Italy, Proceedings*, LNCS Vol. 2309, pages 2–17. Springer, 2002.

Maya provides management of proofs for structured specifications; Hol-Casl is a prover for Casl basic specifications. Here, these two are combined

`Baumeister:2000:ASC`.

Hubert Baumeister and Didier Bert. Algebraic specification in CASL. In M. Frappier and H. Habrias, editors, *Software Specification Methods: An Overview Using a Case Study*, FACIT (Formal Approaches to Computing and Information Technology), pages 209–224. Springer, 2000.

Explains the basic features of CASL specifications using the warehouse case study.

`Baumeister:2000:RAD`.

Hubert Baumeister. Relating abstract datatypes and Z-schemata. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 366–382. Springer, 2000.

Defines an institution for the logic underlying Z. Shows a translation of Z-schemata to abstract datatypes over that institution.

`Baumeister:2000:SBE`.

Hubert Baumeister and Alexandre V. Zamulin. State-based extension of CASL. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Integrated Formal Methods, Second International Conference, IFM 2000, Dagstuhl Castle, Germany, Proceedings*, LNCS Vol. 1945, pages 3–24. Springer, 2000.

Presents an extension of CASL for writing model-oriented specifications. The extension is based on the state-as-algebra approach.

`Baumeister:2004:CASL-Semantics`.

Hubert Baumeister, Maura Cerioli, Anne Haxthausen, Till Mossakowski, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. CASL semantics. In *CASL Reference Manual*, LNCS Vol. 2960 (IFIP Series), part III. Springer, 2004. Edited by D. Sannella and A. Tarlecki.

Presents the complete semantics of CASL in natural semantics style.

`Bert:2000:ASO`.

Didier Bert and S. Lo Presti. Algebraic specification of operator-based multimedia scenarios. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 383–400. Springer, 2000.

Presents a set of algebraic operators in CASL to create complex scenarios. Provides a semantics in a temporal model and shows how to derive some properties of the scenarios.

`Bidoit:1998:ASC`.

Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. In A. M. Haeberer, editor, *Algebraic Methodology and Software Technology, 7th International Conference, AMAST'98, Amazonia, Brazil, January 1999, Proceedings*, LNCS Vol. 1548, pages 341–357. Springer, 1998. An extended and improved version is [`Bidoit:2002:ASC`].

Motivates and presents CASL architectural specifications.

`Bidoit:2002:ASC`.

Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273, 2002.

Gives an informal motivation for and presentation of CASL architectural specifications, with hints on their semantics and use in the development process.

`Bidoit:2002:GDL`.

Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Global development via local observational construction steps. In K. Diks and W. Rytter, editors, *Mathematical Foundations of Computer Science 2002, 27th International Symposium, MFCS 2002, Warsaw, Poland, Proceedings*, LNCS Vol. 2420, pages 1–24. Springer, 2002.

> *Studies development steps that apply local constructions in a global context, and gives the semantics of a version of Casl architectural specifications, including their observational interpretation.*

`Bidoit:2004:CASL-UM`.

Michel Bidoit and Peter D. Mosses. *Casl User Manual*. LNCS Vol. 2900 (IFIP Series). Springer, 2004. With chapters by Till Mossakowski, Donald Sannella, and Andrzej Tarlecki.

> *Illustrates and discusses how to write Casl specifications, with additional chapters on foundations, tools, and libraries, a realistic case study, and a quick-reference overview of Casl.*

`Bidoit:2004:CFS`.

Michel Bidoit, Donald Sannella, and Andrzej Tarlecki. Toward component-oriented formal software development: An algebraic approach. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future, Proc. 9th Monterey Software Engineering Workshop, Venice, Italy, Sep. 2002*, LNCS Vol. 2941. Springer, 2004.

> *Provides a light-weight introduction to [`Bidoit:2002:GDL`], with an illustrative example.*

`Borzyszkowski:2000:GIC`.

Tomasz Borzyszkowski. Generalized interpolation in Casl. *Information Processing Letters*, 76:19–24, 2000.

> *Gives a proof of the Craig Interpolation Property for the partial many-sorted first-order logic, underlying Casl. This property is crucial for results presented in [`Borzyszkowski:2002:LSS`].*

`Borzyszkowski:2000:HOL`.

Tomasz Borzyszkowski. Higher-order logic and theorem proving for structured specifications. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 401–418. Springer, 2000.

> *Formulates conditions under which we can reuse the HOL logic to reason about structured specifications built over institutions mapped into HOL. It works also for the* structured part *of Casl.*

`Borzyszkowski:2002:LSS`.

Tomasz Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science*, 286:197–245, 2002.

> *Presents completeness results for proof systems for structured specifications. Also introduces a methodology for reusing complete proof systems for systems that are not complete.*

`Brand:2000:DPT`.

Mark G. J. van den Brand and Jeroen Scheerder. Development of parsing tools for Casl using generic language technology. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 89–105. Springer, 2000.

> *Describes the architecture of a Casl parser based on the SGLR parsing technology developed for Asf+Sdf, and discusses the mapping to abstract syntax trees represented as ATerms.*

`Brand:2000:EAT`.

Mark G. J. van den Brand, Hayco A. de Jong, Paul Klint, and Pieter A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000.

> *Describes an efficient and generic representation of tree-like data structures, and reports on several case studies, including the abstract syntax of Casl.*

`COMPASS:1997`.

Maura Cerioli, Martin Gogolla, Hélène Kirchner, Bernd Krieg-Brückner, Zhenyu Qian, and Markus Wolf, editors. *Algebraic System Specification and Development: Survey and Annotated Bibliography*. BISS Monographs. Shaker, 2nd edition, 1998.

> *Provides an overview of the state of the art in algebraic specification at the end of the 90's, with a comprehensive bibliography. Discusses semantics, structuring constructs, specific algebraic paradigms, methodology issues, and existing tools.*

`Cerioli:1997:PSP`.

Maura Cerioli, Anne Haxthausen, Bernd Krieg-Brückner, and Till Mossakowski. Permissive subsorted partial logic in Casl. In M. Johnson, editor, *Algebraic Methodology and Software Technology, 6th International Conference, AMAST'97, Sydney, Australia, Proceedings*, LNCS Vol. 1349, pages 91–107. Springer, 1997.

> *Presents the permissive subsorted partial logic used in the Casl semantics.*

`Cerioli:1999:TEP`.

Maura Cerioli, Till Mossakowski, and Horst Reichel. From total equational to partial first-order logic. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, chapter 3. Springer, 1999.

> *Presents partial first-order logic, both model theory and logical deduction. Compares partial specifications to error algebras and order-sortedness.*

`Choppy:1999:UCS`.

Christine Choppy and Gianna Reggio. Using Casl to specify the requirements and the design: A problem specific approach – complete version. Technical Report DISI-TR-99-33, Univ. of Genova, 1999. This is an extended version of [`Choppy:2000:UCS`], including complete case studies.

> *Shows how formal specification skeletons may be associated with the structuring concepts provided by M. Jackson's Problem Frames, used to provide a first gross structure and characterization of the system under study.*

`Choppy:2000:UCS`.

Christine Choppy and Gianna Reggio. Using CASL to specify the requirements and the design: A problem specific approach. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 104–123. Springer, 2000. An extended version is provided in [`Choppy:1999:UCS`].

> *Shows how formal specification skeletons may be associated with the structuring concepts provided by M. Jackson's Problem Frames, used to provide a first gross structure and characterization of the system under study.*

`Choppy:2003:TFG`.

Christine Choppy and Gianna Reggio. Towards a formally grounded software development method. Technical Report DISI-TR-03-35, Univ. of Genova, August 2003.

> *Presents guidelines for writing (and meanwhile understanding) descriptions/specifications, both in property-oriented and model-oriented styles. Provides visual descriptions, and formal specifications in CASL-LTL.*

`Choppy:2003:IUC`.

Christine Choppy and Gianna Reggio. Improving use case based requirements using formally grounded specifications (complete version). Technical Report DISI-TR-03-45, Univ. of Genova, October 2003. A short version is to appear in Proc. FASE 2004.

> *Presents a technique for improving use case based requirements, using the formally grounded development of the requirements specification (in CASL and CASL-LTL).*

`CoFI:2004:CASL-RM`.

CoFI (The Common Framework Initiative). *CASL Reference Manual.* LNCS Vol. 2960 (IFIP Series). Springer, 2004.

> *Gives full details of the design of CASL: an informal language summary, concrete and abstract syntax, well-formedness and model-class semantics, and proof rules. Includes the libraries of basic datatypes.*

`CoFI:2004:CASL-Summary`.

CoFI Language Design Group. CASL summary. In *CASL Reference Manual*, LNCS Vol. 2960 (IFIP Series), part I. Springer, 2004. Edited by B. Krieg-Brückner and P. D. Mosses.

> *Gives an informal summary of the CASL constructs for basic, structured, architectural, and library specifications. Defines sublanguages and lists proposed extensions of CASL.*

`CoFI:2004:CASL-Syntax`.

CoFI Language Design Group. CASL syntax. In *CASL Reference Manual*, LNCS Vol. 2960 (IFIP Series), part II. Springer, 2004. Edited by B. Krieg-Brückner and P. D.Mosses.

> *Defines the lexical, concrete, and abstract syntax of CASL.*

`Coscia:1999:JJT`.

Eva Coscia and Gianna Reggio. JTN: A Java-targeted graphic formal notation for reactive and concurrent systems. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering, Second International Conference, FASE'99, Amsterdam, The Netherlands, Proceedings*, LNCS Vol. 1577, pages 77–97. Springer, 1999.

JTN is a formal graphic notation for Java-targeted design specifications (i.e., of systems that will be implemented using Java).

`Costa:1997:SAD`.

Gerardo Costa and Gianna Reggio. Specification of abstract dynamic datatypes: A temporal logic approach. *Theoretical Computer Science*, 173(2):513–554, 1997.

Proposes a logic which combines many-sorted first-order logic with branching-time combinators for the specification of dynamic-data types.

`Haveraaen:1999:FSE`.

Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modeling. *Nordic Journal of Computing*, 6(3):241–270, 1999.

Descripes the development of a software family for seismic simulations. Algebraic methods are used for domain and software architecture engineering. Quantitative estimates of the benefits are made.

`Haveraaen:2000:2TS`.

Magne Haveraaen. A 2-tiered software process model for utilizing CASL. Technical Report 208, Dept. of Informatics, Univ. of Bergen, October 2000.

Describes a software process model where CASL is used for domain engineering.

`Haveraaen:2000:CSA`.

Magne Haveraaen. Case study on algebraic software methodologies for scientific computing. *Scientific Programming*, 8(4):261–273, 2000.

Presents the notion of algebraic software methodologies and their use for domain engineering and software architecture design.

`Hoffman:2000:SAS`.

Piotr Hoffman. Semantics of architectural specifications. Master's thesis, Warsaw Univ., 2000. In Polish.

Defines and discusses static and model semantics of architectural specifications, as well as a semantics for programs and a verification semantics, which makes use of so-called sharing maps.

`Hoffman:2001:VAS`.

Piotr Hoffman. Verifying architectural specifications. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting, Genova, Italy, 2001, Selected Papers*, LNCS Vol. 2267, pages 152–175. Springer, 2001.

Develops techniques for verifying architectural specifications w.r.t. a non-generative semantics for institutions with logical amalgamation, obtaining full verification for first-order logic.

`Hoffman:2003:VGC`.

Piotr Hoffman. Verifying generative Casl architectural specifications. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 233–252. Springer, 2003.

Presents an institution-independent proof-calculus for architectural specifications, complete w.r.t. a generative semantics, and applies it to the full Casl institution.

`Hoffmann:2003:AHO`.

Kathrin Hoffmann and Till Mossakowski. Algebraic higher order nets: Graphs and Petri nets as tokens. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 253–267. Springer, 2003.

Case study in HasCasl. Graphs and Petri nets become first-class citizens and can be used as tokens in Petri nets.

`Hussmann:1999:ADT`.

Heinrich Hussmann, Maura Cerioli, Gianna Reggio, and Françoise Tort. Abstract data types and UML models. Technical Report DISI-TR-99-15, Univ. of Genova, 1999.

Examines the relationship between object-oriented models (using UML) and the classical algebraic approach to data abstraction (using Casl).

`Hussmann:2000:UC`.

Heinrich Hussmann, Maura Cerioli, and Hubert Baumeister. From UML to Casl (static part). Technical Report DISI-TR-00-06, Univ. of Genova, 2000.

Introduces step by step a semantic translation of UML class diagrams into Casl specifications in a way that the result may be integrated with the semantics of other kinds of diagrams.

`IFIP:1999:AFS`.

Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification.* IFIP State-of-the-Art Reports. Springer, 1999.

Presents state-of-the art surveys of the major research topics in the area of algebraic specifications, written by leading experts in the field.

`Klin:2000:ISS`.

Bartek Klin. An implementation of static semantics for architectural specifications in Casl. Master's thesis, Warsaw Univ., 2000. In Polish.

Describes algorithmic aspects of static analysis of Casl, including the cell calculus for architectural specifications.

`Klin:2001:CAC`.

Bartek Klin, Piotr Hoffman, Andrzej Tarlecki, Lutz Schröder, and Till Mossakowski. Checking amalgamability conditions for Casl architectural specifications. In J. Sgall, A. Pultr, and P. Kolman, editors, *Mathematical Foundations of Computer Science 2001, 26th International Symposium, MFCS 2001, Marianske Lazne, Czech Republic, Proceedings*, LNCS Vol. 2136, pages 451–463. Springer, 2001.

Provides static analysis for Casl architectural specifications with cell calculus.

Ledoux:2000:FSM.

Franck Ledoux, Jean-Marc Mota, Agnès Arnould, Catherine Dubois, Pascale Le Gall, and Yves Bertrand. Formal specification for a mathematics-based application domain: Geometric modeling. Technical Report 51, LaMI, Université d'Evry-Val d'Essonne, Evry, 2000.

Gives a first comparison of using Casl and the B method on the chamfering operation in topology-based modeling.

Ledoux:2000:HLO.

Franck Ledoux, Agnès Arnould, Pascale Le Gall, and Yves Bertrand. A high-level operation in 3D modeling: A Casl case study. Technical Report 52, Lami, Université d'Evry-Val d'Essonne, Evry, 2000.

Provides a Casl case study in geometric modeling and presents the different useful Casl features for geometric modeling.

Ledoux:2001:GMC.

Franck Ledoux, Agnès Arnould, Pascale Le Gall, and Yves Bertrand. Geometric modeling with Casl. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting, Genova, Italy, 2001, Selected Papers*, LNCS Vol. 2267, pages 176–200. Springer, 2001.

Gives a specification methodology dedicated to topology-based modeling. This methodology is commented and illustrated with several examples.

Ledoux:2001:SFC.

Franck Ledoux, Jean-Marc Mota, Agnès Arnould, Catherine Dubois, Pascale Le Gall, and Yves Bertrand. Spécifications formelles du chanfreinage. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL), Nancy, France*. ADER/LORIA, June 2001.

Gives a complete case study of using Casl and the B method in topology-based modeling. Includes foundations of dedicated methodology.

Ledoux:2002:SPF.

Franck Ledoux, Jean-Marc Mota, Agnès Arnould, Catherine Dubois, Pascale Le Gall, and Yves Bertrand. Spécifications formelles du chanfreinage. *Technique et Science Informatiques*, 21(8):1–26, 2002.

An extended version of [Ledoux:2001:SFC]. Gives a complete case study of using CASL and the B method in topology-based modeling. Includes foundations of dedicated methodology.

Machado:2002:UTC.

Patricia D. L. Machado and Donald Sannella. Unit testing for Casl architectural specifications. In K. Diks and W. Rytter, editors, *Mathematical Foundations of Computer Science 2002, 27th International Symposium, MFCS 2002, Warsaw, Poland, Proceedings*, LNCS Vol. 2420, pages 506–518. Springer, 2002.

Studies the problem of testing modular systems against Casl architectural specifications, focussing on unit testing.

`Mossakowski:1998:COS`.

Till Mossakowski. Colimits of order-sorted specifications. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, 1997, Selected Papers*, LNCS Vol. 1376, pages 316–332. Springer, 1998.

Proves cocompleteness of the CASL signature category and explains the relation to order-sorted algebra.

`Mossakowski:1998:SSA`.

Till Mossakowski, Kolyang, and Bernd Krieg-Brückner. Static semantic analysis and theorem proving for CASL. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, 1997, Selected Papers*, LNCS Vol. 1376, pages 333–348. Springer, 1998.

Describes the CASL tool set, including the overload resolution algorithm and encodings to higher-order logic.

`Mossakowski:1999:TOC`.

Till Mossakowski. Translating OBJ3 to CASL: The institution level. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, 13th International Workshop, WADT'98, Lisbon, Portugal, 1998, Selected Papers*, LNCS Vol. 1589, pages 198–215. Springer, 1999.

Presents different translations of OBJ3 to CASL, using different treatments of OBJ3's total retracts.

`Mossakowski:2000:CST`.

Till Mossakowski. CASL: From semantics to tools. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 6th International Conference, TACAS 2000, Berlin, Germany, Proceedings*, LNCS Vol. 1785, pages 93–108. Springer, 2000.

Gives a description of the CASL tool set and the HOL-CASL theorem prover.

`Mossakowski:2000:SAI`.

Till Mossakowski. Specification in an arbitrary institution with symbols. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 252–270. Springer, 2000.

Adds symbols to institutions, needed for CASL symbol maps.

`Mossakowski:2000:SPH`.

Till Mossakowski, Anne Haxthausen, and Bernd Krieg-Brückner. Subsorted partial higher-order logic as an extension of CASL. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 126–145. Springer, 2000.

This was the first proposal for a higher-order extension of CASL, superseded by HASCASL [`Schroeder:2002:HIS`].

`Mossakowski:2001:EDG`.

Till Mossakowski, Serge Autexier, and Dieter Hutter. Extending development graphs with hiding. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Genova, Italy, Proceedings*, LNCS Vol. 2029, pages 269–283. Springer, 2001.

Presents the kernel formalism for structured theorem proving that is used in the CASL proof calculus.

`Mossakowski:2001:IIS`.

Till Mossakowski and Bartek Klin. Institution-independent static analysis for CASL. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting, Genova, Italy, 2001, Selected Papers*, LNCS Vol. 2267, pages 221–237. Springer, 2001.

Makes the CASL tool set as much institution independent as possible.

`Mossakowski:2002:RCO`.

Till Mossakowski. Relating CASL with other specification languages: The institution level. *Theoretical Computer Science*, 286:367–475, 2002.

Provides translations from other specification languages to CASL, as well as translations among sublanguages, including those needed for the CASL tool set.

`Mossakowski:2003:ACS`.

Till Mossakowski, Horst Reichel, Markus Roggenbach, and Lutz Schröder. Algebraic-coalgebraic specification in CoCASL. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 376–392. Springer, 2003. Extended version submitted for publication.

Proposes a coalgebraic extension of CASL, including cogenerated, simple and structured cofree and modal logic.

`Mossakowski:2003:CWM`.

Till Mossakowski, Markus Roggenbach, and Lutz Schröder. CoCASL at work – modelling process algebra. In H. P. Gumm, editor, *Coalgebraic Methods in Computer Science, CMCS'03, Warsaw, Poland, Proceedings*, ENTCS Vol. 82.1. Elsevier, 2003.

Presents a case study in CoCASL, specifying CCS and CSP coalgebraically.

`Mossakowski:2003:FHS`.

Till Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 359–375. Springer, 2003.

Provides a semantics for heterogeneous specifications involving both different institutions and institution translations of different kinds.

`Mossakowski:2003:CCA`.

Till Mossakowski, Anne Haxthausen, Donald Sannella, and Andrzej Tarlecki. CASL, the Common Algebraic Specification Language: Semantics and proof theory. *Computing and Informatics*, 22:285–321, 2003.

Gives an overview of a simplified version of the CASL syntax, semantics and proof calculus, for basic, structured and architectural specifications.

`Mossakowski:2004:CASL-Logic`.

Till Mossakowski, Piotr Hoffman, Serge Autexier, and Dieter Hutter. CASL logic. In *CASL Reference Manual*, LNCS Vol. 2960 (IFIP Series), part IV. Springer, 2004. Edited by T. Mossakowski.

Presents proof calculi that support reasoning about CASL specifications; proves soundness and discusses completeness.

`Mosses:1996:CoFI`.

Peter D. Mosses. CoFI: The Common Framework Initiative for algebraic specification. *Bulletin of the EATCS*, 59:127–132, June 1996. An updated version is [`Mosses:2001:CoFI`].

Presents CoFI, describing the aims and goals.

`Mosses:1997:CAS`.

Peter D. Mosses. CASL for ASF+SDF users. In M. P. A. Sellink, editor, *ASF+SDF'97, Proc. 2nd Intl. Workshop on the Theory and Practice of Algebraic Specifications*, volume ASFSDF-97 of *Electronic Workshops in Computing*. British Computer Society, 1997.

Gives an overview of CASL, comparing it to ASF+SDF.

`Mosses:1997:CoFI`.

Peter D. Mosses. CoFI: The Common Framework Initiative for algebraic specification and development. In M. Bidoit and M. Dauchet, editors, *TAPSOFT'97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France, Proceedings*, LNCS Vol. 1214, pages 115–137. Springer, 1997.

Describes a tentative design for CASL, motivating some of the design choices.

`Mosses:1999:CGT`.

Peter D. Mosses. CASL: A guided tour of its design. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, 13th International Workshop, WADT'98, Lisbon, Portugal, 1998, Selected Papers*, LNCS Vol. 1589, pages 216–240. Springer, 1999.

Indicates the major issues in the CASL design, explains and illustrates the main concepts and constructs. Based on a $\frac{1}{2}$-day tutorial.

`Mosses:2000:CAS`.

Peter D. Mosses. CASL and Action Semantics. In P. D. Mosses and H. Moura, editors, *AS 2000, Third International Workshop on Action Semantics, Recife, Brazil, Proceedings*, BRICS NS-00-6, pages 62–78. Dept. of Computer Science, Univ. of Aarhus, 2000.

Gives an overview of CASL, and considers pros and cons of using it as meta-notation in action semantic descriptions of programming languages.

`Mosses:2000:CCU`.

Peter D. Mosses. Casl for CafeOBJ users. In K. Futatsugi, A. T. Nakagawa, and T. Tamai, editors, *CAFE: An Industrial-Strength Algebraic Formal Method*, chapter 6, pages 121–144. Elsevier, 2000.

Gives an overview of Casl, comparing it to CafeOBJ.

`Mosses:2001:CoFI`.

Peter D. Mosses. CoFI: The common framework initiative for algebraic specification and development. In G. Păun, G. Rozenberg, and A. Salomaa, editors, *Current Trends in Theoretical Computer Science: Entering the 21st Century*, pages 153–163. World Scientific, 2001.

Describes the aims, goals, and initial achievements of CoFI, extending and updating [`Mosses:1996:CoFI`].

`Reggio:1999:CLC`.

Gianna Reggio, Egidio Astesiano, and Christine Choppy. Casl-Ltl: A Casl extension for dynamic reactive systems – summary. Technical Report DISI-TR-99-34, Univ. of Genova, 1999. Revised August 2003, see [`Reggio:2003:CLC`].

Describes the Casl-Ltl extension language proposed for dynamic systems specification, with dynamic sorts and temporal formulas.

`Reggio:1999:CFD`.

Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hussmann. A Casl formal definition of UML active classes and associated state machines. Technical Report DISI-TR-99-16, Univ. of Genova, 1999. A short version is published in [`Reggio:2000:AUA`].

Presents the labelled transition system associated with an active class using Casl.

`Reggio:1999:MPU`.

Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hussmann. Making precise UML active classes modeled by state charts. Technical Report DISI-TR-99-14, Univ. of Genova, 1999.

Presents the labelled transition system associated with an active class using Casl.

`Reggio:2000:ASU`.

Gianna Reggio, Maura Cerioli, and Egidio Astesiano. An algebraic semantics of UML supporting its multiview approach. In D. Heylen, A. Nijholt, and G. Scollo, editors, *Algebraic Methods in Language Processing, AMiLP 2000*, TWLT Vol. 16. Univ. of Twente, 2000.

Using Casl as a metalanguage, proposes a semantics for class diagrams, state machines and overall systems described using the UML.

`Reggio:2000:AUA`.

Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hussmann. Analysing UML active classes and associated state machines – A lightweight approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering, Third International Conference, FASE 2000, Berlin, Germany, Proceedings*, LNCS Vol. 1783, pages 127–146. Springer, 2000. An extended version is provided in [`Reggio:1999:CFD`].

Presents the labelled transition system associated with an active class using Casl.

`Reggio:2000:CCC`.
Gianna Reggio and Lorenzo Repetto. Casl-Chart: A combination of statecharts and of the algebraic specification language Casl. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, Proceedings*, LNCS Vol. 1816, pages 243–257. Springer, 2000.

Presents a combination of statecharts and Casl.

`Reggio:2000:CCS`.
Gianna Reggio and Lorenzo Repetto. Casl-Chart: Syntax and semantics. Technical Report DISI-TR-00-1, Univ. of Genova, 2000.

Presents the complete syntax and semantics of a combination of statecharts and Casl.

`Reggio:2001:RSU`.
Gianna Reggio, Maura Cerioli, and Egidio Astesiano. Towards a rigorous semantics of UML supporting its multiview approach. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Genova, Italy, Proceedings*, LNCS Vol. 2029, pages 171–186. Springer, 2001.

Using Casl as a metalanguage, proposes a semantics for class diagrams, state machines and overall systems described using the UML.

`Reggio:2003:CLC`.
Gianna Reggio, Egidio Astesiano, and Christine Choppy. Casl-Ltl: A Casl extension for dynamic reactive systems – version 1.0 – summary. Technical Report DISI-TR-03-36, Univ. of Genova, 2003. A revision of [`Reggio:1999:CLC`].

Describes the Casl-Ltl extension language proposed for dynamic systems specification, with dynamic sorts and temporal formulae.

`Roggenbach:2000:SRN`.
Markus Roggenbach, Lutz Schröder, and Till Mossakowski. Specifying real numbers in Casl. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Château de Bonas, France, 1999, Selected Papers*, LNCS Vol. 1827, pages 146–161. Springer, 2000.

Presents a weak first-order theory of real numbers in Casl.

`Roggenbach:2001:TTS`.
Markus Roggenbach and Lutz Schröder. Towards trustworthy specifications I: Consistency checks. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting, Genova, Italy, 2001, Selected Papers*, LNCS Vol. 2267, pages 305–327. Springer, 2001.

Introduces a calculus for proving consistency of Casl specifications; the syntax-driven approach exploits in particular the Casl structuring operations

`Roggenbach:2003:CCN`.

Markus Roggenbach. CSP-Casl – A new integration of process algebra and algebraic specification. In F. Spoto, G. Scollo, and A. Nijholt, editors, *Algebraic Methods in Language Processing, AMiLP 2003*, TWLT Vol. 21, pages 229–243. Univ. of Twente, 2003.

Describes the integration of the process algebra CSP and the algebraic specification language Casl into one language, with denotational semantics in the process part and loose semantics for the datatypes.

`Roggenbach:2004:CASL-Libraries`.

Markus Roggenbach, Till Mossakowski, and Lutz Schröder. Casl libraries. In *Casl Reference Manual*, LNCS Vol. 2960 (IFIP Series), part V. Springer, 2004.

Provides libraries of basic datatypes in Casl, including order-theoretic and basic algebraic concepts, simple and structured datatypes, and graphs.

`Salauen:2002:SAC`.

Gwen Salaün, Michel Allemand, and Christian Attiogbé. Specification of an access control system with a formalism combining CCS and Casl. In *Proc. of the 7th International Workshop on Formal Methods for Parallel Programming: Theory and Applications, FMPPTA'02*, USA, 2002. IEEE Press.

Advocates a formalism which combines the CCS process algebra with the Casl algebraic specification language, presents formal foundations of this combination, and illustrates it with a real size case study: an access control system to a set of buildings.

`Sannella:2000:ASP`.

Donald Sannella. Algebraic specification and program development by stepwise refinement. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation, 9th International Workshop, LOPSTR'99, Venice, Italy, 1999 Selected Papers*, LNCS Vol. 1817, pages 1–9. Springer, 2000.

Provides an overview of formal algebraic notions of refinement step.

`Sannella:2000:CoFI`.

Donald Sannella. The common framework initiative for algebraic specification and development of software. In D. Bjørner, M. Broy, and A. V. Zamulin, editors, *Perspectives of System Informatics, Third International Andrei Ershov Memorial Conference, PSI'99, Akademgorodok, Novosibirsk, Russia, Proceedings*, LNCS Vol. 1755, pages 1–9. Springer, 2000.

Gives an overview of CoFI, with emphasis on the features of Casl.

`Sannella:2001:CoFI-RP`.

Donald Sannella. The common framework initiative for algebraic specification and development of software: Recent progress. In M. Cerioli and G. Reggio, editors, *Recent Trends in Algebraic Development Techniques, 15th International Workshop, WADT 2001, Joint with the CoFI WG Meeting, Genova, Italy, 2001, Selected Papers*, LNCS Vol. 2267, pages 328–343. Springer, 2001.

Reports on progress with CoFI during 1998-2001.

`Schroeder:2001:ACE`.

Lutz Schröder, Till Mossakowski, and Andrzej Tarlecki. Amalgamation in Casl via enriched signatures. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *Automata, Languages and Programming, 28th International Colloquium, ICALP 2001, Crete, Greece, Proceedings*, LNCS Vol. 2076, pages 993–1004. Springer, 2001. Extended version to appear in Theoretical Computer Science.

Presents definition of and results about enriched Casl, which restores the lacking amalgamation property.

`Schroeder:2001:SAS`.

Lutz Schröder, Till Mossakowski, Andrzej Tarlecki, Piotr Hoffman, and Bartek Klin. Semantics of architectural specifications in Casl. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering, 4th International Conference, FASE 2001, Genova, Italy, Proceedings*, LNCS Vol. 2029, pages 253–268. Springer, 2001. Extended version to appear in Theoretical Computer Science.

Solves the problems of Casl architectural specifications with subsorts by introducing enriched Casl and a diagram static semantics.

`Schroeder:2002:HIS`.

Lutz Schröder and Till Mossakowski. HasCasl: Towards integrated specification and development of Haskell programs. In H. Kirchner and C. Ringeissen, editors, *Algebraic Methods and Software Technology, 9th International Conference, AMAST 2002, Saint-Gilles-les-Bains, Reunion Island, France, Proceedings*, LNCS Vol. 2422, pages 99–116. Springer, 2002.

The central paper explaining HasCasl, a higher-order extension of Casl including type constructors, polymorphism and recursion.

`Schroeder:2003:CCP`.

Lutz Schröder. Classifying categories for partial equational logic. In R. Blute and P. Selinger, editors, *Category Theory and Computer Science, CTCS'02*, ENTCS Vol. 69. Elsevier, 2003.

Establishes correspondence results between partial equational theories, of which Casl signatures are a special case, and categories with certain finite limits, in preparation for the semantics of HasCasl.

`Schroeder:2003:HMP`.

Lutz Schröder. Henkin models of the partial λ-calculus. In M. Baaz and J. M. Makowsky, editors, *Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, Proceedings*, LNCS Vol. 2803, pages 498–512. Springer, 2003.

Shows that categorical models of the partial lambda-calculus and intensional Henkin models, as used in the semantics of HasCasl, are equivalent

Schroeder:2003:MID.

Lutz Schröder and Till Mossakowski. Monad-independent dynamic logic in HasCasl. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, LNCS Vol. 2755, pages 425–441. Springer, 2003. Extended version to appear in Journal of Logic and Computation.

Monad-independent dynamic logic in the framework of HasCasl; admits reasoning about termination and total correctness.

Schroeder:2003:MIH.

Lutz Schröder and Till Mossakowski. Monad-independent Hoare logic in HasCasl. In M. Pezzè, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Warsaw, Poland, Proceedings*, LNCS Vol. 2621, pages 261–277. Springer, 2003.

Hoare logic for arbitrary monads (e.g., exceptions, non-determinism, references, input/output) in the framework of HasCasl.

Schroeder:2004:ASC.

Lutz Schröder, Till Mossakowski, Andrzej Tarlecki, Bartek Klin, and Piotr Hoffman. Amalgamation in the semantics of Casl. *Theoretical Computer Science*. To appear; extends [Schroeder:2001:SAS, Schroeder:2001:ACE].

Solves the problems of Casl architectural specifications with subsorts by introducing enriched Casl and a diagram static semantics.

Schroeder:2004:MID.

Lutz Schröder and Till Mossakowski. Monad-independent dynamic logic in HasCasl. *Journal of Logic and Computation*. To appear; extends [Schroeder:2003:MID].

Monad-independent dynamic logic in the framework of HasCasl; admits reasoning about termination, partial and total correctness.

Tarlecki:2003:AST.

Andrzej Tarlecki. Abstract specification theory: An overview. In M. Pizka and M. Broy, editors, *Models, Algebras and Logic of Engineering Software*, NATO Science Series: Computer & Systems Sciences Vol. 191, pages 43–79. IOS Press, 2003.

Provides an overall view of abstract specification and software development theory, including a version of Casl architectural specifications with an example, semantics and verification rules.

References

1. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. Casl: The Common Algebraic Specification Language. *Theoretical Comput. Sci.*, 286(2):153–196, 2002.
2. E. Astesiano, M. Broy, and G. Reggio. Algebraic specification of concurrent systems. In *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, chap. 13. Springer, 1999.
3. E. Astesiano and M. Cerioli. Free objects and equational deduction for partial conditional specifications. *Theoretical Comput. Sci.*, 152:91–138, 1995.
4. H. Baumeister and A. V. Zamulin. State-based extension of Casl. In *IFM 2000*, LNCS 1945, pages 3–24. Springer, 2000.
5. M. Bidoit and P. D. Mosses. *Casl User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.
6. M. Bidoit, D. Sannella, and A. Tarlecki. Architectural specifications in Casl. *Formal Aspects of Comput.*, 13:252–273, 2002.
7. T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Comput. Sci.*, 286:197–245, 2002.
8. M. G. J. van den Brand, J. Scheerder, J. J. Vinju, and E. Visser. Disambiguation filters for scannerless generalized LR parsers. In *CC 2002*, LNCS 2304, pages 143–158. Springer, 2002.
9. P. Burmeister. Partial algebras — survey of a unifying approach towards a two-valued model theory for partial algebras. *Algebra Universalis*, 15:306–358, 1982.
10. P. Burmeister. *A Model Theoretic Oriented Approach to Partial Algebras*. Akademie-Verlag, Berlin, 1986.
11. P. Burmeister, M. Llabrés, and F. Rosselló. Pushout complements for partly total algebras. *Math. Struct. in Comput. Sci.*, 12(2):177–201, 2002.
12. M. Cerioli. *Relationships between Logical Formalisms*. PhD thesis, TD-4/93, Università di Pisa-Genova-Udine, 1993.
13. M. Cerioli, A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Permissive subsorted partial logic in Casl. In *AMAST'97*, LNCS 1349, pages 91–107. Springer, 1997.
14. M. Cerioli, T. Mossakowski, and H. Reichel. From total equational to partial first-order logic. In *Algebraic Foundations of Systems Specification*, IFIP State-of-the-Art Reports, chapter 3. Springer, 1999.

15. I. Claßen, M. Große-Rhode, and U. Wolter. Categorical concepts for parameterized partial specification. *Math. Struct. in Comput. Sci.*, 5:153–188, 1995.
16. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.cofi.info`.
17. R. Diaconescu. An institution-independent proof of Craig Interpolation Property. *Studia Logica*, 76(3), 2004.
18. S. Even. *Graph Algorithms.* Computer Science Press, 1979.
19. R. van Glabbeek. The meaning of negative premises in transition system specifications II. In *ICALP'96*, LNCS 1099, pages 502–513. Springer, 1996.
20. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *J. ACM*, 39(1):95–146, 1992.
21. J. A. Goguen and J. Meseguer. Completeness of many-sorted equational logic. *ACM SIGPLAN Notices*, 17(1):9–17, 1982.
22. J. A. Goguen and J. Meseguer. Eqlog: Equality, types, and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming. Functions, Relations and Equations*, pages 295–363. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.
23. J. A. Goguen and G. Rosu. Institution morphisms. *Formal Aspects of Comput.*, 13:274–307, 2002.
24. J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pages 80–144. Prentice Hall, 1978.
25. J. A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In J. A. Goguen and G. Malcolm, editors, *Software Engineering with OBJ: Algebraic Specification in Action*, chapter 1. Kluwer, 2000.
26. R. Harper and B. Pierce. Design issues in advanced module systems. In B. Pierce, editor, *Advanced Topics in Types and Programming Languages.* MIT Press. To appear.
27. H. Herrlich and G. Strecker. *Category Theory.* Allyn and Bacon, 1973.
28. P. Hoffman. Verifying architectural specifications. In *WADT 2001*, LNCS 2267, pages 152–175. Springer, 2001.
29. P. Hoffman. Verifying generative CASL architectural specifications. In *WADT 2002*, LNCS 2755, pages 233–252. Springer, 2003.
30. G. Kahn. Natural semantics. In K. Fuchi and M. Nivat, editors, *Programming of Future Generation Computers*, pages 237–258. North Holland, 1988.
31. B. Klin, P. Hoffman, A. Tarlecki, L. Schröder, and T. Mossakowski. Checking amalgamability conditions for CASL architectural specifications. In *MFCS 2001*, LNCS 2136, pages 451–463. Springer, 2001.
32. J. W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987.
33. V. Manca, A. Salibra, and G. Scollo. Equational type logic. *Theoretical Comput. Sci.*, 77:131–159, 1990.
34. J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.
35. J. Meseguer. Conditional rewriting as a unified model of concurrency. *Theoretical Comput. Sci.*, 96(1):73–156, 1992.
36. J. Meseguer. Membership algebra as a logical framework for equational specification. In *WADT'97*, LNCS 1376, pages 18–61. Springer, 1998.

37. T. Mossakowski. Equivalences among various logical frameworks of partial algebras. In *CSL'95*, LNCS 1092, pages 403–433. Springer, 1996.
38. T. Mossakowski. Colimits of order-sorted specifications. In *WADT'97*, LNCS 1376, pages 316–332. Springer, 1998.
39. T. Mossakowski. Specification in an arbitrary institution with symbols. In *WADT'99*, LNCS 1827, pages 252–270. Springer, 2000.
40. T. Mossakowski. Comorphism-based Grothendieck logics. In *MFCS 2002*, LNCS 2420, pages 593–604. Springer, 2002.
41. T. Mossakowski. Relating CASL with other specification languages: The institution level. *Theoretical Comput. Sci.*, 286:367–475, 2002.
42. T. Mossakowski. Foundations of heterogeneous specification. In *WADT 2002*, LNCS 2755, pages 359–375. Springer, 2003.
43. T. Mossakowski. Refinement for CASL – language summary, semantics and proof calculus. Available at `http://www.informatik.uni-bremen.de/cofi/papers/ref.pdf`, 2004.
44. T. Mossakowski, S. Autexier, and D. Hutter. Extending development graphs with hiding. In *FASE 2001*, LNCS 2029, pages 269–283. Springer, 2001.
45. T. Mossakowski, A. Haxthausen, D. Sannella, and A. Tarlecki. CASL, the Common Algebraic Specification Language: Semantics and proof theory. *Comput. and Informatics*, 22:285–321, 2003.
46. T. Mossakowski, H. Reichel, M. Roggenbach, and L. Schröder. Algebraic-coalgebraic specification in CoCASL. In *WADT 2002*, LNCS 2755, pages 376–392. Springer, 2003.
47. T. Mossakowski, D. Sannella, and A. Tarlecki. A simple refinement language for CASL. Submitted to WADT 2004, 2004.
48. P. D. Mosses. Unified algebras and institutions. In *LICS'89*, pages 304–312. IEEE, 1989.
49. P. D. Mosses. CoFI: The Common Framework Initiative for algebraic specification and development. In *TAPSOFT'97*, LNCS 1214, pages 115–137. Springer, 1997.
50. P. D. Mosses. Formatting CASL specifications using LaTeX. In [16], 2004.
51. P. Padawitz. *Computing in Horn Clause Theories*. Springer, 1988.
52. B. Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
53. T. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, 1988.
54. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL: A CASL extension for dynamic reactive systems – version 1.0 – summary. Tech. Rep. DISI-TR-03-36, Univ. of Genova, 2003.
55. H. Reichel. *Initial Computability, Algebraic Specifications and Partial Algebras*. Oxford Science Publications, 1987.
56. M. Roggenbach. CSP-CASL – A new integration of process algebra and algebraic specification. In *AMiLP 2003*, TWLT Vol. 21, pages 229–243. Univ. of Twente, 2003.
57. M. Roggenbach and T. Mossakowski. Methodological guidelines for CASL. Unpublished manuscript, 2004.
58. M. Roggenbach and L. Schröder. Towards trustworthy specifications I: Consistency checks. In *WADT 2001*, LNCS 2267, pages 305–327. Springer, 2001.
59. M. Roggenbach, L. Schröder, and T. Mossakowski. Specifying real numbers in CASL. In *WADT'99*, LNCS 1827, pages 146–161. Springer, 2000.

60. L. Schröder and T. Mossakowski. HasCasl: Towards integrated specification and development of Haskell programs. In *AMAST 2002*, LNCS 2422, pages 99–116. Springer, 2002.
61. L. Schröder, T. Mossakowski, and C. Maeder. HasCasl – Integrated functional specification and programming. Language summary. Available at `http://www.informatik.uni-bremen.de/agbkb/forschung/formal_methods/CoFI/HasCASL`, 2003.
62. L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman, and B. Klin. Semantics of architectural specifications in Casl. In *FASE 2001*, LNCS 2029, pages 253–268. Springer, 2001.
63. L. Schröder, T. Mossakowski, A. Tarlecki, B. Klin, and P. Hoffman. Amalgamation in the semantics of Casl. *Theoretical Comput. Sci.* To appear.
64. J. R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, Massachusetts, 1967.
65. A. Tarlecki. Moving between logical systems. In *WADT'95*, LNCS 1130, pages 478–502. Springer, 1996.
66. J. W. Thatcher, E. G. Wagner, and J. B. Wright. Specification of abstract data types using conditional axioms. Technical Report RC 6214, IBM Yorktown Heights, 1981.
67. E. Wagner. On the category of Casl signatures. Presentation at WADT'99, Bonas, 1999.

Index of Library and Specification Names

AbelianGroup 394
AbelianGroup_in_ConstructField 395
AbelianGroup_in_ExtCRing 397
AbelianGroup_in_Int_Add 400
AbelianGroup_in_RichCRing 399
AbelianGroup_in_Ring_add 394
AbelianGroup_in_VectorSpace 440
Algebra 449
Algebra_I 369
Algebra_II 374
Algebra_in_Matrix 450
AntisymmetricRelation 388
Array 371, 413
AsymmetricRelation 388

Bag 412
Basic/Algebra_I 368, 393
Basic/Algebra_II 431
Basic/Graphs 421
Basic/LinearAlgebra_I 439
Basic/LinearAlgebra_II 449
Basic/MachineNumbers 453
Basic/Numbers 369, 379
Basic/RelationsAndOrders 368, 387
Basic/SimpleDatatypes 401
Basic/StructuredDatatypes 405
BinTree 372, 414
BinTree2 372, 415
Boolean 401
BooleanAlgebra 390
BooleanAlgebra_in_Boolean 402
BooleanAlgebra_in_PowerSet 408

CARDINAL 453
Char 402
CommutativeMonoid 393
CommutativeMonoid_in_Bag 413
CommutativeMonoid_in_Int_Mult 399
CommutativeMonoid_in_Nat_Add 399
CommutativeMonoid_in_Nat_Mult 399
CommutativeRing 394
ConstructFactorialRing 375, 432
ConstructField 370, 394
ConstructPolynomial 433
ConstructVector 377, 443
ConstructVSWithBase 376, 441
CRing_in_CPolynomial 434

DecimalFraction 368, 384

EqRel_in_ExtCRing 370, 397
EqRel_in_ExtGroupAction 375, 436
EqRel_in_RichCRing 399
EqRel_in_RichGroupAction 436
EquivalenceRelation 389
EuclidianRing 431
EuclidianRing_in_Int 437

EuclidianRing_in_Polynomial 434
ExtAbelianGroup 396
ExtAlgebra 450
ExtBooleanAlgebra 391
ExtCARDINAL 458
ExtCommutativeMonoid 395
ExtCommutativeRing 370, 397
ExtEuclidianRing 435
ExtFactorialRing 435
ExtField 370, 398
ExtFreeAlgebra 450
ExtFreeVectorSpace 450
ExtGroup 395
ExtGroupAction 435
ExtINTEGER 458
ExtIntegralDomain 397
ExtMonoid 395
ExtMonoidAction 435
ExtPartialOrder 391
ExtRing 370, 396
ExtTotalOrder 369, 391
ExtTwoComplement 458
ExtVectorSpace 441
ExtVSWithBase 442

FactorialRing 375, 432
FactorialRing_in_ExtEuclRing 436
FactorialRing_in_RichEuclidianRing 436
Field 370, 395
Field_in_Rat 400
Finite 411
FreeAlgebra 377, 450
FreeAlgebra_in_Polynomial 451
FreeCommutativeMonoid_in_Bag 437
FreeMonoid_in_List 437
FreeVectorSpace 377, 449
FreeVectorSpace_in_VSWithBase 451

GenerateBag 371, 412
GenerateBinTree 414
GenerateBinTree2 415
GenerateList 371, 408
GenerateMap 371, 410
GenerateNTree 416
GenerateNTree2 417
GenerateSet 371, 406
Graph 372, 374, 421
GraphColorability 373, 427
GraphHomomorphism 373, 428
Graphs 372
GraphToSet 373, 423
Group 393
Group_in_ExtRing 370, 396
Group_in_RichRing 398
Group_in_SymmetricGroup 444
GroupAction 435
GroupAction_in_VectorSpace 440

Int 366, 381
INTEGER 455
IntegralDomain 394
IntegralDomain_in_Field 395
IntegralDomain_in_Int 369, 400
IntInfinity 374, 432
IrreflexiveRelation 388

K3_3 374, 429
K5 374, 429
KTree 372, 419
KTree2 372, 419

LeftTotalRelation 390
LinearAlgebra_I 375
LinearAlgebra_II 377
List 371, 408

MachineNumbers 377
Map 410
Matrix 445
Maybe 406
Minor 374, 428
Monoid 393
Monoid_in_List 410
MonoidAction 434

Nat 366, 379
NonUniqueEdgesGraph 374, 430
NTree 372, 416
NTree2 372, 417
Numbers 365

Pair 406

PartialEquivalenceRelation 389
PartialOrder 389
PartialOrder_in_Bag 413
PartialOrder_in_ExtBooleanAlgebra 392
PartialOrder_in_Set 407
Paths 373, 425
Planar 374, 429
Polynomial 434
PowerSet 371, 407
PreOrder 389
PreOrder_in_ExtCRing 369, 370, 397
PreOrder_in_ExtMonoidAction 375, 435
PreOrder_in_RichCRing 399
PreOrder_in_RichMonoidAction 436

Rat 367, 383
ReflexiveRelation 388
Relation 388
RelationsAndOrders 368
RichAbelianGroup 398
RichAlgebra 450
RichBoolean 402
RichBooleanAlgebra 392
RichCommutativeMonoid 398
RichCommutativeRing 398
RichEuclidianRing 436
RichFactorialRing 436
RichField 399
RichFreeAlgebra 451
RichFreeVectorSpace 451
RichGraph 373, 422
RichGroup 398
RichGroupAction 436
RichIntegralDomain 399
RichMonoid 398
RichMonoidAction 436
RichPartialOrder 392
RichRing 398
RichTotalOrder 369, 392
RichVectorSpace 446
RichVSWithBase 446
RightUniqueRelation 390
Ring 394

Set 406
ShortestPaths 373, 427
SimilarityRelation 389
SimpleDatatypes 370
StrictOrder 389
StrictTotalOrder 390
String 410
StructuredDatatypes 370
SymmetricClosure 373, 425
SymmetricGraph 373, 424
SymmetricGroup 444
SymmetricRelation 388

TotalMap 411
TotalOrder 369, 390
TotalOrder_in_CARDINAL 458
TotalOrder_in_Int 392
TotalOrder_in_INTEGER 458
TotalOrder_in_IntInfinity 433
TotalOrder_in_Nat 392
TotalOrder_in_Rat 392
TotalOrder_in_TwoComplement 458
TransitiveRelation 388
TwoComplement 456

Vector 443
VectorSpace 439
VectorSpace_in_Field 440
VectorSpace_in_Vector 444
VectorSpace_in_VectorTuple 446
VectorSpaceLC 376, 440
VectorTuple 442
VSWithBase 376, 441
VSWithBase_in_Field 446
VSWithBase_in_Matrix 447
VSWithBase_in_Vector 446
VSWithBase_in_VectorSpace 377, 447

Abstract Syntax Sorts and Constructors

%cons I:39
%def I:39
%implies I:39
%mono I:39

ALTERNATIVE I:15, 30, II:77, 78, 82, 83, 89, 90, III:150, 177
AMALGAMATION I:55, II:80, III:244, 259
amalgamation I:55, II:80, III:244, 259
ANNOTATION II:105
ANNOTATION-GROUP II:105
ANNOTATION-LINE II:105
APPLICATION I:23, II:77, III:166
application I:23, II:77, III:166
ARCH-SPEC I:50, II:79, 84, 92, III:232, 251
ARCH-SPEC-DEFN I:50, II:79, 84, 92, III:232, 251
arch-spec-defn I:50, II:79, 84, III:232, 251
ARCH-SPEC-NAME I:51, II:80, 84, 93, III:232
ARCH-UNIT-SPEC I:53, II:80, III:239
arch-unit-spec I:53, II:80, III:239
ARG-DECL I:13, II:76, 81, 89, III:144
arg-decl I:13, II:76, 81, III:144
assoc-op-attr I:12, II:76, 81, III:143
ATOM I:21, 31, II:77, 78, III:162, 184
AXIOM I:18, II:77, III:159
AXIOM-ITEMS I:18, II:77, III:159
axiom-items I:18, II:77, III:159

BASIC-ARCH-SPEC I:51, II:79, 92, III:233, 252
basic-arch-spec I:51, II:79, 84, III:233, 252
BASIC-ITEMS I:9, II:76, 81, 88, III:139
BASIC-SPEC I:9, II:76, 81, 88, III:138, IV:318
basic-spec I:9, II:76, 81, III:138
BINARY-OP-ATTR I:12, II:76, III:143
BRACED-ID I:25, II:78, III:168
braced-id I:25, II:78, III:168
BRACKET-ID I:25, II:78, III:168
bracket-id I:25, II:78, III:168

CAST I:32, II:78, III:187
cast I:32, II:78, III:187
CHAR II:100
CLOSED-SPEC I:40, II:79, III:210
closed-spec I:40, II:79, III:210, IV:319
CLOSED-UNIT-SPEC I:53, II:80, III:240
closed-unit-spec I:53, II:80
comm-op-attr I:12, III:143
COMMENT II:104
COMMENT-GROUP II:104
COMMENT-LINE II:104
COMMENT-OUT II:104
COMP-MIX-TOKEN I:47, II:79, 84, III:223
comp-mix-token I:47, II:79, 84, III:223
COMP-SORT-ID I:47, II:79, 84, III:223
comp-sort-id I:47, II:79, 84, III:223
COMPONENT II:89
COMPONENTS I:16, II:77, 82, III:151
CONDITIONAL I:24, II:78, III:168
conditional I:24, II:78, III:168

CONJUNCTION I:20, II:77, III:161
conjunction I:20, II:77, III:161

DATATYPE-DECL I:15, II:77, 82, 89, III:149
datatype-decl I:15, II:77, 82, III:149
DATATYPE-ITEMS I:14, II:77, III:147
datatype-items I:14, II:77, III:147
DEFINEDNESS I:22, II:77, III:164
definedness I:22, II:77, III:164
DIGIT II:99
DIGITS II:100
DIRECT-LINK I:59, II:80, III:271
direct-link I:59, II:80, 85, III:271
DISJUNCTION I:20, II:77, III:161
disjunction I:20, II:77, III:161
DOT-WORDS II:99
DOWNLOAD-ITEMS I:58, II:80, III:270
download-items I:58, II:80, III:270

empty-braces I:25, II:78, III:168
empty-brackets I:25, III:168
EMPTY-BRS I:25, II:78, III:168
EQUIVALENCE I:20, II:77, III:162
equivalence I:20, II:77, III:162
existential I:19, III:160
EXISTL-EQUATION I:22, II:77, III:165
existl-equation I:22, II:77, III:165
EXTENSION I:39, II:79, III:208
extension I:39, II:79, III:208, IV:318

false-atom I:21, III:163
FIT-ARG I:42, 44, II:79, 83, 91, III:214, 218, IV:322, 323
FIT-ARG-UNIT I:56, II:80, 84, 93, III:245, 260
fit-arg-unit I:56, II:80, 84, III:245, 260
FIT-SPEC I:42, II:79, III:214
fit-spec I:42, II:79, 83, III:214, IV:322
FIT-VIEW I:44, 45, II:79, III:218
fit-view I:44, 45, II:79, III:218, IV:323, 324
FLOATING II:101
FORMULA I:19, II:77, 82, 83, 89, 91, 95, 96, III:159
FRACTION II:101
FREE-DATATYPE I:16, II:77, III:152
free-datatype I:16, II:77, III:152
FREE-SPEC I:39, II:79, III:209
free-spec I:39, II:79, III:209, IV:319

GENERICITY I:40, II:79, 83, III:211, IV:320
genericity I:40, II:79, 83, III:211, IV:320
GROUP-ARCH-SPEC II:92
GROUP-SPEC II:91
GROUP-UNIT-TERM II:93

HEX-CHAR II:101
HIDDEN I:37, II:78, III:206
hidden I:37, II:78, III:206
hide II:83

ID I:25, II:78, 82, 90, III:168
id I:25, II:78, 82, III:168
idem-op-attr I:12, III:143
IMPLICATION I:20, II:77, III:161
implication I:20, II:77, III:161
implicit I:45, II:79, 83, III:221
IMPORTED I:40, II:79, 83, III:211
imported I:40, II:79, 83, III:211
IMPORTS IV:321
imports IV:321
INDIRECT-LINK I:59, II:80, III:271
indirect-link I:59, II:80, III:271
ISO-DECL I:29, II:78, III:176
iso-decl I:29, II:78, III:176
ITEM-NAME I:58, II:80, 85, 93, III:270
ITEM-NAME-MAP I:58, II:80, III:270
item-name-map I:58, II:80, III:270
ITEM-NAME-OR-MAP I:58, II:80, 85, 93, III:270

LABEL II:105
LETTER II:99
LIB-DEFN I:58, II:80, 85, 93, III:268, IV:359
lib-defn I:58, II:80, 85, III:268
LIB-ID I:59, II:80, 85, III:271
LIB-ITEM I:58, II:80, 85, 93, III:268, 270, IV:358
LIB-NAME I:59, II:80, 85, 93, III:271
LIB-VERSION I:59, II:80, 85, III:271
lib-version I:59, II:80, 85, III:271
LITERAL II:90
LOCAL-SPEC I:40, II:79, III:210

local-spec I:40, II:79, III:210, IV:319
LOCAL-UNIT I:56, II:80, III:244, 259
local-unit I:56, II:80, III:244, 259
LOCAL-VAR-AXIOMS I:18, II:77, III:158
local-var-axioms I:18, II:77, III:158

MEMBERSHIP I:31, II:78, III:186
membership I:31, II:78, III:186
MIX-TOKEN I:25, 47, II:78, 79, 82, 84, 90, 92, III:168, 223
MIXFIX II:90, 91

NEGATION I:21, II:77, III:162
negation I:21, II:77, III:162
NUMBER II:100

OP-ATTR I:12, II:76, 81, 89, III:143
OP-DECL I:11, II:76, III:142
op-decl I:11, II:76, 81, III:142
OP-DEFN I:13, II:76, III:144
op-defn I:13, II:76, III:144
OP-HEAD I:13, II:76, 81, 89, III:144
OP-ITEM I:11, II:76, 81, 88, III:141
OP-ITEMS I:11, II:76, III:141
op-items I:11, II:76, III:141
OP-NAME I:11, II:78, 82, 90, III:142
OP-SYMB I:23, II:78, 82, III:166
OP-TYPE I:11, II:76, 81, 89, III:142
ops-kind I:45, III:221
OPT-SIGN II:101

PARAMS I:40, II:79, 83, III:211, IV:320
params I:40, II:79, 83, III:211, IV:321
PARTIAL-CONSTRUCT I:15, II:77, III:150
partial-construct I:15, II:77, III:150
PARTIAL-OP-HEAD I:13, II:76, III:144
partial-op-head I:13, II:76, III:144
PARTIAL-OP-TYPE I:11, II:76, III:142
partial-op-type I:11, II:76, III:142
PARTIAL-SELECT I:16, II:77, III:151
partial-select I:16, II:77, III:151
PATH II:101
PATH-CHAR II:101
PATH-WORD II:101
PLACE II:90
PRED-DECL I:13, II:76, III:146
pred-decl I:13, II:76, 81, III:146
PRED-DEFN I:14, II:77, III:147
pred-defn I:14, II:77, III:147
PRED-HEAD I:14, II:77, 81, 89, III:147
pred-head I:14, II:77, 81, III:147
PRED-ITEM I:13, II:76, 81, 89, III:145
PRED-ITEMS I:13, II:76, III:145
pred-items I:13, II:76, III:145
PRED-NAME I:13, II:78, 82, 90, III:145
PRED-SYMB I:21, II:77, 82, III:164
PRED-TYPE I:14, II:76, 81, 89, III:147
pred-type I:14, II:76, 81, III:147
PREDICATION I:21, II:77, III:164
predication I:21, II:77, III:164
preds-kind I:45, III:221

QUAL-ID I:45, II:79, III:221
qual-id I:45, II:79, III:221
QUAL-OP-NAME I:23, II:78, 90, III:166
qual-op-name I:23, II:78, III:166
QUAL-PRED-NAME I:21, II:77, 90, III:164
qual-pred-name I:21, II:77, III:164
QUAL-VAR I:23, II:77, III:166
qual-var I:23, II:77, III:166
QUAL-VAR-NAME II:90
QUANTIFICATION I:19, II:77, III:160
quantification I:19, II:77, 82, III:160
QUANTIFIER I:19, II:77, 82, 89, III:160
QUOTED-CHAR II:100

REDUCTION I:37, II:78, III:206
reduction I:37, II:78, III:206, IV:318
RENAMING I:37, II:78, 83, 91, III:205
renaming I:37, II:78, 83, III:205
RESTRICTION I:37, II:78, 83, 91, III:206
RESULT-UNIT I:51, II:80, 84, III:233, 252
result-unit I:51, II:80, 84, III:233, 252
REVEALED I:37, II:79, III:206
revealed I:37, II:79, III:206

SIG-ITEMS I:10, II:76, 81, 88, III:140
SIGN II:99
SIGNS II:99
SIMPLE-ID I:25, II:78, 82, 90, III:168
SOME-GENERICS II:91
SOME-IMPORTED II:91
SOME-PARAMS II:91
SOME-SYMB-KIND II:92
SORT I:11, II:78, 82, 90, III:141
SORT-DECL I:11, II:76, III:141
sort-decl I:11, II:76, 81, III:141

SORT-GEN I:17, II:77, III:157
sort-gen I:17, II:77, III:157
SORT-ID I:25, 47, II:78, 79, 82, 84, 90, 92, III:168, 223
SORT-ITEM I:10, 29, II:76, 78, 81, 83, 88, 90, III:140, 175
SORT-ITEMS I:10, II:76, III:140
sort-items I:10, II:76, 81, III:140
SORT-LIST I:11, II:76, 81, III:142
sort-list I:11, II:76, 81, III:142
SORTED-TERM I:24, II:78, III:167
sorted-term I:24, II:78, III:167
sorts-kind I:45, III:221
SPEC I:36, 42, II:78, 83, 91, III:204, 214, IV:317, 321
SPEC-DEFN I:40, II:79, 83, 91, III:211, IV:319
spec-defn I:40, II:79, 83, III:211, IV:320
SPEC-INST I:42, II:79, III:214
spec-inst I:42, II:79, III:214, IV:321, 322
SPEC-NAME I:41, II:79, 84, 92, III:211
STRING II:100
STRONG-EQUATION I:22, II:77, III:165
strong-equation I:22, II:77, III:165
SUBSORT-DECL I:29, II:78, III:176
subsort-decl I:29, II:78, III:176
SUBSORT-DEFN I:29, II:78, III:176
subsort-defn I:29, II:78, III:176
SUBSORTS I:30, II:78, III:177
subsorts I:30, II:78, III:177
SYMB I:45, II:79, 84, 92, III:221
SYMB-ITEMS I:45, II:79, 83, 92, III:221
symb-items I:45, II:79, 83, III:221
SYMB-KIND I:45, II:79, 83, III:221
SYMB-MAP I:46, II:79, 84, 92, III:222
symb-map I:46, II:79, 84, III:222
SYMB-MAP-ITEMS I:46, II:79, 83, 92, III:222
symb-map-items I:46, II:79, 83, III:222
SYMB-OR-MAP I:46, II:79, 84, 92, III:222

TERM I:23, 32, II:77, 78, 82, 83, 89, 95, 96, III:165, 187
TERMS I:23, II:77, 82, 89, 95, III:166
terms I:23, II:77, 82, III:166
TEXT II:104
TEXT-LINE II:104
TEXT-LINES II:104
TOKEN I:25, II:78, 82, 90, III:168
TOTAL-CONSTRUCT I:15, II:77, III:150
total-construct I:15, II:77, 82, III:150
TOTAL-OP-HEAD I:13, II:76, III:144
total-op-head I:13, II:76, 81, III:144
TOTAL-OP-TYPE I:11, II:76, III:142
total-op-type I:11, II:76, 81, III:142
TOTAL-SELECT I:16, II:77, III:151
total-select I:16, II:77, 82, III:151
TRANSLATION I:37, II:78, III:205
translation I:37, II:78, III:205, IV:318
true-atom I:21, II:77, III:163
TRUTH I:21, II:77, III:163
TYPE I:45, II:79, 84, 92, III:221

UNION I:38, II:79, III:207
union I:38, II:79, III:207, IV:318
unique-existential I:19, III:160
UNIT-APPL I:56, II:80, III:245, 260
unit-appl I:56, II:80, III:245, 260
UNIT-BINDING I:54, II:80, 84, 93, III:240, 255
unit-binding I:54, II:80, 84, III:240, 255
UNIT-DECL I:52, II:79, 84, 92, III:235, 253
unit-decl I:52, II:79, 84, III:235, 253
UNIT-DECL-DEFN I:51, II:79, 84, 92, III:233, 252
UNIT-DEFN I:52, II:80, 84, 92, III:236, 254
unit-defn I:52, II:80, 84, III:236, 254
UNIT-EXPRESSION I:54, II:80, 84, 93, III:240, 255
unit-expression I:54, II:80, 84, III:240, 255
UNIT-IMPORTED I:52, II:80, 84, III:235, 253
unit-imported I:52, II:80, 84, III:235, 253
UNIT-NAME I:52, II:80, 84, 93, III:235, 253
UNIT-OP-ATTR I:12, II:76, III:143
unit-op-attr I:12, II:76, III:143
UNIT-REDUCTION I:55, II:80, III:243, 258

`unit-reduction` I:55, II:80, III:243, 258

`UNIT-SPEC` I:52, II:80, 84, 92, III:237

`UNIT-SPEC-DEFN` I:52, II:80, 84, 92, III:237

`unit-spec-defn` I:52, II:80, 84, III:237

`UNIT-TERM` I:54, II:80, 84, 93, III:242, 257

`UNIT-TRANSLATION` I:55, II:80, III:242, 257

`unit-translation` I:55, II:80, 84, III:242, 257

`UNIT-TYPE` I:53, II:80, 84, III:238

`unit-type` I:53, II:80, 84, III:238

`universal` I:19, II:77, 82, III:160

`URL` II:101

`VAR` I:18, II:78, 82, 90, III:158

`VAR-DECL` I:18, II:77, 82, 89, III:158

`var-decl` I:18, II:77, 82, III:158

`VAR-ITEMS` I:17, II:77, III:158

`var-items` I:17, II:77, III:158

`VERSION-NUMBER` I:59, II:80, 85, 93, III:271

`version-number` I:59, II:80, 85, III:271

`VIEW-DEFN` I:43, II:79, 83, 91, III:217, IV:322

`view-defn` I:43, II:79, 83, III:217, IV:323

`VIEW-NAME` I:44, II:79, 84, 92, III:217

`VIEW-TYPE` I:43, II:79, 83, 91, III:217, IV:323

`view-type` I:43, II:79, 83, III:217, IV:323

`WORD` II:99

`WORD-CHAR` II:99

`WORDS` II:99

Symbol Index

(Δ, Ψ)
many-sorted enrichment III:135
subsorted enrichment III:175
$(\Sigma_I, \langle \Sigma_1, \ldots, \Sigma_n \rangle, \Sigma_B)$ generic signature III:202
$(\Sigma^I, \Sigma_1, \ldots, \Sigma_n {\to} \Sigma)$ signature of generic unit with import III:228
$(\Sigma^I, \overline{\Sigma} {\to} \Sigma)$ signature of generic unit with import III:228
$(\Sigma^I, U\Sigma)$ signature of generic unit with import III:228
(Σ_s, σ, GS_s) static denotation of a view III:203
$(\sigma^{\mathsf{S}}, \sigma^{\mathsf{TF}}, \sigma^{\mathsf{PF}}, \sigma^{\mathsf{P}})$ many-sorted signature morphism III:126
$(a_1, \ldots, a_n)$ n-tuple III:116
$\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ function III:116
$(C_s, U\Sigma)$ architectural signature III:228
(E, U) architectural model III:230
$(\mathcal{G}_m, \mathcal{V}_m, \mathcal{A}_m, \mathcal{T}_m)$ model global environment III:266
$(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s)$ static global environment III:266
$(\mathcal{G}_s, \mathcal{V}_s, \mathcal{A}_s, \mathcal{T}_s), (\mathcal{DG}, \mathit{Th})$ verification global environment IV:316
(k, Ident) identifier as a symbol III:191
$(\mathcal{M}_I, \langle \mathcal{M}_1, \ldots, \mathcal{M}_{n'} \rangle, \mathcal{M}_B)$ model semantics for a generic specification III:203
$(\mathcal{M}_s, GS_m)$ model semantics of a view III:203
$(N_1, \mathcal{DG}_1) \leadsto^J (N_2, \mathcal{DG}_2)$ refinement between nodes in a development graph IV:350
$(p, \Sigma_1, \ldots, \Sigma_n {\to} \Sigma)$ based signature of generic unit with import III:249
$(p, \overline{\Sigma} {\to} \Sigma)$ based signature of generic unit with import III:249
$(p, U\Sigma)$ based signature of generic unit with import III:249
(P_s, B_s, D) extended static unit context III:249
$(P_s, B_s, D) + (P'_s, B'_s, D')$ addition of extended static unit contexts III:249
(S', F') sort-generation constraint I:8, III:134
(S', F', σ) sort-generation constraint III:134
$(S, \mathit{TF}, \mathit{PF}, P)$
many-sorted signature extension III:125
many-sorted signature fragment III:125
many-sorted signature I:6, III:124
$(S, \mathit{TF}, \mathit{PF}, P) \cup (S', \mathit{TF}', \mathit{PF}', P')$ union of many-sorted signature fragments III:125
$(S, \mathit{TF}, \mathit{PF}, P, \leq)$
subsorted signature extension III:170
subsorted signature fragment III:170
subsorted signature I:27, III:169

$(S, TF, PF, P, \leq) \cup (S', TF', PF', P', \leq')$ union of subsorted signature fragments III:170

(S^M, F^M, P^M) many-sorted model III:128

(w, s) function profile I:6, III:124

$\{x^1_{s_1}, \ldots, x^n_{s_n}\}$ sorted variable set III:131

$\neg\varphi$ negation formula III:133

$||\Sigma||$ symbols in a signature III:192

$||\sigma||$ symbol map induced by a signature morphism III:195

$|\Sigma|$

signature symbols in a many-sorted signature III:126

signature symbols in a signature I:5

signature symbols in a subsorted signature III:171

$|\Sigma|$ signature symbols in a subsorted signature III:191

$|\sigma|$

function on signature symbols arising from a many-sorted signature morphism III:128

function on signature symbols arising from a subsorted signature morphism III:171, 191

$|A|$ cardinality of a set A III:116

$|ASP|$ architectural specification without axioms IV:338

$|\mathcal{C}|$ class of objects in a category III:117

$|w|$ length of a sequence III:116

$\leq$ subsort embedding I:27, III:169

$\langle a_1, \ldots, a_n \rangle$ sequence III:116

$\langle M_1, \ldots, M_n \rangle$ compatible models III:229

$\langle s_1, \ldots, s_n \rangle^M$ product of carrier sets III:129

$=$ token for a language with equality I:62

$\exists! X.\varphi$ unique-existential quantification III:133

$\exists X.\varphi$ existential quantification III:133

$\forall x_s.\varphi$ universal quantification III:132

$[\![t]\!]_\rho$ value of a term in a many-sorted model with respect to an assignment III:136

$\Box$ dummy denotation in extended static semantics IV:335

$\vdash_\Sigma$ entailment relation 291

$\models$ satisfaction relation of an institution 5, 123

$\emptyset$

empty semantic object III:117

empty signature III:193

Γ context IV:334, 348

$\Gamma[B'/B]$ substitution of unit names in context IV:334

Γ_{gen} generic context IV:334, 348

$\Gamma_s, \Gamma_m : \mathtt{SPEC} \models \Psi$ semantic consequence of a specification 326

$\Gamma_s, \Gamma_m : \mathtt{SPEC}_1 \overset{\rightsquigarrow}{\rightsquigarrow} \mathtt{SPEC}_2$ refinement between specifications IV:326

$\Gamma_s : \mathtt{SPEC} \vdash \Psi$ provability from a specification 326

$\Gamma_s : \mathtt{SPEC}_1 \rightsquigarrow \mathtt{SPEC}_2$ refinement proof IV:327

Δ

many-sorted signature extension III:125

subsorted signature extension III:170

$\Delta \cup \Delta'$

union of many-sorted signature extensions III:125

union of subsorted signature extensions III:170

Γ_m model global environment III:266

Γ_s static global environment III:266

$\eta(R(\mathtt{SPEC})) \rightsquigarrow^J_\Sigma \bigcup_{i=1\ldots n} \eta_i(\overline{R}(\mathtt{SPEC}_i))$ refinement between translated specifications IV:328

$\iota_{\Sigma \subseteq \Sigma \cup \Sigma'}$ injection III:193

$\nu \setminus Z$ removing variables from substitution IV:280

$\nu \colon X \longrightarrow T_\Sigma(Y)$ substitution IV:279

ρ assignment III:135

$\rho : X \to M$ assignment III:135

$\rho[x_s \mapsto a]$ augmented assignment III:135

Σ

many-sorted signature I:6, III:124

signature in an institution I:5

subsorted signature I:27, III:169

- σ
  - many-sorted signature morphism I:7, III:126
  - signature morphism in an institution I:6
  - subsorted signature morphism I:28, III:171
- $\sigma'(S', F', \sigma)$
  - translation of a constraint along a many-sorted signature morphism III:134
  - translation of a constraint along a subsorted signature morphism III:174
- $\Sigma_1, \ldots, \Sigma_n \to \Sigma$ generic unit signature III:228
- $\Sigma \hookrightarrow \Sigma'$
  - many-sorted signature inclusion III:127
  - subsorted signature inclusion III:171
- $\overline{\Sigma} \to \Sigma$ generic unit signature III:228
- $\sigma(\Delta)$ extension of σ along Δ III:199
- $\sigma(\varphi)$
  - translation of a formula along a many-sorted signature morphism III:134
  - translation of a formula along a subsorted signature morphism III:174
- $\Sigma^{\#}$ many-sorted signature associated with a subsorted signature I:28, III:172
- $\sigma^{\#}$ many-sorted signature morphism associated with a subsorted signature morphism I:28, III:172
- $\Sigma_A(\Delta)$ extension of Σ_A along Δ III:199
- Σ^N signature of a node IV:293
- σ^{P} predicate symbol map III:126
- σ^{PF} partial function symbol map III:126
- σ^{S} sort map III:126
- σ^{TF} total function symbol map III:126
- $\Sigma \cup \Delta$
  - union of a many-sorted signature and a many-sorted signature extension III:125
  - union of a subsorted signature and a subsorted signature extension III:170
- $\Sigma \cup \Sigma'$
  - union of many-sorted signatures III:125
  - union of subsorted signatures III:170, 193
- $\Sigma|^{SSYs}$ signature co-generated in a signature by a set of signature symbols 198
- $\Sigma|_{SSYs}$ signature generated in a signature by a set of signature symbols 197
- φ formula III:131
- $\varphi \to t \mid t'$ conditional term III:131
- $\varphi \Leftrightarrow \varphi'$ equivalence formula III:133
- $\varphi \Rightarrow \varphi'$ implication formula III:132
- $\varphi[\nu]$ application of substitution to formula IV:280
- $\varphi \wedge \varphi'$ conjunction formula III:133
- $\varphi \vee \varphi'$ disjunction formula III:133
- $\Phi \vdash \varphi$ entailment 281
- $\Phi \models_{\Sigma} \varphi$ semantic consequence 283
- Ψ set of sentences III:135
- ψ sentence III:134
- Ψ^N local axioms of a node IV:293

- A alternative construct I:15
- $A_1 \times \cdots \times A_n$ Cartesian product III:116
- $A \overset{\text{fin}}{\to} B$ set of finite maps from A to B. III:116
- $A \rightharpoonup B$ set of partial functions from A to B III:116
- $A \to B$ set of total functions from A to B III:116
- $A\Sigma$ architectural signature III:228
- $A : \Sigma$ unit name A has signature Σ IV:348
- $A[B'/B]$ substitution of one unit name for another IV:334
- AM architectural model III:230
- $\mathcal{AM}$ architectural specification III:230
- a qua C injection into a union of syntactic categories III:117
- $\mathbf{ArchMod}(C_s, U\Sigma)$ domain of architectural models over a static

unit context and a unit signature III:230
ArchSig domain of architectural signatures III:228
ArchSpec domain of architectural specifications III:230
ArchSpec$(A\Sigma)$ domain of architectural specifications over an architectural signature III:230
ArchSpecName domain of architectural specification names III:228
ASN architectural specification name I:51, III:228
ASP architectural specification I:51
Assignment domain of assignments III:135
Atom atomic logic axioms I:62
$A \uplus B$ disjoint union III:117
$A \cup B$ union III:116
$Ax(T)$ axioms of a theory IV:291

BasedParUnitSig domain of based signatures of generic units with import III:249
BI basic item I:9
B_s static based unit context III:249

C
  constructor component I:15
  token for a language with sort generation constraints I:62
  unit context III:231
$C^{\emptyset}$ empty unit context III:231
$\mathcal{C}_s^{\emptyset}$ empty extended static unit context III:249
$C_s^{\emptyset}$ empty static context III:229
$C[UN/\mathcal{U}]$ extension of unit context by unit declaration III:231
$C[UN/UEv]$ extension of unit context by unit definition III:231
Carrier domain of carriers III:128
Carriers domain of sort-indexed families of carriers III:128
CAT quasicategory of categories III:117
$complete(f, S)$ completion of a function to a larger domain III:117
CompMod$(\Sigma_1, \ldots, \Sigma_n)$ domain of compatible models over $\Sigma_1, \ldots, \Sigma_n$ III:229
Cond positive conditional logic axioms without predicates I:62
Constraint domain of sort-generation constraints III:134
$context \vdash phrase \Rightarrow result$ model semantics judgement 119
$context \vdash phrase \rhd\!\!\rhd\!\!\rhd\, result$ verification semantics judgement 317
$context \vdash phrase \rhd\!\!\rhd\, result$ extended static semantics judgement 251
$context \vdash phrase \rhd result$ static semantics judgement 119
$\mathcal{C}_s$ extended static unit context III:249
C_s static unit context III:228
$ctx(P_s, B_s, D)$ static unit context in an extended static unit context III:249

D signature diagram III:248
DD datatype declaration I:14
$\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$ development graph IV:293
$dgm(P_s, B_s, D)$ diagram of an extended static unit context III:249
$\mathcal{DG} \vdash L$ derivation relation for development graphs 298
Diag domain of signature diagrams III:248
$\mathrm{dom}(\Gamma)$ domain of a context IV:334
$Dom(\mathcal{C}_s)$ domain of an extended static unit context III:249
$Dom(f)$ domain of a function III:116
$D(t)$ definedness formula III:133

E unit environment III:230
e edge in a diagram III:248
$Edges(D)$ set of edges of $Shape(D)$ III:248
$EmptyExplicit(\Sigma^{basic}, SSY)$ forget signature symbols component III:201
Enrichment domain of many-sorted enrichments III:135
Eq atomic logic axioms without predicates I:62

Extension domain of many-sorted signature extensions III:125

$Ext(h)$ extension of symbol map along a generic specification III:224

$ExtID(h)$ extension of identifier map along a generic specification III:224

$Ext(r)$ extension of a symbol map III:196

F

formula I:19

generic unit III:229

f function name I:7, III:124

$F \oplus M'$ amalgamation of a generic unit and a compatible model III:229

$f \circ g$ composition of morphisms III:117

FA fitting argument I:42

false falsity III:132

FAU fitting argument unit I:56

FI file identifier I:60

$FinSeq(A)$ set of finite sequences of elements from A III:116

$FinSet(A)$ set of finite subsets of A III:116

F^M function symbol-indexed family of functions III:128

FM fitting morphism I:43

f^M function symbol interpretation I:7, III:128

$F^M_{ws}(f)$ function symbol interpretation III:128

FOAlg first-order logic axioms without predicates I:62

FOL first-order logic axioms I:62

Formula domain of formulas III:131

f^{p}_{ws} qualified partial function symbol III:191

FQTerm domain of fully-qualified terms III:131

f^{t}_{ws} qualified total function symbol III:191

FunName universe of function symbols III:124, 142, 172

FunProfile domain of function profiles III:124

FunSet domain of function symbol sets III:124

$FV(\varphi)$ free variables of a formula III:133

$FV(t)$ free variables of a term III:133

f_{ws} qualified function symbol III:126

$f_{ws}\langle t_1, \ldots, t_n \rangle$ term formed by function application III:131

$f_{ws} \sim_F f_{ws'}$ overloading relation on qualified operation symbols I:27, III:171

$f(x)$ function application III:116

f_x xth item in an indexed family III:116

GCond generalized positive conditional logic axioms without predicates I:62

GD global directory III:268

GenSig domain of generic signatures III:202

GenSpec domain of semantic objects underlying generic specification III:203

GHorn generalized positive conditional logic axioms I:62

GlobalDir domain of global directories III:268

$graph(f)$ graph of a function III:116

GS_m model semantics for a generic specification III:203

$GS_m((\mathcal{M}^A_1, \sigma_1), \ldots, (\mathcal{M}^A_n, \sigma_n))$ application of model semantics for a generic specification to fitting arguments III:203

GS_s generic signature III:202

$GS_s((\Sigma^A_1, \sigma_1), \ldots, (\Sigma^A_n, \sigma_n))$ application of generic signature to fitting arguments III:202

h

many-sorted homomorphism I:7, III:129

signature symbol map III:195

Homomorphism domain of many-sorted homomorphisms III:129

Horn positive conditional logic axioms I:62

$h|_\Sigma$
- reduct of a homomorphism with respect to a many-sorted signature inclusion 130
- reduct of a homomorphism with respect to a signature inclusion 194

$h|_\sigma$
- reduct of a homomorphism with respect to a many-sorted signature morphism 130
- reduct of a subsorted homomorphism with respect to a signature morphism 174

I identifier I:46

id_A identity morphism on A III:117

IDAsSym(*Ident*) identifier as a symbol III:192

ImpUnitSig domain of signatures of generic units with import III:228

IN item name I:59, III:266

ItemName domain of item names III:266

k kind of symbol III:191

$ker(f)$ kernel of a function III:116

$\mathcal{L}$ set of links in a development graph IV:293

LI
- library identifier III:267
- library item I:58

LibId domain of library identifiers III:267

LibName domain of library names III:267

LN library name I:58, III:267

M
- many-sorted model I:7, III:128
- subsorted model III:173
- unit III:229

$\mathcal{M}$
- class of many-sorted models III:128
- class of models in an institution I:38, III:202

$M_1 \oplus \ldots \oplus M_n$ amalgamation of compatible models III:229

$M \cong M'$ isomorphism III:131

$\mathcal{M}_\perp$ class of models over the empty signature III:193

$\overline{M}$ compatible models III:229

matches matching relation between signature symbols and symbol III:192

MEv model evaluator III:230

Mod
- model functor in many-sorted institution III:130
- model functor of an institution III:123

$\mathbf{Mod}(\Sigma)$
- category of Σ-models of an institution I:5, III:123
- category of many-sorted Σ-models I:7, III:129

$\mathbf{Mod}(\sigma)$
- translation of models in an institution I:6, III:123

$\mathbf{Mod}_\perp(\Sigma)$ partial model functor IV:339

$\mathbf{Mod}(D)$ class of all $Nodes(D)$-indexed model families consistent with D III:249

$\mathbf{Mod}_{\mathcal{DG}}(N)$ model class of a node in a development graph IV:294

Model domain of many-sorted models III:128

ModelClass domain of classes of many-sorted models III:128

ModEval domain of model evaluators III:230

$\mathbf{ModEval}(\Sigma)$ domain of model evaluators over a signature III:230

$M \models (S', F', \sigma)$ satisfaction of a sort-generation constraint 137

$M \models_\rho \varphi$
- satisfaction of a formula by a many-sorted model under an assignment 136
- satisfaction of a formula by a subsorted model under an assignment 175

$M \models \psi$
- satisfaction of a sentence by a many-sorted model 137

satisfaction of a sentence by a subsorted model 175

$M|_{\Sigma}$
reduct of a model with respect to a many-sorted signature inclusion 130
reduct of a model with respect to a signature inclusion 194

$M|_{\sigma}$
reduct of a model with respect to many-sorted signature morphism 130
reduct of a subsorted model with respect to a signature morphism 174

N version number I:59
$\mathcal{N}$ set of nodes in development graph IV:293
$N \Rightarrow \Psi$ local implication IV:295
$name(SSY)$ name of a signature symbol III:192
$Nodes(D)$ set of objects of $Shape(D)$ III:248

$O \overset{\sigma}{-\,-\!\!\succ} N$ local theorem link IV:295
$O \xrightarrow{\sigma} N$ local definition link IV:293
$O \overset{\sigma}{=\,=\!\Rightarrow} N$ global theorem link IV:295
$O \overset{\sigma}{\Longrightarrow\!\!\!\Rightarrow} N$ global reachability IV:293
$O \xRightarrow{\sigma} N$ global definition link IV:293
$O \xRightarrow[cons]{\sigma} N$ conservative extension (definition link) IV:297
$O \overset{\sigma}{\underset{cons}{=\,=\!\Rightarrow}} N$ conservative extension (theorem link) IV:296
$O \xRightarrow[def]{\sigma} N$ definitional extension (definition link) IV:297
$O \overset{\sigma}{\underset{def}{=\,=\!\Rightarrow}} N$ definitional extension (theorem link) IV:296
$O \xRightarrow[free]{\sigma} N$ free definition link IV:293
$O \overset{\sigma}{\underset{free\ \theta}{=\,=\!\Rightarrow}} N$ free theorem link IV:295
$O \xRightarrow[hide]{\sigma} N$ hiding definition link IV:293
$O \overset{\sigma}{\underset{hide\ \theta}{=\,=\!\Rightarrow}} N$ hiding theorem link IV:295
$O \xRightarrow[mono]{\sigma} N$ monomorphic extension (definition link) IV:297
$O \overset{\sigma}{\underset{mono}{=\,=\!\Rightarrow}} N$ monomorphic extension (theorem link) IV:296
$O{>}\!\overset{\sigma}{\Longrightarrow\!\!\!\Rightarrow} N$ local reachability IV:294
OI operation item I:11
OS operation symbol I:23

P
set of predicate symbols III:124
token for a language with partiality I:62

p
node in a diagram III:248
path III:267
predicate name I:7, III:124

$\overline{\mathcal{P}}(UT)$ set of generic unit names not used IV:341
PartialFun domain of function symbol interpretations III:128
PartialFuns domain of function symbol-indexed families of functions III:128
$ParUnitSig$ domain of generic unit signatures III:228
$Path$ domain of paths III:267
PF set of partial function symbols III:124
$PFMap$ domain of partial function symbol maps III:126
PF_{ws} set of partial function symbols with profile ws I:6, III:124
PI predicate item I:13
P^{M} predicate symbol-indexed family of predicates III:128
p^{M} predicate symbol interpretation I:7, III:128
$PMap$ domain of predicate symbol maps III:126
$P^{M}_{w}(p)$ predicate symbol interpretation III:128
Pred domain of predicate symbol interpretations III:128

PredName universe of predicate symbols III:124, 146, 172
PredProfile domain of predicate profiles III:124
Preds domain of predicate symbol-indexed families of predicates III:128
PredSet domain of predicate sets III:124
Pres category of presentations IV:291
P_s based static context for generic units III:249
PS predicate symbol I:21
$\mathcal{P}(UT)$ set of generic unit names used IV:341
P_w set of predicate symbols with profile w I:6, III:124
p_w qualified predicate symbol III:126, 191
$p_w\langle t_1, \ldots, t_n\rangle$ formula formed by predicate application III:132
$p_w \sim_P p_{w'}$ overloading relation on qualified predicate symbols I:27, III:171

q node in a diagram III:248
QualFunName domain of qualified function symbols III:126
QualPredName domain of qualified predicate symbols III:126
QualVarName domain of qualified variable names III:131

R
- renaming I:55
- restriction I:55

r symbol map III:195
$R = (\Phi, \alpha, \beta)$ institution comorphism IV:291
$\overline{R}(\mathcal{SP})$ specification augmented by translation axioms IV:349
$R(\mathcal{DG})$ translation of a development graph along a comorphism IV:297
$r|^{\Sigma}$ induced signature morphism 198
$r|^{\Sigma}_{\Sigma'}$ induced signature morphism 199

S set of sorts I:6, III:124
s sort name III:124, 191

Sen
- sentence functor in many-sorted institution III:135
- sentence functor of an institution III:123

Sen(Σ)
- set of Σ-sentences of an institution I:5, III:123
- set of many-sorted Σ-sentences I:8, III:132

Sen(σ)
- translation of sentences in an institution I:6, III:123

Sentence domain of sentences III:134
Set category of sets III:117
$Set(A)$ set of all subsets of A III:116
$Shape(D)$ shape category of a signature diagram III:248

SI
- signature item I:17
- sort item I:10

Sig
- category of many-sorted signatures III:128
- category of signatures of an institution I:5, III:123

SigFragment domain of many-sorted signature fragments III:125
Signature domain of many-sorted signatures III:124
SignatureMorphism domain of many-sorted signature morphisms III:126
SigSym domain of signature symbols III:126
SigSymMap domain of signature symbol maps III:195
$Sig(T)$ signature of a theory IV:291
SL symbol list I:37
S^M sort-indexed family of carriers III:128
SM symbol mapping I:37
s^M carrier set (sort interpretation) I:7, III:128
SMap domain of sort maps III:126
$S^M(s)$ carrier set (sort interpretation) III:128
SN specification name I:41
Sort universe of sorts III:124, 141

SortRelation domain of subsort embeddings III:169
SortSet domain of sort sets III:124
SP specification I:37
StBasedUnitCtx domain of static based unit contexts III:249
ExtStUnitCtx domain of extended static unit contexts III:249
StParUnitCtx domain of based static contexts for generic units III:249
$strip(GS_s)$ stripping down a verification generic signature to a generic signature IV:315
$strip(V_s)$ stripping down a verification view signature to a view signature IV:316
StUnitCtx domain of static unit contexts III:228
Sub token for a language with subsorting I:62
SubMod model functor in subsorted institution III:174, 191
SubMod(Σ) category of subsorted Σ-models III:174
SubSen sentence functor in subsorted institution III:174, 191
SubSen(Σ) set of subsorted Σ-sentences III:174
SubSig domain of subsorted signatures III:169
SubSig category of subsorted signatures III:171, 191
SubsortedExtension domain of subsorted signature extensions III:170
SubsortedSigFragment domain of subsorted signature fragments III:170
SY symbol I:40, III:191
Sym domain of symbols III:191
$SymAsSigSym(SY)$ symbol as a signature symbol III:192
SymKind domain of symbol kinds III:191
SymMap domain of symbol maps III:195

T term I:23
t fully-qualified term III:131
$t \overset{\mathrm{e}}{=} t'$ existential equation III:132
$t \overset{\mathrm{s}}{=} t'$ strong equation III:133
$t[\nu]$ application of substitution to term IV:280
TF set of total function symbols III:124
TFMap domain of total function symbol maps III:126
TF_{ws} set of total function symbols with profile *ws* I:6, III:124
Th category of theories IV:291
$Th_{\mathcal{DG}}(N)$ theory of a node in a development graph IV:294
true truth III:133
TY operation type I:11

U unit III:229
$\mathcal{U}$ unit specification III:230
u URL III:267
$\mathcal{U} \oplus MEv$ import extension of a unit specification by a model evaluator III:230
$U\Sigma$ unit signature III:228
UD unit declaration or definition I:51
UE unit expression I:51
UEv unit evaluator III:230
U_m model universal environment III:267
UN unit name I:52, III:228
Unit domain of units III:229
unit singleton set III:116
Unit(Σ) domain of units over a signature III:229
Unit$(\Sigma_1, \ldots, \Sigma_n \rightarrow \Sigma)$ domain of generic units over a generic unit signature III:229
Unit$_\perp(\Sigma \rightarrow \Sigma')$ set of partial unit functions IV:339
UnitCtx domain of unit contexts III:231
UnitEnv domain of unit environments III:230
UnitEnv$(\mathcal{C}_s)$ domain of unit environments over an extended static context III:250
UnitEnv(C_s) domain of unit environments over a static unit context III:230
UnitEval domain of unit evaluators III:230

UnitEval($U\Sigma$) domain of unit evaluators over a unit signature III:230
UnitName domain of unit names III:228
UnitSig domain of unit signatures III:228
UnitSpec domain of unit specifications III:230
UnitSpecName domain of unit specification names III:228
UnitSpec($U\Sigma$) domain of unit specifications over a unit signature III:230
$UnivEnv_m$ domain of model universal environments III:267
$UnivEnv_s$ domain of static universal environments III:267
Url domain of URLs III:267
U_s static universal environment III:267
USN unit specification name III:228
USP unit specification I:52

v
variable I:23
version III:267
Var universe of variable names III:131
Variables domain of sorted variable sets III:131
VD variable declaration I:17
VerArchSig verification architectural signatures IV:358
VerGenSig verification generic signature IV:312
Version domain of versions III:267
VerUnitSig verification unit signatures IV:358
VerViewSig verification view signature IV:316
ViewSig domain of static semantic objects underlying views III:203
ViewSpec domain of model semantic objects underlying views III:203
V_m model semantics of a view III:203
VN view name I:43
V_s static denotation of a view III:203

W map from constructors to sets of partial selectors III:148
w predicate profile I:6, III:124
w^M product of carrier sets I:7
ws function profile III:124

X sorted variable set I:7, III:131
x variable III:131
$X + \{x_s\}$ extending a sorted variable set with overriding III:131
$X + X'$ combining sorted variable sets with overriding III:131
x_s qualified variable name III:131

Concept Index

$\perp$-Σ-model IV:339
$\perp$-unit family consistent with Γ_{gen} IV:341
ω-complete partial order V:368
ω-cpo V:368
n-colorable graph V:373

abbreviated
  annotation II:103
  comment II:103
  grammar II:81
Abelian group V:369
abstract syntax
  relationship to concrete syntax II:73
abstract syntax II:73, 75
accidental inconsistency III:181
action
  group V:374
  monoid V:374
addition
  admissible III:249
  compatible III:250
admissible addition III:249
admit weak amalgamation IV:291
algebra
  free V:377
  linear V:375, 377
  many-sorted partial I:7
  over a field V:377
algebra V:369, 374
alternative
  sequence II:87
  subsort I:30
alternative I:15, 30, III:150, 177
amalgamability, ensures III:250, IV:334
amalgamate I:50
amalgamation
  of compatible models III:229
  unit I:55, III:244
amalgamation I:55, III:259
ambiguity
  architectural specification II:94
  formula II:94
  structured specification II:94
  term II:94
ambiguity I:24, II:93
analysis
  lexical II:88, 97
  mixfix grouping II:88, 93, 95
annotation
  abbreviated II:103
  associativity II:107
  authors II:111
  bracket II:103
  conflicting II:105
  conservative extension I:39, II:110
  date II:111
  definitional extension I:39, II:111
  display I:25, II:106
  extension II:110
  global II:105
  grouping II:103
  HTML II:106
  implicit associativity II:108
  implied axiom II:110

- implied extension I:39, II:111
- label II:106
- LaTeX II:106
- list II:109
- literal syntax II:108
- local associativity II:108
- mixfix display II:106
- monomorphic extension I:39, II:111
- non-nested II:103
- number II:108
- parsing II:106
- precedence II:107
- preceding II:104
- RTF II:106
- semantic II:110, III:208
- single-line II:105
- string II:109
- trailing II:104

annotation II:103, 105
application
- operation I:23, III:166
- predicate I:8, 21, III:163
- unit I:56, III:245, 260

applicative semantics III:262, IV:338
architectural
- basic specification I:51, III:233, 252
- concept I:49
- model III:230
- signature III:228
- specification definition I:51, III:232, 251
- specification model I:50
- specification name I:51
- specification reference I:51
- specification I:49, 50, III:227, 232
- unit specification I:53, III:239

argument
- fitting list I:42
- fitting morphism I:42
- fitting specification I:42
- fitting view I:45
- fitting I:56, III:245
- sort I:6, III:124
- specification I:34
- unit III:245

arguments
- compatible fitting morphisms I:42
- compatible I:50

array V:371
ASCII II:97
assertion, definedness I:8
assignment III:135
associated element V:370, 375
associativity
- annotation II:107
- attribute I:12, II:108
- implicit annotation II:108
- local annotation II:108

associativity III:143
assumed property I:49
atomic
- formula I:8, 21, 31, III:184
- logic I:67

atomic formula III:162
attribute
- associativity I:12, II:108
- commutativity I:12
- idempotency I:12
- operation I:12, III:143
- unit I:12

authors annotation II:111
auxiliary symbol I:36
axiom
- implied annotation II:110
- list I:18
- of choice V:377

axiom I:5, 18, 31, III:159, 184

bag V:371, 374
base V:375, 377
basic
- algebraic structure V:369
- architectural specification I:51, III:233, 252
- datatypes V:363
- item I:9
- many-sorted specification I:9, III:138
- specification concept I:5
- specification framework I:5
- specification semantics I:6
- specification I:5, III:123
- subsorted specification I:27, III:175

binary tree V:372
bipartite graph V:373
body, specification I:34, 41
braces, grouping I:36
bracket

  annotation II:103
  comment II:103
branching structure V:372

calculus, proof IV:275
carrier
  non-empty III:128
  set I:7, III:128
Casl
  extension I:68
  institution with qualified symbols III:190
  institution with symbols III:191
  sublanguage I:61
  sublanguages I:62
Casl-Ltl extension I:68
cast, term I:32, III:187
category theory III:117
CATS V:364
character
  quoted token II:100
  representation V:370
  set II:97
character V:370
closed
  specification I:34, 36, 40, III:210
  subsignature III:197
  unit specification I:53, III:239
coalgebraic extension I:68
CoCasl extension I:68
cocone, weakly amalgamable IV:291, 349
cogenerated signature III:198
comment
  abbreviated II:103
  bracket II:103
  formatting II:104
  grouping II:103
  HTML II:105
  LaTeX II:105
  multi-line II:104
  non-nested II:103
  RTF II:105
  single-line II:104
comment II:103
commenting-out II:104
commutative monoid V:374
commutativity
  attribute I:12
commutativity III:143
comorphism
  condition IV:292, 349
  institution IV:291
compactness IV:285
compatibility
  of global environments III:232
  unit term I:55
  with extended static context III:250
compatible
  addition to unit context III:231
  additions III:250
  arguments I:50
  extension of unit context III:231
  extensions III:250
  fitting argument morphisms I:42
  fitting morphism III:214
  global environments III:266
  models III:229
  signature morphisms III:195
  static and model global environments III:204
  unit context III:231
  unit III:229
  universal environments III:268
complete
  lattice V:368
  partial order V:368
complete IV:283
completeness
  for the extended static semantics IV:338
  of calculus for basic specifications IV:284
  of extended static analysis III:262
  of rules for development graphs IV:308
completeness IV:277, 310, 353
completion function III:117
component
  constructor I:16
  sort I:16
  syntactic II:75
composition
  model I:49
  morphism III:117
  unit I:54, III:227
compound
  identifier translation I:47

identifier I:47, II:96, III:223
concept
architectural I:49
basic specification I:5
library I:57
structuring I:33
subsorting I:27
concrete syntax
relationship to abstract syntax II:73
concrete syntax II:73, 87
conditional
generalized positive logic I:66
positive logic I:65
term I:24, III:168
conflicting annotation II:105
congruence IV:280
conjunction I:20
connected graph V:373
connectedness V:375
connective, logical I:19, III:160
consequence I:6
conservative
extension annotation I:39, II:110
extension III:209, IV:296
conservativity IV:310
consistency IV:297, 310
consistent with $\varGamma$
model family IV:353
unit family IV:353
consistent I:6, III:249
constant
qualified I:24
constant I:6, III:124, 165
constraint
sort-generation I:8, 9
translation III:134
constraint I:6
CONSTRUCT... V:370
construction, pushout I:35
constructor
component I:16
partial I:15
syntactic II:75
total I:15
context
generic IV:334, 348
static IV:330
unit III:231
context-free
grammar meta-notation II:87
grammar II:75, 87
parsing II:88
context IV:330, 334, 348
cpo V:368
Craig interpolation property IV:289
CSP-CASL extension I:69
current signature I:34, III:204

data structure V:371
datatype
declaration I:15, 30, III:149
free declaration I:16, 30, 39, III:152, 178
generated declaration I:17
structured V:370
datatype I:14, 30, III:147
datatypes, basic V:363
date annotation II:111
decimal
fraction II:108, V:368
notation II:108, V:366
declaration
datatype I:15, 30, III:149
free datatype I:16, 30, 39, III:152, 178
generated datatype I:17
global variable I:17, III:158
isomorphism I:29, III:176
local variable I:18, III:158
operation I:11, III:142
partial selector I:16
predicate I:13, III:146
repeated I:10
signature I:10, 29, III:140
sort I:10, 11, III:141
subsort I:29, III:176
total selector I:16
unit I:51, 52, III:235, 253
variable scope I:18
variable I:18
decomposition, task I:49
deduction, natural IV:277
definedness
assertion I:8
formula I:22, III:164
definedness I:22
definition

architectural specification I:51, III:232, 251
generic specification I:41
library I:58
link IV:293
named specification I:41
operation I:13, III:144
partial operation I:13
predicate I:14, III:147
specification I:40, III:211, 266
subsort I:29, III:176
total operation I:13
unit specification I:52, III:237
unit I:51, 52, III:236, 254
view I:43, III:216
definitional extension
annotation I:39, II:111
definitional extension III:209, IV:296
degree function V:374
dependencies between units III:248
dependencies, diagram III:249
derivation
rule IV:280
derivation IV:281
derived grammar II:95
determinant V:375
development graph IV:275, 289, 293
diagram
extension III:248
of dependencies III:249
signature III:248
direct link I:59
directed graph V:372
directory, global I:57, III:267, 268
disambiguated I:46
disambiguation II:93
disjoint union I:30, III:117
disjointly extends III:248
disjunction I:20
display
annotation I:25, II:106
format II:97, 98
mixfix annotation II:106
distributed library I:58
divisibility V:370
division with remainder V:375
domain, semantic III:118
downloading I:58, 59, III:267
Eigenvariable condition IV:282
elided rule III:120
elimination oracle for free definition links IV:309
embedding
explicit I:32
implicit I:32
operation I:28
subsort I:27, 48, III:169
symbol III:172
embedding I:27
empty
signature III:193
specification I:9
encoding of the CASL logic IV:285
enrich I:36
enriched CASL IV:292
enrichment
subsorted III:175
enrichment III:135
ensures amalgamability III:250, IV:334
entailment system IV:285, 290
entity, syntactic kind II:75
enumeration type I:17
environment
global, compatible static and model III:204
global I:34, 51, 57, 58, III:204, 266
local I:6, 34, 58, III:138, 204
model global III:266
static global III:266
static universal III:267
unit III:230
universal model III:267
environment IV:330
environments
compatible global III:266
compatible universal III:268
equality feature I:65
equation
existential I:8, 22, III:164
strong I:8, 22, III:164
equation I:22
equivalence
formula I:31
relation V:368
term I:31
equivalence I:20, III:162

euclidian ring V:374
evaluator
  model IV:330
  unit III:230, IV:330
existential
  equation I:8, 22, III:164
  quantification I:19, III:160
expansion of formula, term I:21, 31, III:162
explicit
  embedding I:32
  qualification I:46
exponentiation II:108
expression, unit I:54, III:240, 255
expressiveness, levels I:65
Ext... V:370
extended
  signature III:200
  specification V:368
  static semantics III:247
  static unit context III:249
extends
  disjointly III:248
  model III:130
extension
  Casl I:68
  annotation II:110
  Casl-Ltl I:68
  coalgebraic I:68
  CoCasl I:68
  conservative annotation I:39, II:110
  conservative III:209
  Csp-Casl I:69
  definitional annotation I:39, II:111
  definitional III:209
  free I:30, 34
  HasCasl I:68
  HetCasl I:69
  higher-order I:68
  implicational III:209
  implied annotation I:39, II:111
  import III:230
  monomorphic annotation I:39, II:111
  monomorphic III:209
  of a diagram III:248
  of extended static unit context III:249
  reactive I:68
  refinement language I:69
  Sb-Casl I:69
  signature morphism III:199
  signature III:125, 194
  specification I:34, 36, III:208
  structured level I:69
  symbol map III:196
  symbol mapping I:35
extension I:36, 39
extensions
  compatible III:250
  free I:36

factorial ring V:374
false I:21
falsity III:163
feature
  equality I:65
  partiality I:64
  predicate I:64
  sort generation constraint I:65
  subsorting I:64
features, orthogonal language I:64
field V:369
final
  signature morphism III:193
  signature union III:194
  sink III:194
  union III:194
finite map III:116
first-order
  logic I:65
  many-sorted structure I:7
fitting
  argument list I:42
  argument morphism I:42
  argument specification I:42
  argument view I:45
  argument I:56, III:245
  compatible argument morphisms I:42
  morphism I:35
  view I:44, III:218
flat specification IV:291
flattenable IV:294
flattening I:36
floating-point number II:108
formal argument III:245
format, display II:97, 98

formatting a comment II:104
formula
  atomic I:8, 21, 31, III:184
  definedness I:22, III:164
  equivalence I:31
  expansion I:21, 31, III:162
  membership I:31, III:186
  satisfaction III:136
  translation III:134
  well-sorted I:21, 31, III:162
formula III:159
fraction, decimal II:108
fragment
  signature III:125
  union III:125
framework, basic specification I:5
free
  algebra V:374, 377
  datatype declaration I:16, 30, 39, III:152, 178
  extension I:30, 34
  extensions I:36
  monoid V:374
  specification I:36, 39, III:209
  variable III:133
full subsignature III:196
fully-qualified
  symbol III:195
  term I:8, III:131
function
  completion III:117
  generic unit I:52
  graph III:116
  partial, symbol I:6, III:124
  partial I:7, III:116, 128
  persistent III:229
  profile I:6, III:124
  qualified symbol III:126
  total, symbol I:6, III:124
  total III:116, 128
  unit type I:50
  unit I:49, III:227

generalized positive conditional logic I:66
generated
  datatype declaration I:17
  signature III:197
  sort III:137
generated I:9
generating signature morphism III:193
generation, sort I:17, III:157, 183
generative
  model semantics IV:338
  semantics III:262
generic
  context IV:334, 348
  signature III:202
  specification definition I:41
  specification import I:35
  specification instantiation I:34, III:214
  specification I:40, III:202, 212
  unit function I:52
  unit signature IV:330
  unit specification IV:330
  unit III:229
  view instantiation I:45
generic I:34
Gentzen-style IV:277, 282
global
  annotation II:105
  compatible environments III:266
  directory I:57, III:267, 268
  environment, verification IV:316, 358
  environment I:34, 51, 57, 58, III:204, 266
  model environment III:266
  static environment III:266
  variable declaration I:17, III:158
globally reachable IV:293
grammar
  abbreviated II:81
  context-free II:75, 87
  derived II:95
  normal II:76
graph
  homomorphism V:373
  of a function III:116
  properties V:373
graph V:372
group
  action V:374
  symmetric V:376
group V:369
grouping
  annotation II:103

braces I:36
comment II:103
mixfix analysis II:88, 93, 95
guaranteed property I:49

HasCasl extension I:68
HetCasl extension I:69
Hets V:364
hiding
reduction I:37
signature I:34
unit parts I:55
hiding I:35, 40, III:206
higher-order extension I:68
homomorphism
many-sorted I:7, III:129
homomorphism I:5
HTML
annotation II:106
comment II:105

idempotency
attribute I:12
idempotency III:143
identifier
compound translation I:47
compound I:47, II:96, III:223
mixfix I:25, II:96
qualified I:46
unqualified I:23
identifier I:23, 25, III:165, 168
identity
map I:47
translation I:37
implication
reverse I:20
implication I:20, III:161
implicational extension III:209
implicit
associativity annotation II:108
embedding I:32
implied
axiom annotation II:110
extension annotation I:39, II:111
import
extension III:230
generic specification I:35
specification I:41
unit III:235, 254
view I:43
imported unit I:52
inclusion
signature III:127, 194
subsort III:169
incomplete IV:308
incompleteness
of rules for development graphs IV:308
incompleteness IV:277, 285
inconsistency, accidental III:181
inconsistent
generic unit specification IV:330
specification IV:330
inconsistent I:6
independence, institution III:190
indirect link I:60
induction rule IV:280
infer I:5
inference, preserve I:6
infix precedence I:24
initial model I:34, 40
input syntax II:97
instantiation
generic specification I:34, III:214
generic view I:45
pushout III:200
specification I:42
subsort preservation I:48
instantiation I:42, 47
institution
Casl, with qualified symbols III:190
Casl, with symbols III:191
comorphism IV:291
independence versus proof calculus IV:290
independence III:190, IV:276, 290
independent semantics III:120
institution I:5, III:123
integer V:366
integral domain V:369
intended consequences IV:276
interface I:49
Internet I:57
irreducible element V:370, 375
ISO Latin-1 II:97
isomorphism
declaration I:29, III:176

isomorphism III:131
item, basic I:9

kernel
  language of structured specifications IV:289
  language IV:289
kernel III:116
key word, sign II:97, 98
kind of syntactic entity II:75

label annotation II:106
LALR(1) II:88
language
  for naming sublanguages I:61
  kernel IV:289
  orthogonal features I:64
  refinement extension I:69
  regular II:97
largest subsignature I:35
LaTeX
  annotation II:106
  comment II:105
layout II:97
Leibniz formula V:376
levels of expressiveness I:65
lexical
  analysis II:88, 97
  symbol II:88, 97
lexicographical order I:59
library
  concept I:57
  definition I:58
  distributed I:58
  local I:58
  name I:59, III:267
  primary location I:59
  specification I:57
  version I:59
library III:266
linear
  algebra V:375, 377
  combination V:376
  visibility I:6, 10, 51, 57, 58, III:139
link
  definition IV:293
  direct I:59
  indirect I:60
  theorem IV:295

list
  annotation II:109
  fitting argument I:42
  of symbols I:46
  symbol map I:46
  symbol III:221
list V:371
literal syntax annotation II:108
LL(1) II:88
local
  associativity annotation II:108
  assumption IV:280
  axiom IV:293
  environment I:6, 34, 58, III:138, 204
  implication IV:295
  library I:58
  specification I:36, 40, III:210
  unit I:56, III:244, 259
  variable declaration I:18, III:158
locally reachable IV:294
location, primary library I:59
logic
  atomic I:67
  first-order I:65
  generalized positive conditional I:66
  positive conditional I:65
logic IV:276, 290
logical connective I:19, III:160

machine number V:377
many-sorted
  basic specification I:9, III:138
  first-order structure I:7
  homomorphism I:7, III:129
  model I:7, III:128
  partial algebra I:7
  reduct I:7, III:130
  sentence I:8, III:132
  signature morphism I:7, III:126
  signature I:6, III:124
  term I:7, III:131
map
  finite III:116
  identity I:47
  signature symbol III:195
  symbol, induced by a signature morphism III:195
  symbol III:195, 222
map V:371

mapping
  symbol extension I:35
  symbol I:35, 46
matching
  of symbol maps III:195
  of symbols III:192
mathematical sign II:98
matrix
  multiplication V:375
matrix V:375
'maybe'-type V:371
membership
  formula I:31, III:186
  predicate I:28
  symbol III:172
membership I:31
meta-notation, context-free grammar II:87
minor of a graph V:374
mixfix
  display annotation II:106
  grouping analysis II:88, 93, 95
  identifier I:25, II:96
  notation I:25
  token I:25
model
  architectural specification I:50
  architectural III:230
  class semantics III:189
  compatible III:229
  composition I:49
  evaluator IV:330
  extends III:130
  family consistent with Γ IV:334, 353
  global environment III:266
  initial I:34, 40
  many-sorted I:7, III:128
  reduct III:130
  semantics, generative IV:338
  semantics III:119, 138
  subsorted I:28, III:173
  universal environment III:267
model I:5, 7, 28
monoid
  action V:374, 375
  commutative V:374
  free V:374
monoid V:369
monomorphic
  extension annotation I:39, II:111
  extension III:209, IV:296
morphism
  composition III:117
  final signature III:193
  fitting argument I:42
  fitting I:35
  many-sorted signature I:7, III:126
  signature I:6, 35, III:196
  specification I:35, III:202
  subsorted signature I:28
morphism I:5
morphisms
  compatible fitting argument I:42
multi-line comment II:104
multiplication, matrix V:375

name
  architectural specification I:51
  library I:59, III:267
  of signature symbol III:192
  qualified operation I:24
  qualified predicate I:22
  specification I:41
  unqualified predicate I:22
  view I:44
named
  specification definition I:41
  specification I:34, 40, III:212, 266
  view I:35, 43
natural deduction IV:277
negation I:21, III:162
negative premise III:120
'no confusion' III:155
'no junk' III:155
non-empty carrier III:128
non-generic unit III:229
non-linear visibility I:6, 14, III:147
non-nested
  annotation II:103
  comment II:103
nonterminal symbol II:87
normal grammar II:76
notation
  decimal II:108
  mixfix I:25
number
  annotation II:108

  floating-point II:108
  symbol II:100
  version III:267
number V:365

OBJ3 IV:287
objects of a category III:117
obligation, proof IV:275
observer V:371
omission, parentheses II:106
operation
  application I:23, III:166
  attribute I:12, III:143
  declaration I:11, III:142
  definition I:13, III:144
  embedding I:28
  partial, definition I:13
  partial, type I:12
  profile I:12, III:142
  projection I:28
  qualified name I:24
  total, definition I:13
  total, type I:11
  type I:11
operation I:7, 11, III:124
optional symbol II:87
oracle
  for conservative extension IV:309
  for free theorem links IV:309
orbit V:375
order
  lexicographical I:59
order-sorted approach I:27
order V:368
origin, symbol I:54
orthogonal language features I:64
overloaded symbol I:7, 46, III:126
overloading relation I:27, III:170

pair V:371
parameter
  specification I:34, 41
  view I:43
parentheses, omission II:106
parse tree II:88
parsing
  annotation II:106
  context-free II:88
partial
  constructor I:15
  function symbol I:6, III:124
  function I:7, III:116, 128
  many-sorted algebra I:7
  model semantics IV:339
  operation definition I:13
  operation type I:12
  order V:368
  selector declaration I:16
partiality feature I:64
path
  in a graph V:373
path II:101, III:267
permutation V:376
persistent
  function III:229
persistent I:50, IV:330
place-holder I:25
planar graph V:374
polynomial V:374, 377
positive
  conditional logic I:65
  generalized conditional logic I:66
  integer V:366
postfix precedence I:24
pre-development graph IV:312
precedence
  annotation II:107
  architectural specification II:94
  conditional II:95
  formula II:94
  function application II:94
  infix I:24, II:95
  mixfix I:24
  postfix I:24, II:94
  prefix I:24, II:95
  rule I:36
  structured specification II:94
  term II:94
preceding annotation II:104
predicate
  application I:8, 21, III:163
  declaration I:13, III:146
  definition I:14, III:147
  feature I:64
  membership I:28
  profile I:6, 14, III:124
  qualified name I:22
  qualified symbol III:126

symbol I:6, III:124
type I:14, III:147
unqualified name I:22
predicate I:7, 13, III:128
prefix precedence I:24
presentation I:6, IV:291
preservation
subsort by instantiation I:48
preserve
inference I:6
satisfaction I:6
primary library location I:59
principle, 'same name, same thing' I:36, 38
product, scalar V:375
production rule II:75, 87
profile
function I:6, III:124
operation I:12, III:142
predicate I:6, 14, III:124
projection
operation I:28
subsort I:32
symbol III:172
proof
calculus IV:275, 289, 347
obligation IV:275, 276, 293, 295
rule IV:280
rules for development graphs IV:298
system I:5
property
assumed I:49
guaranteed I:49
pushout
construction I:35
for instantiation III:200
selected IV:330

qualification, explicit I:46
qualified
constant I:24
function symbol III:126
identifier I:46
operation name I:24
predicate name I:22
predicate symbol III:126
symbol I:7
symbols, Casl institution with III:190
variable I:23, III:166
quantification
existential I:19, III:160
unique-existential I:19, III:160
universal I:19, III:160
quantification I:19
quoted
character token II:100
string symbol II:100

rational number V:367
reactive extension I:68
real number V:366
reduct
many-sorted I:7, III:130
model III:130
subsorted model III:174
reduct I:6, 35, III:130
reduction
hiding I:37
revealing I:37
specification I:36
unit I:55, III:243, 258
reduction I:37, III:206
reference
architectural specification I:51
specification I:34, III:266
reference I:57
refinement language extension I:69
reflexive relation V:368
regular language II:97
regularity condition I:27
relation
overloading I:27, III:170
satisfaction III:123
relation V:368
remainder V:375
renaming unit parts I:55
repeated declaration I:10
repetition
symbol II:87
syntactic II:75
requirement specification I:49
reserved word, sign II:98
restriction, unit I:55
result sort I:6, III:124
reuse, specification I:57
revealing
reduction I:37

revealing III:206
reverse implication I:20
rich specification V:368
ring
  euclidian V:374
  factorial V:374
ring V:369
RTF
  annotation II:106
  comment II:105
rule
  derivation IV:280
  elided III:120
  precedence I:36
  production II:75, 87
  proof IV:280
  semantic III:119

'same name, same thing'
  principle I:36, 38
satisfaction
  of a formula III:136
  of a sentence III:137
  of a sort-generation constraint
    III:137
  preserve I:6
  relation III:123
  subsorted III:175
  two-valued I:8
  unit type I:53
satisfaction I:5, 8, III:135
Sb-Casl extension I:69
scalar product V:375
scope, variable declaration I:18
second-order logic IV:285
selected pushout IV:330
selector
  partial declaration I:16
  total declaration I:16
self-contained specification I:36, 58
semantic
  annotation II:110, III:208, IV:296
  domain III:118
  rules III:119
  sharing III:242
semantically follows IV:283
semantics
  applicative III:262, IV:338
  basic specification I:6
  extended static III:247
  generative III:262
  institution-independent III:120
  model class III:189
  model III:119, 138
  partial model IV:339
  static I:54, III:118, 138
  structured specification I:34
semantics I:6, III:115
sentence
  many-sorted I:8, III:132
  satisfaction III:137
  subsorted, translation III:174
  subsorted I:28, III:174
sentence I:5, 7, 28
sequence
  alternative II:87
  symbol II:87
sequence III:116
set
  carrier I:7, III:128
  character II:97
  symbol I:35
set III:116, V:371
shared symbol I:43
sharing
  between symbols I:54
  semantic III:242
shortest path V:373
sign
  key II:97, 98
  mathematical II:98
  reserved II:98
  token II:99
  unreserved II:98
sign V:376
signature
  architectural III:228
  cogenerated III:198
  current I:34, III:204
  declaration I:10, 29, III:140
  diagram III:248
  empty III:193
  extended III:200
  extension III:125, 194
  final morphism III:193
  fragment union III:125
  fragment, subsorted III:170
  fragment III:125

generated III:197
generic III:202
hiding I:34
inclusion, subsorted III:171
inclusion III:127, 194
many-sorted morphism I:7
many-sorted, morphism III:126
many-sorted I:6, III:124
morphism induced by a symbol map III:199
morphism, extension III:199
morphism, generating III:193
morphism, subsorted III:171
morphism, transportable IV:304
morphism I:6, 35, III:196
morphisms leaving names unchanged III:195
morphisms, compatible III:195
morphisms, union III:196
subsorted fragment union III:170
subsorted morphism I:28
subsorted I:27, III:169
symbol map III:195
symbol III:191
symbols III:126
translation I:34
union, final III:194
union III:193
unit III:228
signature I:5, 6, 27
simple datatype V:370
single-line
annotation II:105
comment II:104
sink
final III:194
sink III:194, 250
site I:57, III:267
smallest subsignature I:35
sort
argument I:6
component I:16
declaration I:10, 11, III:141
generation constraint feature I:65
generation I:17, III:137, 157, 183
result I:6, III:124
sort-generation constraint
satisfaction III:137
sort-generation constraint I:8, 9, III:134
sort I:6, 10, 29
sorted term I:24, III:167
sorts
argument III:124
sorts III:124
sound IV:283
soundness
for the extended static semantics IV:338
of extended static semantics III:262
of rules for development graphs IV:308
soundness IV:277, 353
space II:97
spanning tree V:373
specialize I:36
specification
architectural definition I:51, III:232, 251
architectural model I:50
architectural name I:51
architectural reference I:51
architectural unit I:53, III:239
architectural I:49, 50, III:227, 232
argument I:34
basic architectural I:51, III:233, 252
basic concept I:5
basic framework I:5
basic many-sorted I:9
basic semantics I:6
basic subsorted I:27
basic I:5, III:123
body I:34, 41
closed unit I:53, III:239
closed I:34, 36, 40, III:210
definition I:40, III:211, 266
empty I:9
extension I:34, 36, III:208
fitting argument I:42
free I:36, 39, III:209
generic definition I:41
generic import I:35
generic instantiation I:34, III:214
generic I:40, III:202
import I:41
in comment II:105
inconsistent IV:330

  instantiation I:42
  library I:57
  local I:36, 40, III:210
  many-sorted basic III:138
  methodology V:363
  morphism I:35, III:202
  name I:41
  named definition I:41
  named I:34, 40, III:266
  parameter I:34, 41
  reduction I:36
  reference I:34, III:266
  requirement I:49
  reuse I:57
  self-contained I:36, 58
  structured semantics I:34
  structured I:33, 36, III:204
  structuring I:33
  subsorted basic III:175
  subsorted III:169
  subsorting I:27
  translation I:36, 37
  union I:34, 36, 38, III:207
  unit definition I:52, III:237
  unit I:52, III:230, 237, 255
start, no symbol II:87
static
  analysis tool III:247
  context IV:330
  extended semantics III:247
  global environment III:266
  semantics of architectural specifications IV:330, 333
  semantics I:54, III:118, 138
  unit context III:228
  universal environment III:267
string
  annotation II:109
  quoted symbol II:100
string V:371
strong equation I:8, 22, III:164
structure, many-sorted first-order I:7
structured
  datatype V:370
  specification semantics I:34
  specification I:33, 36, III:204
structuring
  concept I:33
  specification I:33
subcontext IV:334
subdiagram III:248
sublanguage
  naming language I:61
  of CASL I:61, 62
subsignature
  closed III:197
  full III:196
  largest I:35
  smallest I:35
  subsorted III:170
subsignature III:126, 194
subsort
  alternative I:30
  declaration I:29, III:176
  definition I:29, III:176
  embedding I:27, 48, III:169
  inclusion III:169
  preservation by instantiation I:48
  projection I:32
subsort I:27
subsorted
  basic specification I:27, III:175
  enrichment III:175
  model reduct III:174
  model I:28, III:173
  satisfaction III:175
  sentence I:28, III:174
  signature extension III:170
  signature fragment III:170
  signature inclusion III:171
  signature morphism I:28, III:171
  signature I:27, III:169
  specification III:169
  subsignature III:170
  translation of sentence III:174
  union of signature fragments III:170
subsorting
  concept I:27
  feature I:64
  specification I:27
substitution
  lemma IV:283
substitution IV:279
symbol
  auxiliary I:36
  embedding III:172
  fully-qualified III:195
  lexical II:88, 97

list I:46, III:221
map induced by a signature morphism III:195
map list I:46
map, extension III:196
map, matching III:195
map III:195, 222
mapping extension I:35
mapping I:35, 46
matching III:192
membership III:172
name III:192
no start II:87
nonterminal II:87
number II:100
optional II:87
origin I:54
overloaded I:7, 46, III:126
partial function I:6, III:124
predicate I:6, III:124
projection III:172
qualified function III:126
qualified predicate III:126
qualified I:7
quoted string II:100
repetition II:87
sequence II:87
set I:35
shared I:43
signature III:126, 191
terminal II:87
total function I:6, III:124
symbol I:5, III:191
symbols
CASL institution with III:191
sharing between I:54
symmetric
graph V:373
group V:376
relation V:368
syntactic
component II:75
constructor II:75
entity kind II:75
repetition II:75
syntax
abstract II:73, 75
concrete II:73, 87
input II:97
literal annotation II:108
system, proof I:5

task decomposition I:49
term
cast I:32, III:187
conditional I:24, III:168
equivalence I:31
expansion I:21, 31, III:162
fully-qualified I:8, III:131
many-sorted I:7, III:131
sorted I:24, III:167
unit compatibility I:55
unit I:54, III:242, 257
well-sorted I:31, III:162
term I:23, 32, III:165, 187
terminal symbol II:87
theorem link IV:295
theory
category III:117
morphism IV:291
theory IV:291
token
mixfix I:25
quoted character II:100
sign II:99
word II:99
token I:25, II:99
total
constructor I:15
function symbol I:6, III:124
function III:116, 128
operation definition I:13
operation type I:11
order V:368
selector declaration I:16
trailing annotation II:104
transitive
closure V:373
relation V:368
translating development graphs along institution comorphisms IV:297
translation
compound identifier I:47
from structured specification to development graph IV:311
identity I:37
of *SP* along σ IV:348
of a constraint III:134

  of formula III:134
  signature I:34
  specification I:36, 37
  subsorted sentence III:174
  unit I:55, III:242, 257
translation I:6, 35, 37, III:205
transportable signature morphism IV:304
tree
  parse II:88
tree V:371, 373
true I:21
truth I:21, III:163
tuple III:116
two complement V:377
two-valued satisfaction I:8
type
  enumeration I:17
  operation I:11
  partial operation I:12
  predicate I:14, III:147
  total operation I:11
  unit function I:50
  unit satisfaction I:53
  unit I:53, III:238

undefined value I:8
undirected graph V:373
union
  disjoint I:30, III:117
  final III:194
  of signature fragments III:125
  of subsorted signature fragments III:170
  signature morphisms III:196
  signature, final III:194
  signature III:193
  specification I:34, 36, 38, III:207
  subsorted signature fragments III:170
unique factorization V:375
unique-existential quantification I:19, III:160
unit
  amalgamation I:55, III:244
  application I:56, III:245, 260
  architectural specification I:53
  argument III:245
  attribute I:12
  closed specification I:53
  compatible III:229
  composition I:54, III:227
  context III:231
  declaration I:51, 52, III:235, 253
  definition I:51, 52, III:236, 254
  dependencies III:248
  element V:370
  environment III:230
  evaluator III:230, IV:330
  expression I:54, III:240, 255
  family consistent with Γ IV:353
  function type I:50
  function I:49, III:227
  generic function I:52
  generic III:229
  hiding parts I:55
  import I:52, III:235, 254
  local I:56, III:244, 259
  non-generic III:229
  reduction I:55, III:243, 258
  renaming parts I:55
  restriction I:55
  signature III:228, IV:330
  specification definition I:52, III:237
  specification, architectural III:239
  specification, closed III:239
  specification, generic IV:330
  specification, inconsistent generic IV:330
  specification I:52, III:230, 237, 255
  term compatibility I:55
  term I:54, III:242, 257
  translation I:55, III:242, 257
  type satisfaction I:53
  type I:53, III:238
unit III:143
universal
  compatible environments III:268
  model environment III:267
  quantification I:19, III:160
  static environment III:267
unqualified
  identifier I:23
  predicate name I:22
unreserved
  sign II:98
  word II:98
URL I:59, II:101, III:267

valid object III:118
value
  of a term III:135, 136
  undefined I:8
variable
  capture problem IV:280
  declaration scope I:18
  declaration I:18
  free III:133
  global declaration I:17, III:158
  local declaration I:18, III:158
  qualified I:23, III:166
variable I:17
vector
  space V:375, 377
vector V:375
verification
  global environment IV:316, 358
  semantics IV:276, 289, 312
version
  library I:59
  number I:59, III:267
view
  definition I:43, III:216, 217
  fitting argument I:45
  fitting I:44, III:218
  generic instantiation I:45
  import I:43
  name I:44
  named I:35, 43
  parameter I:43
view I:43, III:203
visibility
  linear I:6, 10, 51, 57, 58, III:139
  non-linear I:6, 14, III:147

weakly amalgamable cocone IV:289, 291, 349
well-formedness I:9
well-sorted formula, term I:21, 31, III:162
word
  key II:97, 98
  reserved II:98
  token II:99
  unreserved II:98

zero divisor V:370