Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Hubert Comon   Claude Marché
Ralf Treinen (Eds.)

Constraints in Computational Logics

Theory and Applications

International Summer School, CCL'99
Gif-sur-Yvette, France, September 5-8, 1999
Revised Lectures

Springer

Volume Editors

Hubert Comon
École Normale Supérieure de Cachan, Laboratoire Spécification et Vérification
61, avenue du Président Wilson, 94234 Cachan Cedex, France
E-mail: Hubert.Comon@lsv.ens-cachan.fr

Claude Marché
Ralf Treinen
Université Paris-Sud, Laboratoire de Recherche en Informatique
Bâtiment 490, 91405 Orsay Cedex, France
E-mail: {marche/treinen}@lri.fr

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Constraints in computational logics : theory and applications ;
international summer school ; revised lectures / CCL ’99.
Gif-sur-Yvette, France, September 5 - 8, 1999. Hubert Comon ... (ed.).
- Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ;
Milan ; Paris ; Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2002)
ISBN 3-540-41950-0

CR Subject Classification (1998): F.4.1, I.2.3, F.3.1, D.1.6, G.1.6, D.3.3

ISSN 0302-9743
ISBN 3-540-41950-0 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10782264 06/3142 5 4 3 2 1 0

Preface

CCL (*Construction of Computational Logics* and later *Constraints in Computational Logic*) is the name of an ESPRIT working group which met regularly from 1992 to 1999 (see `http://www.ps.uni-sb.de/ccl/`). It united research teams from Germany, France, Spain, and Israel, and was managed by the company COSYTEC.

In its final few years, the main emphasis of the working group was on *constraints* — techniques to solve them and combine them and applications ranging from industrial applications to logic programming and automated deduction. At the end of the working group, in fall 1999, we organized a summer school, intending to summarize the main advances achieved in the field during the previous 7 years. The present book contains the (revised) lecture notes of this school. It contains six chapters, each of which was written by some member(s) of the working group, covering the various aspects of constraints in computational logic. We intend it to be read by non specialists, though a prior knowledge in first-order logic and programming is probably necessary.

Constraints provide a declarative way of representing infinite sets of data. As we (attempt to) demonstrate in this book, they are well suited for the *combination* of different logical or programming paradigms. This is known since the 1980s for *constraint logic programming*, but has been combined with functional programming in more recent years; a chapter (by M. Rodríguez-Artalejo) is devoted to the combination of constraints, logic, and functional programming.

The use of constraints in automated deduction is more recent and has turned out to be very successful, moving the control from the meta-level to the constraints, which are now first-class objects. This allows us to keep a history of the reasons why deductions were possible, hence restricting further deductions. A chapter of this book (by H. Ganzinger and R. Nieuwenhuis) is devoted to constraints and theorem proving.

Constraints are not only a nice mathematical construction. The chapter (by H. Simonis) on industrial applications shows the important recent developments of constraint solving in real life applications, for instance scheduling, decision making, and optimization.

Combining constraints (or combining decision procedures) has emerged during the last few years as an important issue in theorem proving and verification. Constraints turn out to be an adequate formalism for combining efficient techniques on each particular domain, thus yielding algorithms for mixed domains. There is now a biannual workshop on these topics, of which the proceedings are published in the LNAI series. The chapter on Combining Constraints Solving (by F. Baader and K. Schulz) introduces the subject and surveys the results.

Before these four chapters on applications of constraint solving, the introductory chapter (by J.-P. Jouannaud and R. Treinen) provides a general introduction to constraint solving. The chapter on constraint solving on terms (by H. Comon and C. Kirchner) introduces the constraint solving techniques which are used in, e.g. applications to automated deduction.

Every chapter includes an important bibliography, to which the reader is referred for more information.

We wish to thank the reviewers of these notes, who helped us improve the quality of this volume. We also thank the European Union who supported this work for 6 years and made possible the meeting in Gif.

January 2001

Hubert Comon
Claude Marché
Ralf Treinen

Full Addresses of Contributors and Editors

Franz Baader
Rheinisch-Westfälische Technische Hochschule Aachen
Lehr- und Forschungsgebiet Theoretische Informatik
Ahornstrasse 55
D-52074 Aachen
Germany
E-mail: `baader@informatik.rwth-aachen.de`
Web: `http://www-lti.informatik.rwth-aachen.de/ti/baader-en.html`

Hubert Comon
Laboratoire Spécification et Vérification
École Normale Supérieure de Cachan
61, avenue du Président Wilson
F-94234 Cachan Cedex
France
E-mail: `Hubert.Comon@lsv.ens-cachan.fr`
Web: `www.lsv.ens-cachan.fr/~comon`

Harald Ganzinger
Max-Planck-Institut für Informatik
Programming Logics Group, Room 601
Im Stadtwald
D-66123 Saarbrücken
Germany
E-mail: `hg@mpi-sb.mpg.de`
Web: `http://www.mpi-sb.mpg.de/~hg/`

Jean-Pierre Jouannaud
Laboratoire de Recherche en Informatique
Bâtiment 490
Université Paris-Sud
F-91405 Orsay Cedex
France
E-mail: `jouannau@lri.fr`
Web: `http://www.lri.fr/demons/jouannau`

Claude Kirchner
LORIA & INRIA
615, rue du Jardin Botanique
BP-101
F-54602 Villers-lès-Nancy
France
E-mail: `Claude.Kirchner@loria.fr`
Web: `http://www.loria.fr/~ckirchne/`

Claude Marché
Laboratoire de Recherche en Informatique
Bâtiment 490
Université Paris-Sud
F-91405 Orsay Cedex
France
E-mail: `marche@lri.fr`
Web: `http://www.lri.fr/demons/marche`

Robert Nieuwenhuis
Technical University of Catalonia (UPC)
Department of Software (LSI), Building C6, Room 112
Jordi Girona 1
E-08034 Barcelona
Spain
E-mail: `roberto@lsi.upc.es`
Web: `http://www-lsi.upc.es/~roberto/home.html`

Mario Rodríguez-Artalejo
Universidad Complutense de Madrid
Dpto. de Sistemas Informáticos y Programación
Edificio Fac. Matemáticas
Av. Complutense s/n
E-28040 Madrid
Spain
E-mail: `mario@eucmos.sim.ucm.es`

Klaus U. Schulz
Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilians-Universität München
Oettingenstrasse 67
D-80538 München
Germany
E-mail: `schulz@cis.uni-muenchen.de`
Web: `http://www.cis.uni-muenchen.de/people/schulz.html`

Helmut Simonis
COSYTEC SA
4, rue Jean Rostand
F-91893 Orsay Cedex
France

Current Address:
Parc Technologies Ltd
2nd Floor, The Tower Building
11 York Road
London SE1 7NX
United Kingdom
E-mail: `Helmut.Simonis@parc-technologies.com`

Ralf Treinen
Laboratoire de Recherche en Informatique
Bâtiment 490
Université Paris-Sud
F-91405 Orsay Cedex
France
E-mail: `treinen@lri.fr`
Web: `http://www.lri.fr/demons/treinen`

Contents

1 Constraints and Constraint Solving: An Introduction

Jean-Pierre Jouannaud[1] and Ralf Treinen[1]*

Laboratoire de Recherche en Informatique, Université Paris-Sud, Orsay, France

1.1 Introduction

The central idea of constraints is to *compute with descriptions of data* instead of to compute with data items. Generally speaking, a constraint-based computation mechanism (in a broad sense, this includes for instance deduction calculi and grammar formalisms) can be seen as a two-tired architecture, consisting of

- A language of data descriptions, and means to compute with data descriptions. Predicate logic is, for reasons that will be explained in this lecture, the natural choice as a framework for expressing data descriptions, that is *constraints*.
- A computation formalism which operates on constraints by a well-defined set of operations. The choice of this set of operations typically is a compromise between what is desirable for the computation mechanism, and what is feasible for the constraint system.

In this introductory lecture we will try to explain basic concepts of constraint-based formalisms in an informal manner. In-depth treatment is delegated to the lectures dedicated to particular subjects and to the literature.

This lecture is organised as follows: We start with a small tour of constraint-based formalisms comprising constraint logic programming, constraints in automated deduction, constraint satisfaction problems, constrained grammars in computational linguistics, constraint-based program analysis, and constraints in model checking. In this first section, we will employ a somewhat naive view of constraint systems since we will assume that we have complete and efficient means to manipulate descriptions of data (that is, constraints). This allows to cleanly separate calculus and constraints, which can be considered as an ideal situation.

In the following sections we will see that, in reality, things are not that simple. In Section 1.3 we will investigate a particular family of constraint systems, so-called feature constraints, in some depth. It will turn out that complete procedures to compute with constraints may be too costly and that we may have to refine our calculus to accommodate for incomplete constraint

* Both authors supported by the ESPRIT working group CCL-II, ref. WG # 22457.

handling. Section 1.4 will present such a refinement for constraint logic programming. In Section 1.5, finally, we will address the question of how the basic algorithms to compute with constraints can be implemented in the same programming calculus than the one that uses these algorithms.

1.2 A First Approach to Constraint Based Calculi

1.2.1 First-Order Logic as a Language

The role of this section is to give the basics of first-order logic viewed as a language for expressing relations over *symbolic data items*. It can be skipped by those readers aware of these elementary notions.

A *term* is built from *variables*, denoted by upper-case letters X, Y, Z, etc. (following the tradition of logic programming) or by lower-case letters x, y, z, etc. (following the tradition of mathematical logic), and *function symbols* of a given *arity*, usually denoted by words of lower-case letters as f, a, *plus*. We assume a sufficient supply of function symbols of any arity. Examples of terms are a, $g(a, X)$, $f(g(X, Y), X)$ and $f(g(a, b), a)$. The number of arguments of a function symbol in a term must be equal to its given arity. Terms without variables like a or $f(g(a, b), a)$ are called *ground*, they are the data items of the logic language. The set of function symbols is called the *signature* of the language of terms.

A *substitution* is a mapping from variables to terms. Substitutions are usually denoted by Greek letters like σ, τ. Since it is usually sufficient to consider mappings that move only a finite number of variables, substitutions can be written as in $\sigma = \{X \mapsto a, Y \mapsto b\}$ and $\tau = \{X \mapsto Z\}$, where it is understood that all variables map to themselves unless mentioned otherwise. The application of a substitution to a term is written in postfix notation, and yields another term, for instance $f(g(X, Y), X)\sigma = f(g(a, b), a)$ and $f(X, Y)\tau = f(Z, Y)$. The application of a substitution to a term t yields an *instance* of t (hence a *ground instance* if the term obtained is ground). One of the basic ideas of logic is that a term denotes the set of its ground instances, hence is a description of a set of data items.

Different terms may have common ground instances, that is, the two sets of data items that they describe may have a non-empty intersection. When this is the case, the two terms are said to be *unifiable*. For an example, $f(X, g(Y))$ and $f(g(Z), Z)$ are unifiable, since they have $f(g(g(a)), g(a))$ as a common ground instance. The substitution $\tau = \{X \mapsto g(g(a)), Y \mapsto a, Z \mapsto g(a)\}$ is called a *unifier* of these two terms. Does there exist a term whose set of ground instances is the intersection of the sets of ground instances of two unifiable terms s and t? The answer is affirmative, and the *most general instance* of s and t is obtained by instantiating any of them by their *most general unifier* or *mgu*. In the above example, the mgu is $\sigma = \{X \mapsto g(g(Y)), Z \mapsto g(Y)\}$. It should be clear that any unifier τ is an instance of the mgu, by the substitution $\tau' = \{Y \mapsto a\}$ in our example. This is captured by writing the composition of substitutions in diagrammatic order: $\tau = \sigma\tau'$. Renaming the variables of the most general instance does not change its set of ground instances, we say that the most general instance, and the most general unifier as well, are defined up to *renaming of variables*. It should be clear that we can unify n terms $t_1, \ldots, t_n$ as well. In practice, we often need to unify sequences $\overline{s}$ and $\overline{t}$ of terms, that is, find a most general unifier σ such that $s_1\sigma = t_1\sigma, \ldots, s_n\sigma = t_n\sigma$.

So far, we have considered finite terms only. In practice, as we will see later, it is often convenient or necessary to consider infinite terms as well. Examples of infinite terms are

$$g(a, g(a, g(a, \ldots g(a, \ldots))))$$
$$g(a, g(h(a), g(h(h(a)), \ldots g(h(\ldots(h(a)), \ldots)))))$$

The above discussion for finite terms remains valid for infinite *rational terms*, that is, infinite terms having a finite number of subterms. In our example of infinite terms, the first is regular while the second is not. An alternative characterisation of regular terms is that they can be recognised by finite tree automata. Regular terms can therefore be described finitely, in many ways, in particular by graphs whose nodes are labelled by the symbols of the signature. Apart from its complexity, unification of infinite regular terms enjoys similar properties as unification of finite terms.

An *atom* is built from terms and *predicate symbols* of a given arity, usually also denoted by words of lower-case letters. There is no risk of confusion with function symbols, since terms and atoms are of two different kinds. We assume a sufficient supply of predicate symbols of any arity. Examples are *connect(X,Y,join(a,b))* or *append(nil,cons(a,nil),cons(a,nil))*, the latter is called *closed* since it contains no variable.

Formulae are built from atoms by using the binary logical connectives $\wedge$, $\vee$ and $\Rightarrow$, the unary connective $\neg$, the logical constants $True$ and $False$, the existential quantifier $\exists$ and the universal quantifier $\forall$. Several consecutive universal or existential quantifications are grouped together. An example of a formula is $\forall XY \exists Z\ plus(X, Z, succ(Y))$. *Clauses* are particular formulae of the form $\forall \overline{X}\ B_1 \vee \ldots \vee B_n \vee \neg A_1 \ldots \vee \neg A_m$, where $m \geq 0, n \geq 0$, and $\overline{X}$ denotes the sequence of variables occurring in the formula. The atoms B_i are said to be *positive*, and the A_j *negative*. The universal quantification $\forall \overline{X}$ will usually be implicit, and parentheses are often omitted. The clause may equivalently be written $(A_1 \wedge \ldots \wedge A_m) \Rightarrow (B_1 \vee \ldots \vee B_n)$, using conjunctions and implication, or simply $B_1, \ldots, B_m \leftarrow A_1, \ldots, B_n$, a notation inherited from Logic Programming. Clauses with at most one positive atom are called *Horn clauses*, the positive atom is called its *head*, and the set of negative ones form its *body*. A Horn clause without body is called a *fact*, and one without head a *query*.

To know more: More on unification of terms and automata techniques is to be found in the chapter *Constraint Solving on Terms*. See also [JK91] for a modern approach to unification and [CDG+99] on automata techniques. The basic logic notations can be found in many text book covering logical methods, for instance [BN98].

1.2.2 From Logic Programming to Constrained Logic Programming

Historically, the need of constraints arose from *Logic Programming* (LP), one of the classical formalisms following the paradigm of *declarative programming.* Let us recall the basics of logic programming before we show how constraints make LP an even more natural and efficient declarative language. A *program* is a sequence of *definite* Horn clauses, that is, Horn clauses with a non-empty head. The declarative semantics of a logic program P is the least set M_P of ground atoms such that if $B \leftarrow A_1, \ldots, A_n$ is a ground instance of a program clause of P and if $A_1, \ldots, A_n \in M_P$ then $B \in M_P$. Such a least set does always exist for Horn clauses (but not for arbitrary clauses).

Here is an example of logic program:

```
arc(a,b).
arc(b,c).
arc(c,d).
arc(c,e).
path(X,X).
path(X,Y) ← arc(X,Z),path(Z,Y).
```

The idea of this program is that ground terms a, b, c, etc. are nodes of a graph, *arc* defines the edges of the graph, and $path(X, Y)$ whether there is a directed path from X to Y.

While functional or imperative programs execute when provided with input data, logic programs execute when provided with *queries*, that is, a conjunction of atoms, also called a *goal.* Asking whether there is an arc from c to e is done by the query $\neg arc(c, e)$, which we could also write $\longleftarrow arc(c, e)$. Knowing that it is a query, we will simply write it $arc(c, e)$. The answer to this query should be affirmative, and this can simply be found by searching the program facts. In contrast, the answer to the query $arc(c, X)$ should be all possible nodes X to which there is an arc originating in c. Here, we need to non-deterministically unify the query with the program facts. Finally, the answer to the query $path(X, e)$ should be all nodes X from which there is a path ending in e. Here, we need to reflect the declarative semantics of our program in the search mechanism by transforming the query until we fall into the previous case. This is done by the following rule called *resolution*:

$$\text{(Resolution)}\ \frac{P_1(\bar{t}_1), P_2(\bar{t}_2), \ldots P_n(\bar{t}_n)}{Q_1(\bar{r}_1\sigma), \ldots, Q_m(\bar{r}_m\sigma), P_2(\bar{t}_2\sigma), \ldots, P_n(\bar{t}_n\sigma)}$$

where $P_1(\bar{s}) \leftarrow Q_1(\bar{r}_1), \ldots, Q_m(\bar{r}_m)$ is a program clause not sharing any variable with the query $P_1(\bar{t}_1), \ldots, P_n(\bar{t}_n)$ and σ is the most general unifier of $\bar{s}$ and $\bar{t}_1$.

This rule contains two fundamentally different non-determinisms: the choice of the atom to which resolution is applied is don't-care (it does not matter for the completeness of the search procedure which atom we choose to resolve on), while the choice of the program clause is don't-know (we have to try all possible clauses). This non-determinism is dealt with externally: choosing an atom in the query, we have in principle to execute all possible sequences of resolutions steps. Along every execution sequence we construct the sequence of substitutions computed so far. If such a computation sequence terminates in an empty query, then we say that the computation *succeeds* and that the computed substitution is an *answer substitution*, otherwise it *fails*.

In pure LP, all data items are represented as ground terms, and unification is the fundamental method of combining descriptions of data items. If data types like natural numbers are to be used in LP they have to be encoded as terms, and operations on numbers have to be expressed as LP predicates. Let us, for example, extend our example of paths in graphs to weighted graphs:

```
plus(0,X,X).
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
arc(a,b,s(0)).
arc(b,c,s(s(0))).
arc(c,d,s(0)).
arc(c,e,s(0)).
path(X,X,0).
path(X,Y,L) ← arc(X,Z,M),path(Z,Y,N),plus(M,N,L).
```

With this program, the goal $path(a, c, L)$ will succeed with answer substitution $\{L \mapsto s(s(s(0)))\}$. However, computation of the goal $plus(X, s(0), X)$ will not terminate. The problem is that, although searching solutions by enumerating all possible computation paths is complete in the limit, this gives us in general not a complete method for detecting the absence of solutions (not to speak of its terrible inefficiency). What we would like here is to have numbers as a primitive data type in LP.

Another problem with pure LP is that, even with terms as data structures, we might want to have more expressiveness for describing terms, and be able, for instance, to formulate the query "is there a path of weight 10 starting in a and ending in a node different from e"? Our approach to use terms with variables as descriptions of ground terms is obviously not sufficient here.

Constraint Logic Programming (CLP) offers a solution to these deficiencies of LP. The basic idea is to disentangle the resolution rule, by separating the process of replacing an atom of the query by the body of a program clause from the other process of unifying the data descriptions. For this, we need to consider a new syntactic entity, namely *equations* between terms. These equalities must be distinguished from possibly already existing equality atoms, since their use will be quite different. In addition, this will allow us to restrict our syntax for atoms to predicate symbols applied to *different* variables, the value of these variables being recorded via the equalities. A Horn clause is now of the form

$$B \leftarrow A_1, \ldots, A_n, c$$

where B and the A_i are atoms and c is a conjunction of equations usually written again as a set. Our example of weighted graphs could now look like this:

```
plus(X,Y,Z) ← X=0,  Y=Z.
plus(X,Y,Z) ← plus(X',Y,Z'), X=s(X'), Z=s(Z').
arc(X,Y,Z)  ← X=a, Y=b, Z=s(0).
arc(X,Y,Z)  ← X=b, Y=c, Z=s(s(0)).
arc(X,Y,Z)  ← X=c, Y=d, Z=s(0).
arc(X,Y,Z)  ← X=c, Y=e, Z=s(0).
path(X,Y,L) ← X=Y, L=0.
path(X,Y,L) ← arc(X,Z,M), path(Z,Y,N), plus(M,N,L).
```

In this new context, queries are called *configurations.* A configuration consists of a list of atoms *followed by* a list of equations. We can rewrite our resolution rule as:

$$\text{(Resolution-2)}\quad \frac{P_1(\bar{X}_1), P_2(\bar{X}_2), \ldots, P_n(\bar{X}_n), c}{Q_1(\bar{Y}_1), \ldots, Q_m(\bar{Y}_m), P_2(\bar{X}_2), \ldots, P_n(\bar{X}_n), c, d}$$

where $P_1(\bar{X}_1) \leftarrow Q_1(\bar{Y}_1), \ldots, Q_m(\bar{Y}_m), d$ is a program clause not sharing any variable with the configuration $P_1(\bar{X}_1), \ldots, P_n(\bar{X}_n), c$ except $\bar{X}_1$, and $c \wedge d$ is satisfiable.

Unification has disappeared from the resolution rule: it is replaced by constructing the conjunction of the two equation systems and checking their satisfiability. Note that in the constrained resolution rule the parameters $\bar{X}_i$ of the predicate P_i are required to be the same lists of variables in the query and in the program clause, this will have to be achieved by renaming the variables in the program clause when necessary.

Semantically, an equation $X = s$ must be understood as the set of values (for the variable X) which are the ground instances of the term s. When making a conjunction $X = s \wedge X = t$, we intersect these two sets of values, and therefore, the set of solutions is the same as that of $X = most{-}general{-}$

$instance(s, t)$, provided s and t are unifiable. This is why we need to check for satisfiability in our new rule. This satisfiability test will of course be delegated to the old unification algorithm, which is seen here as an "equation solver" which is put aside. Furthermore, we could allow more freedom in the rule by writing in the conclusion of the rule any equations system that is equivalent to $c \wedge d$. This gives the freedom of whether to leave the constraint $c \wedge d$ as it is, or to replace it by some simpler "solved form" (see the chapter *Constraint Solving on Terms* for possible definitions of solved forms).

Execution of our program is now essentially the same as before, we just have separated matters. An important observation is that we have only used the following properties of descriptions of terms:

- we can intersect their denotation (by building the conjunction of the associated equalities)
- we can decide the emptiness of the intersection of their denotations (by unification).

The crucial abstraction step is now: We obtain CLP when we replace equation systems of terms by any system of description of data with the same two properties as above. For example, we could replace the Presburger language for integers used in our last example by the usual integer arithmetic:

```
arc(X,Y,Z)  ← X=a, Y=b, Z=1.
arc(X,Y,Z)  ← X=b, Y=c, Z=2.
arc(X,Y,Z)  ← X=c, Y=d, Z=2.
arc(X,Y,Z)  ← X=c, Y=e, Z=2.
path(X,Y,L) ← X=Y, L=0.
path(X,Y,L) ← arc(X,Z,M), path(Z,Y,N), L = M + N.
```

To know more: An introduction to Logic Programming from a theoretic point of view is to be found in [Apt90].

1.2.3 Constraint Systems (First Version)

We are now ready for a first, naive version of constraint systems. First, we use predicate logic for syntax and semantics of constraints since it provides the following features which are important for most of the constraint-based formalisms:

1. first-order variables to denote values, these variables will be shared with the formalism that uses the constraints (for instance a programming language),
2. conjunction as the primary means to combine constraints, since conjunction of formulas corresponds exactly to the intersection of their respective denotation,

3. existential quantification to express locality of variables. It is existential, in contrast to universal, quantification that matters here since the constraint-based mechanisms that we consider are based on *satisfiability* of descriptions.

Hence, a first definition of a *constraint system* (we will refine this definition in Section 1.4.1) could be as follows:

A constraint system C consists of

- A language L_C of first-order logic, that is a collection of function symbols and of predicate symbols together with their respective arities. The atomic formulae built over this language are called the *atomic constraints* of C.
- a subset of the set of first-order formulae of L_C, called the set of *constraints* of C, containing all atomic constraints, and closed under conjunction, existential quantification and renaming of bound variables. Very often, the set of constraints is the minimal set containing the atomic constraints and with the above properties, in which case we call it the set of *basic* constraints of L_C.
- a first-order structure A_C of the language L_C,
- an algorithm to decide whether a constraint is satisfiable in A_C or not.

Some examples of constraint systems are:

1. *Herbrand*: This is the constraint system used in LP when seen as instance of the CLP scheme. Its language is given by an infinite supply of function symbols and equality as the only predicate. The set of constraints is the set of basic constraints whose atomic constraints are the equality atoms, the structure is the structure of ground terms, where function symbols are just constructors of ground terms and equality is completely syntactic, and the algorithm for testing satisfiability is the unification algorithm (usually attributed to Herbrand).
2. *Herbrand over rational terms*: This is the constraint system introduced by Colmerauer in PROLOG II. It is the same as previously, but the structure is the set of ground atoms built from rational terms. The algorithm for testing satisfiability is due to Huet.
3. *Herbrand with inequations*: This is another extension of Herbrand where constraints are existentially quantified systems of equations and inequations. Decidability of satisfiability of constraints has been shown by Alain Colmerauer. This constraint system was the basis of PROLOG II.
4. *Presburger arithmetic*: The language of Presburger arithmetic consists of a constant 0, a unary function s, and a binary operator $+$. Constraints are the basic constraints, interpreted in the usual way over natural numbers. There is a number of standard methods to decide satisfiability of these constraints. The set of constraints can be extended to the full set of first-order formulas. There are many ways to show decidability of the

full first-order theory, an early algorithm is due to Presburger (hence the name).

5. *Real arithmetic*: The language of real arithmetic contains constants 0 and 1 and binary operators $+$ and $*$, interpreted in the standard way over real numbers. We can take as set of constraints the set of full first-order formulas over this language. It is a surprising result of Tarski that satisfiability of these constraints is again decidable.

We will meet some more interesting constraint systems in this chapters, such as the *finite domain* constraints, and in the other chapters as well.

What happens when we extend Presburger arithmetic with multiplication? It is well-known that satisfiability of systems of polynomial equations over natural numbers is undecidable (this is Hilbert's famous 10th problem shown undecidable by Matijacevič), hence Presburger arithmetic extended by multiplication does not form a proper constraint system according to the definition given above. This is of course somewhat disappointing since multiplication is certainly something we would like to have when dealing with natural numbers. We will see later how to deal with cases where constraint solving in the general case is not decidable, or just not feasible.

To know more: See [Coh90] for an introduction to Constraint Logic Programming, and [CDJK99], [JL87] for a survey. Much more about constraint solving will be said in the chapter *Constraint Solving on Terms* and *Combining Constraint Solving.*

1.2.4 Constraints and Theorem Proving

We demonstrate a first usage of constraints in theorem proving, much more is to be seen in the chapter dedicated to the subject.

Let E be a finite set of equations, for example the classical three equations defining group theory:

$$\begin{cases} (x \times y) \times z = x \times (y \times z) \\ x \times 1 = x \\ x \times x^{-1} = 1 \end{cases}$$

A classical problem is to decide whether a given equation, for example $(x \times y)^{-1} = y^{-1} \times x^{-1}$ in group theory, is a logical consequence of E. This problem, also known as the *word problem*, has been subject to intensive research. The brute force search for a proof using the replacement of equals by equals, although complete, rarely leads to an actual solution. One of the most successful approaches due to Knuth and Bendix is to use *ordered strategies*, that is to use the equations in one way only as so-called *rewrite rules*. Of course, such a strategy is incomplete in general, but completeness can be regained using a completion mechanism based on the computation of some particular equational consequences called *critical pairs*. One requirement of the original method was the *termination* of the rewrite system: the replacement of equals by equals using the ordered strategies should always end up after a finite number of replacement steps.

In the above example of group theory, it is quite easy to fulfill this termination requirement by choosing carefully the way in which to orient the equations. The situation changes if we consider *commutative groups*, adding the equation $x \times y = y \times x$ to the above system. Now the completion procedure fails because commutativity cannot be oriented in either way without loosing termination. Several solutions have been studied to overcome this problem, the solution that interests us here is called *ordered rewriting*.

The idea is very simple: use every equation in one way or the other, depending on the ordering on *the instances on which it is applied*. For example consider the commutativity axiom and assume a total ordering on terms, e.g compare lexicographically the arguments of $\times$, from left to right. Then if $a > b$, $a \times b$ rewrites to $b \times a$ using $x \times y = y \times x$, but not the other way around, since $a \times b > b \times a$, but $b \times a \not> a \times b$.

The equations that we are trying to prove are (implicitly) universally quantified, that is an equation as $x \times y = y \times x$ should be read as "for all ground terms s and t, $s \times t = t \times s$". Hence, as in the case of (C)LP, terms with variables represent the sets of their respective ground instances. In order to describe ordered rewriting we need a total ordering relation on ground terms, denoted $>$. Now, an equation $l = r$ can be replaced by the two constrained rules $l \to r \mid l > r$ and $r \to l \mid r > l$. Since we required the order to be total, we know that for any ground instance of l and r, one of the two rules applies (except in the case $l = r$ when there is of course no need to rewrite). When we require in addition the ordering to be monotonic, that is to satisfy

$$s > t \quad \text{implies} \quad f(\ldots, s, \ldots) > f(\ldots, t, \ldots)$$

for all function symbols f and ground terms s, t, then the ordering on the rule instances propagates to the rewritten terms. Hence, rewriting always decreases terms in the ordering. Assuming then the ordering to be well-founded implies that rewriting a term always ends up in a *normal form*. This latter property is the key to completeness.

Once the equations are split into ordered rewriting rules as described above, we have to complete the rule system in order to obtain a rewrite system that allows to derive all valid equations $s = t$ by rewriting from s and t until a same term u is found. Without going into the technical details here it suffices to say that the completion procedure searches for so-called *critical pairs*, obtained when two rewrite rules $l \to r \mid l > r$ and $g \to d \mid g > d$ can overlap in the same ground term v. For this to happen, say at the top of v to make it simple, a ground substitution σ must exist such that $v = l\sigma = g\sigma$, $l\sigma > r\sigma$ and $g\sigma > d\sigma$. Starting from v, rewriting can yield the two different terms $r\sigma$ and $d\sigma$, and we have to repair this divergence by adding additional rewrite rules. But since the substitution σ is a solution of the *ordering constraint* $l = g \wedge l > r \wedge g > d$, the divergence can be repaired by adding the equation (called critical pair) $r = d \mid l = g \wedge l > r \wedge g > d$, more precisely, the two rules $r \to d \mid l = g \wedge l > r \wedge g > d \wedge r > d$ and $d \to r \mid l = g \wedge l > r \wedge g > d \wedge d > r$. Since we know that the ordering is total, one of the two rules will rewrite one -we don't know which one, of course- of $(r\sigma, d\sigma)$ into the other. This process of adding critical pairs may terminate with a so-called *canonical* rewrite system, that is one which can prove all valid equations by rewriting, or run forever, in which case any valid equation will be provable by rewriting after some time.

Here, we have seen an application of constraints in theorem proving in which constraints play already the role of *defining* the notion of proof, via the definition of ordered rewriting. Constraints can also be used in a completely different way: to restrict the set of possible inferences. In the above discussion, we have overlapped two rules at the top. In practice, we must actually overlap left-hand sides of rules at arbitrary non-variable subterms. Assume now that we can solve the constraints, and replace the ordered rules $l \to r \mid l > r$ by plain rewrite rules $l\sigma \to r\sigma$, where σ is a kind of mgu for the constraint $l > r$. Then overlapping these new rules does not have the same effect at all than overlapping the constrained one they originate from. In the first case, the overlap may happen *inside* the substitution σ, which is impossible for the constrained rule: the use of constraints allows to avoid overlapping inside substitutions originating from previous inferences. This strategy is called *basic*. It is again complete, although it influences in various ways the whole inference process.

To know more: The reader is referred to the chapter *Constraints and Theorem Proving* for a more complete exposition. See [CT94, Nie99] for some results on ordering constraints on trees and their usage in automated deduction.

1.2.5 Constraint Satisfaction Problems

So far we have seen that constraints are useful to define an expressive programming paradigm and as a tool to express strategies in automated deduction. Many important problems already employ constraints in the problem formulation, and the task is to find a solution to these constraints. We demonstrate this with a well known game:

$$\begin{array}{rccccc} & & S & E & N & D \\ + & & M & O & R & E \\ \hline = & M & O & N & E & Y \end{array}$$

Each letter denotes a different number between 0 and 9 such that, when read as an addition, the equation is satisfied.

This problem can be easily formalized in our logical setting, by the following program clauses (we omit the clauses defining *plus*, *mul*, *in* and *neq* predicates, and use $10, 100, 1000, 10000$ as abbreviations for their representation using 0 and *succ*):

```
solution(S,E,N,D,M,O,R,Y) :-
/* X_1 is the value of the first row */
   mul(1000,S,Y_1), mul(100,E,Y_2), mul(10,N,Y_3),
   plus(Y_1,Y_2,Y_4), plus(Y_3,Y_4,Y_5), plus(D,Y_5,X_1),
/* X_2 is the value of the second row */
   mul(1000,M,Z_1), mul(100,O,Z_2), mul(10,R,Z_3),
   plus(Z_1,Z_2,Z_4), plus(Z_3,Z_4,Z_5), plus(E,Z_5,X_2),
/* X_3 is the value of the third row */
   mul(10000,M,T_1), mul(1000,O,T_2), mul(100,N,T_3),
   plus(T_1,T_2,T_5), plus(T_3,T_5,T_6), plus(T_4,T_6,T_7),
   mul(10,E,T_4), plus(Y,T_7,X_3),
/* value of row1 + value of row2 = value of row3 */
   plus (X_1,X_2,X_3),
/* S ... Y between 0 and 9 */
   in(S,0,9), in(E,0,9), ... , in(Y,0,9),
/* S ... Y all different */
   neq(S,E), neq(S,N), ... , neq(R,Y).
```

Prompted by the query $plus(X, Y, Z)$, resolution will search for the values of all the variables. This brute force approach consists in generating all $10 * 9 * \ldots * 3 = 1814400$ assignments of different digits to the letters and checking whether they satisfy the clauses.

We can of course skip this logical formulation, and formulate the problem directly as a constraint satisfaction problem of integer arithmetic, using appropriate predicates and function symbols:

$$\begin{aligned}
&1000 * S + 100 * E + 10 * N + D + 1000 * M + 100 * O + 10 * R + E \\
&\qquad = 10000 * M + 1000 * O + 100 * N + 10 * E + Y \\
&\wedge S \in [1..9] \wedge E \in [0..9] \wedge N \in [0..9] \wedge D \in [0..9] \\
&\wedge M \in [1..9] \wedge O \in [0..9] \wedge R \in [0..9] \wedge Y \in [0..9] \\
&\wedge \neq (S, E, N, D, M, O, R, Y)
\end{aligned}$$

This formulation is much more concise and elegant, of course. It uses a so-called *global* constraint based on the varyadic predicate $\neq$, called *all_diff*, which is true when all its arguments take different integer values. Again, we can solve the constraint by a brute force enumeration of the values of the variables, which would yield about the same amount of work as previously. Modern constraint programming systems, however, can solve this problem in a fraction of a second using a much more clever approach based on *constraint propagation*. Constraint propagation implements a kind of meta-reasoning on the constraints themselves, in order to cut down the search. For an example,

M must be equal to 1, since it is a carry on. In turn, this implies that S must be equal to 8 or 9. Constraint propagation on intervals (also called *finite domains*) was first implemented in the CHIP system [DSH90].

Although the above discussion is certainly just a toy example, this kind of application is of paramount importance in practice. Problems of this kind occur, for instance, in all kind of scheduling problems (think of making a staff plan under restricted availability of the persons, scheduling task among different machines where the execution of tasks has to obey to a certain order, making a timetable for a school, etc.). A more detailed account of this approach is given in Section 1.5.4.

To know more: More about this kind of application is to be told in the chapter *Building Industrial Applications with Constraint Solving.*

1.2.6 Constraint Based Grammar Formalisms

So called *unification grammars* have a long history in computational linguistics where they are employed for the analysis and generation of natural language. Originally, these grammars were defined without using constraints or even predicate logic. The name “unification grammar” is a bit unfortunate since “unification” is the operation of resolving equations, calling them *constraint grammars* is much more appropriate since this name emphasizes the declarative aspect of the formalism. However, these constraint grammars are usually not considered as a generic formalism that can be used with any constraint system (as it is the case with constraint logic programming) but come with so-called feature constraints that we will see in an example below, and that we will investigate in greater depth in Section 1.3.

We will demonstrate the principle with a very simple example. Consider a context-free grammar for English which contains the following rules:

$$\begin{aligned} S &\to NP\ VP \\ NP &\to D\ N \\ VP &\to V\ NP \end{aligned}$$

where the non-terminals stand for *Subject*, *Noun Phrase*, *Verb Phrase*, *Determiner*, and *Noun* respectively. Additionally, there are so-called lexical rules that represent a dictionary:

$N \to$ `girl`

$N \to$ `movie`

$N \to$ `movies`

$D \to$ `a`

$D \to$ `the`

$V \to$ `watches`

This grammars licenses correct English sentences like

```
the girl watches a movie
```

but also incorrect sentences like

```
the girl watches a movies
```

The problem is that context-free grammar rules can only define the possible skeletons of a phrase, however, there are additional conditions to be met for a sentence to be grammatically correct, as for instance correspondence of the numeros of the subject and the verb.

Hence, we augment our context-free grammar with constraints. While a context-free grammar describes a set of parse trees, a constraint grammar describes a set of attributed parse trees. In such an attributed parse tree, nodes can be decorated with functional attributes called *features*, and the constraints in the grammar impose relations between the attributes of the nodes for an attributed parse tree to be admissible, that is to describe a correct English sentence. For example, we might express the correspondence of the numeros of subject and verb by adding some constraints to the structural and to the lexical grammar:

$$\begin{aligned}
S &\to NP\ VP \mid NP.numeros = VP.numeros \\
NP &\to D\ N \mid NP.numeros = D.numeros \wedge D.numeros = N.numeros \\
VP &\to V\ NP \mid VP.numeros = V.numeros \\
N &\to \texttt{girl} \mid N.numeros = singular \\
N &\to \texttt{movie} \mid N.numeros = singular \\
N &\to \texttt{movies} \mid N.numeros = plural \\
D &\to \texttt{a} \mid D.numeros = singular \\
D &\to \texttt{the} \\
V &\to \texttt{watches} \mid V.numeros = singular
\end{aligned}$$

Now, the constraints only allow sentences where the numeros of the noun phrase coincides with the numeros of the verb phrase and where in a noun phrase the numeros of the determiner coincides with the numeros of the noun. An interesting point to note about this grammar is that constraints are used to accomplish two different tasks:

- to express restrictions on the attributes of components, for instance that in a noun phrase the numeros of determiner and noun must coincide; and
- to define the attributes of nodes, for instance the numeros of a noun phrase is defined to be the numeros of its determiner.

Also, note that we did not specify the numeros of the determiner `the` which might be singular or plural. Hence, our grammar licenses both noun phrases `the movie` and `the movies`.

To know more: This section follows the exposition of constraint grammars in [Smo92]. See [Kel93] for the use of feature constraints in computational linguistics.

1.2.7 Constraint-Based Program Analysis

Constraint based program analysis stands a bit apart from the constraint-based formalisms mentioned before since here constraints are not used as a description language to augment the descriptive power of another formalism (programming language, deduction calculus, grammar) but are rather used as a means to *approximate* the meaning of a program.

Constraint-based program analysis uses *set constraints* as a language to approximate the meaning of a program. Different systems of set constraints have been investigated during the last decade, here we use the constraint system of [HJ90].

The logical language of our set constraint system contains a signature Σ, a binary union operator, and a constant $\top$. In addition, the language contains for every n-ary function symbol $f \in \Sigma$ so called *projections* $f_1^{-1}, \ldots, f_n^{-1}$ which are all unary function symbols. The only predicate symbol is the binary symbol $=$. Constraints are possibly existentially quantified conjunctions of atomic constraints.

Basic data items in the interpretation of set constraints are *sets of ground terms.* Function symbols operate now on sets, that is the meaning of $f(M, N)$ is $\{f(m, n) \mid m \in M, n \in N\}$. The symbol $\top$ denotes the set of all ground terms, $\cup$ and $=$ have their usual meaning, and the meaning of $f_i^{-1}(M)$ is

$$\{m_i \mid \exists m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_n \, f(m_1, \ldots, m_i, \ldots, m_n) \in M\}$$

Satisfiability of these constraints is decidable but non-trivial, see the literature for pointers.

As a simple example we demonstrate the approximation of the meaning of a logic program; the method works as well for functional or imperative programs. Consider the well-known logical program for the concatenation of lists:

```
conc(nil,V,V).
conc(H:X,Y,H:Z) ← conc(X,Y,Z).
```

No variable is shared among clauses, this is necessary for the abstraction to work (and can easily be achieved, the case given, by renaming of variables). The abstraction uses one set variable Γ, corresponding to the predicate *conc*, and variables V, H, X, Y, Z. First, we state that a successful computation of *conc* can be obtained by starting from the first clause or by starting from the second clause:

$$\Gamma = conc(nil, V, V) \cup conc(H : X, Y, H : Z)$$

To get constraints for the variables V, X, Y, Z, H we have to look at the clauses where they occur (note that every of these variables has to occur in exactly one clause). The variables V and H do not occur in the bodies of their clause, hence there is no restriction on them and they can denote any term:

$$V = \top$$
$$H = \top$$

The variables X, Y and Z can assume any value that stems from *some* successful computation of *conc*. More precisely, X can take any value such that for *some value* of Y and Z the computation of $conc(X, Y, Z)$ is successful. This is were the approximation happens: the point-wise dependencies between variables are ignored. We get the constraints

$$X = conc_1^{-1}(\Gamma)$$
$$Y = conc_2^{-1}(\Gamma)$$
$$Z = conc_3^{-1}(\Gamma)$$

Hence, the complete set constraint to resolve is

$$\Gamma = conc(nil, V, V) \cup conc(H : X, Y, H : Z) \wedge V = \top \wedge H = \top \wedge$$
$$X = conc_1^{-1}(\Gamma) \wedge Y = conc_2^{-1}(\Gamma) \wedge Z = conc_3^{-1}(\Gamma)$$

This constraint is satisfiable, its minimal solution assigns to V, H, Y and Z the set of all ground terms, and to X the set of all lists.

To know more: A recent survey and bibliography of constraint-based program analysis can be found in [PP97].

1.2.8 Constraint-Based Model Checking

Reactive systems are traditionally modeled by finite labelled graphs, the vertices and edges of which denoting respectively the set of states and the set of transitions between states under the action of the events labelling the edges. Because of the finiteness assumption, these graphs can be seen as automata whose alphabet is the set of events. Safety analysis is concerned with the problem whether a given set of (forbidden) states is accessible from the initial one. This question is of course decidable by computing the set of all reachable states.

The actual description of a reactive system is usually given by a set of concurrent programs updating various variables. A state of the system can then be thought of as a mapping from the set of variables to the set of values they may take, while the events are interpreted as state updates. In this view, the above finiteness assumption becomes overly restrictive for modeling practical applications, since it rules out variables taking their value in an infinite set. A more general view based on the idea of Myhill and Nerode's theorem allows for variables taking (possibly infinite) sets of values in a given state, by merging those states having the same behavior with respect to the set of events. Constraints may then serve to describe the set of values taken by the set of variables in a given state via their (possibly infinitely many) solutions. In practice, the constraints are actually used to label the transitions, and are interpreted both as guards and updates. Depending on the constraint system used in this more general setting, the computation of the set of accessible states may still be in PSPACE, or of a higher complexity, or even become undecidable.

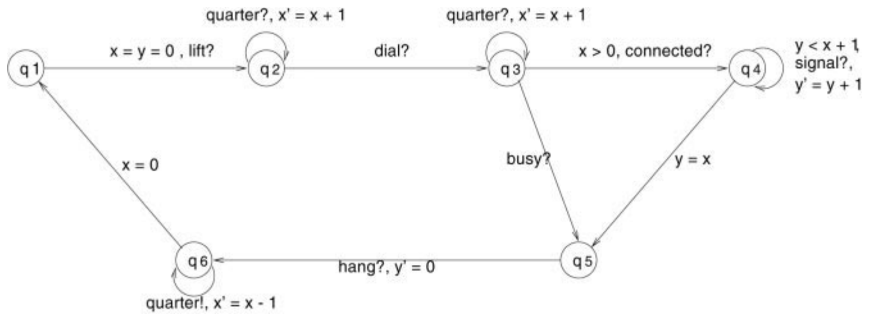


Fig. 1.1. A pay phone controller

The example of a pay phone controller given in Figure 1.1 is from [CJ99a]. There are two variables ranging over natural numbers usually called counters, x for the available credit (number of quarters fed in the machine), and y for the number of time units already used by the customer. The automaton may receive signals such as “quarter?”, meaning that a coin has been fed in the machine, “connected?”, meaning that the connection has been set up, as well as “busy?”, “dial?”, “lift?”, “hang?” and “signal?” whose meaning is self explanatory. The automaton may itself emit the signal “quarter!” meaning that a quarter will be given back to the user. Transitions occur under the reception or emission of a given signal, and may require the satisfaction of some formula relating the variables x and y, as well as their updated values x' and y'. A formula like $x = y$ should then be understood as a guard, while a formula like $y' = 0$ should be understood as an update of the variable y. The following is a valid transition sequence of the automaton, where the name of a state comes with with the values of x and y successively:

$$\begin{aligned}(q_1,0,0) &\xrightarrow{lift?} (q_2,0,0) \xrightarrow{quarter?} (q_2,1,0) \xrightarrow{quarter?} (q_2,2,0)\\ &\xrightarrow{dial?} (q_3,2,0) \xrightarrow{connected?} (q4,2,0) \xrightarrow{signal?} (q_4,2,1)\end{aligned}$$

For instance, the last transition step is acceptable because $(2,0),(2,1) \models y < x+1 \wedge y' = y+1$. Note that two successive states are needed to verify if a guard is satisfied when it involves primed variables.

Here, safety (from the point of view of the telephone company) will require that, e.g., a connection cannot be established without paying beforehand. In the case above, safety properties are decidable. This is shown by reducing the problem to the emptiness problem for the kind of automaton used in this example, and showing that the latter property is decidable.

The constraints used in this example belong to Presburger arithmetic, and indeed, such constraints occur naturally in reactive systems when modeling the various resources used by the system (in our example, the quarters). Discrete time is a resource that can be naturally modeled by a natural number, giving rise to such constraints. But discrete time implies the existence of a global clock monitoring all processes in the system. When there are several independent clocks in a reactive system, discrete time cannot be used anymore, and must be replaced by real time (or an approximation of it). Then, constraints belong to Tarski's theory of real arithmetic. Guards expressing geometric conditions also use real arithmetic.

Unfortunately, decidability of a constraint system is not enough to ensure the decidability of the emptiness problem for a class of automata whose guards are expressed in this system. For example, the emptiness problem for automata with multiple counters using full Presburger arithmetic for expressing the guards is undecidable (since Minsky machines are an instance of this model). Various restrictions (of the constraint language, or of the automata-based underlying model) have been explored for which decidability is recovered.

To know more: The use of constraint in the area of model checking has become quite popular during the last 5 years. See, for example, [FP94, CJ99b] and [DP00, Pod00]. See also, e.g., [BVW94, FO97, CJ99a].

1.3 A Case Study of a Constraint System: Feature Constraints

1.3.1 Motivation

The constraint system Herbrand, that is first-order terms, is often used to model symbolic data. For many applications, however, this constraint system turns out to be too inflexible. This inflexibility has two sources, one is the use of classical first-order trees as underlying data model, and the other is the particular set of atomic constraints provided by the Herbrand constraint system.

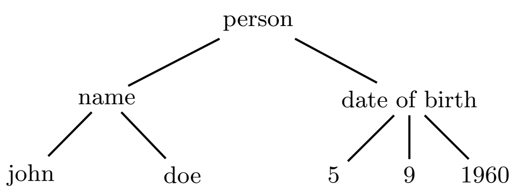


Fig. 1.2. Classical terms to represent data.

The problem with using classical trees as a data model is that the subtrees of a tree described in the Herbrand constraint system are numbered, that is we always have to remember the meaning that we have in mind with the first, second, third, ... subtree of a tree. In the example of Figure 1.2 we have to remember that in the subtree labeled *name* the first component is the first name and the second component is the family name, and that the date of birth is stored in the format day-month-year and not, for instance, month-date-year. In many applications the use of symbolic names to denote subtrees is much more natural. This is illustrated in Figure 1.3 where we used a tree with labeled edges, a so-called *feature tree*, to represent some data about a movie (an exact definition of feature trees will follow soon). Using arbitrary names to denote components recalls of course the notion of *record* which is well-known from many programming languages.

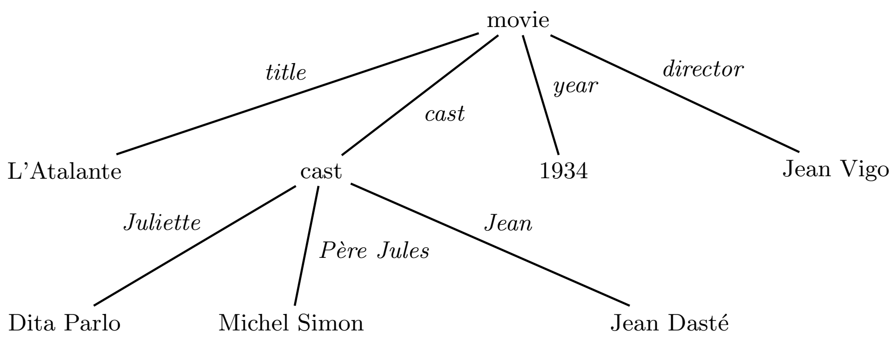


Fig. 1.3. Feature trees to represent data.

The second problem with the Herbrand constraint system comes from its choice of atomic constraints. If we write a constraint $x = f(y, z)$ then we are expressing several things at the same time:

1. that the root of the tree x is labeled with the symbol f,
2. that there are exactly two subtrees at the root of the tree f,
3. that the first subtree of x is denoted by y, and that the second subtree of x is denoted by z.

That the root of x is labeled by f is easily expressed in Herbrand by using an existential quantifier:

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

assuming that f is known to be binary. The other properties, however, are much more cumbersome to express in Herbrand since they require the use of *disjunctions*:

$$\bigvee_{f \in \Sigma_2} \exists y, z\, x = f(y, z) \qquad x \text{ has two subtrees}$$

$$\bigvee_{f \in \Sigma_2} x = f(y, z) \qquad \text{the first (second) subtree of } x \text{ is } y \ (z)$$

where Σ_2 is the set of binary function symbols. Note that disjunctions are not available in the Herbrand constraint system, hence we either have to extend the Herbrand system accordingly, or we have to accommodate for disjunctions in the programming formalism. Even worse, the signature might not be known in advance since we want to have the freedom to use any function symbol in the course of computation (imagine a system where the user interactively enters data items, using any function symbols he likes). Not knowing the signature in advance is usually modeled by assuming the signature to be infinite, for instance to be the set of all words formed over some alphabet. In this case the above disjunctions are not even formulas of first-order predicate logic.

This is where *feature constraints* come in: Atomic feature constraints are more fine-grained than Herbrand-equations. In the constraint system CFT, for example, there are exactly the three kinds of atomic constraints that correspond to the three properties listed above: *label constraints* Ax (sometimes called *sort constraints*) to specify the symbol at the root node of a tree, *arity constraints* $x\{f_1, \ldots, f_n\}$ to specify the labels of the edges departing from the root, and *subtree* constraints $x[f]y$ to specify the subtree of a tree at a specific edge.

1.3.2 The Constraint System CFT

Formal Definition. We assume an infinite set $\mathcal{F}$ of *features* ranged over by f, g, h and an infinite set $\mathcal{L}$ of *labels* ranged over by A, B, C.

A *path* π is a word of features. The *empty path* is denoted by ϵ and the concatenation of paths π and π' as $\pi\pi'$. A path π' is called a *prefix of* π if $\pi = \pi'\pi''$ for some path π''. A *tree domain* is a non-empty prefix-closed set

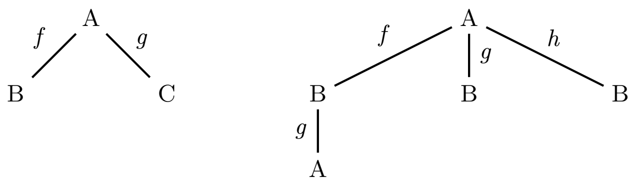


Fig. 1.4. Some simple feature trees.

of paths, that is a set of paths that contains all prefixes of its members. As a consequence, every tree domain contains the empty path ϵ.

A *feature tree* τ is a pair (D, L) consisting of a tree domain D and a function $L : D \to \mathcal{L}$ that we call *labeling function* of τ. Given a feature tree τ, we write D_τ for its tree domain and L_τ for its labeling function. For instance, $\tau_0 = (\{\epsilon, f, g\}, \{(\epsilon, A), (f, B), (g, C)\})$ is a feature tree with domain $D_{\tau_0} = \{\epsilon, f, g\}$ and $L_{\tau_0} = \{(\epsilon, A), (f, B), (g, C)\}$, depicted to the left of Figure 1.4. The *root* of τ is the node ε. The *root label* of τ is $L_\tau(\varepsilon)$, and $f \in \mathcal{F}$ is a *root feature* of τ if $f \in D_\tau$. Given a tree τ and a root feature f of τ, we write as $\tau[f]$ the subtree of τ at path f; formally $D_{\tau[f]} = \{\pi' \mid f\pi' \in D_\tau\}$ and $L_{\tau[f]} = \{(\pi', A) \mid (f\pi', A) \in L_\tau\}$.

An *atomic CFT constraint* is one of the following forms:

$$A(x),\ x[f]y,\ x\{f_1, \dots, f_n\},\ x = y$$

where $A \in \mathcal{L}$ and $f, f_1, \dots, f_n \in \mathcal{F}$. A CFT constraint is a possibly existentially quantified conjunction of atomic CFT constraints.

We define the structure CFT over feature trees in which we interpret CFT constraints. Its universe consists of the set of all feature trees. The constraints are interpreted as follows:

$$\begin{aligned} \tau_1[f]\tau_2 &\text{ iff } \tau_2 = \tau_1[f] \\ A(\tau) &\text{ iff } (\varepsilon, A) \in L_\tau \\ \tau F &\text{ iff } F = D_\tau \cap \mathcal{F} \end{aligned}$$

and $=$ is of course equality. Hence, $\tau_1[f]\tau_2$ means that τ_2 is the subtree of τ_1 at feature f, $A(\tau)$ means that A is the root label of τ, and τF means that F is the set of root features of τ.

For instance, the following constraint can be satisfied by choosing as value of the variable x any of the two trees depicted on Figure 1.4:

$$\exists y, z \left(A(x) \wedge x[f]y \wedge B(y) \wedge x[g]z \wedge z\emptyset \right)$$

We can extend this constraint such that only the left tree of Figure 1.4 is a solution but not the right subtree, by adding any of the following atomic constraints:

$$x\{f, g\}, y\emptyset, C(z)$$

Satisfiability of CFT Constraints. Since existential quantifiers are not relevant when testing satisfiability of constraints we can simply ignore them for the moment, hence for the purpose of checking satisfiability of constraints we assume that a constraint is just a conjunction of atomic constraints.

A CFT constraint c is said to be a *solved form* if it contains no equation and

1. if $x[f]y \in c$ and $x[f]z \in c$ then $y = z$
2. if $A(x) \in c$ and $B(x) \in c$ then $A = B$
3. if $xF \in c$ and $x[f]y \in c$ then $f \in F$
4. if $xF \in c$ and $xG \in c$ then $F = G$

A variable x is called *constrained* by a solved form c if c contains an atomic constraint of one of the forms $A(x)$, xF or $x[f]y$. The set of constrained variables of c is written $C(c)$. For instance, the following constraint is indeed a solved form, and its constrained variables are x and y.

$$A(x) \wedge x[f]y \wedge x[g]z \wedge y\{h, k\} \wedge y[h]x$$

Solved forms are satisfiable. However, we can say more: Whatever values we choose for unconstrained variables there are always values for the constrained variables such that the constraint is satisfied. This is expressed by the following proposition:

Proposition 1. *For every solved form c we have that*

$$CFT \models \forall \bar{x}\ \exists \bar{y}\ c$$

where $\bar{x} = V(c) - C(c)$ and $\bar{y} = C(c)$

Hence, in particular, every solved form is satisfiable. In continuation of the above example we get that the following formula is valid:

$$\forall z\ \exists x, y \Big(A(x) \wedge x[f]y \wedge x[g]z \wedge y\{h, k\} \wedge y[h]x \Big)$$

Proof. (Sketch) Beware that the solved form may contain cycles between constrained variables, as between x and y in the above example. Given feature trees as values of the non-constrained variables, we can see a solved form as a function mapping a tuple of feature trees (indexed by the constrained variables) into a tuple of feature trees of the same kind. We get a solution of the solved form as the minimal fixed point of this function (we can assume w.l.o.g. that there is a label constraint for every constrained variable). This fixed point is obtained by starting with a tuple of trees containing just a root node plus the subtrees corresponding to the values of the unconstrained variables, and taking the least upper bound of the iterative application of the function.

We describe now a *rule-based* algorithm for deciding satisfiability of CFT constraints. More examples of rule-based algorithms will be presented in the chapter *Constraint Solving on Terms*. Such a rule-based algorithm is given by a set of rules transforming constraints into constraints. This yields a decision algorithm for satisfiability provided that the three following properties hold:

1. Soundness: Application of the rules preserves semantics.
2. Termination: There is no infinite sequence of rule applications.
3. Completeness: If no rules applies then we have reached a special form from which we can immediately tell whether it is satisfiable or not.

Note that confluence of the calculus is not necessary here since we require preservation of semantics, hence different normal forms of the same constraint are logically equivalent.

An equation $x = y$ is *eliminated* in a constraint ϕ if $x = y$ is an atomic constraint of ϕ, y is a variable different from x, and at least one of x and y occurs only once in ϕ (hence this occurrence is the one in the equation $x = y$). In this case we also say that the variable occurring only once is *eliminated* in ϕ.

$$\text{(Eq)} \quad \frac{x = y \wedge \phi}{x = y \wedge \phi\{x \mapsto y\}} \quad x \text{ and } y \text{ occur in } \phi,\ x \neq y$$

$$\text{(Triv)} \quad \frac{x = x \wedge \phi}{\phi}$$

$$\text{(SClash)} \quad \frac{Ax \wedge Bx \wedge \phi}{\bot} \quad A \neq B$$

$$\text{(Fun)} \quad \frac{x[f]y \wedge x[f]z \wedge \phi}{x[f]z \wedge y = z \wedge \phi}$$

$$\text{(AClash)} \quad \frac{xF \wedge xG \wedge \phi}{\bot} \quad F \neq G$$

$$\text{(FClash)} \quad \frac{xF \wedge x[f]y \wedge \phi}{\bot} \quad f \notin F$$

Fig. 1.5. Simplification rules for CFT.

The rules are given in Figure 1.5. We write $c \rightarrow_{CFT} d$ if the constraint c reduces in one step by the system of Figure 1.5 to the constraint d.

Here is an example of a reduction to solved form. The redeces of the rule applications are underlined.

$$\begin{array}{l}
\underline{x = y} \wedge x[f]x' \wedge y[f]y' \wedge Ax' \wedge By' \\
x = y \wedge \underline{y[f]x'} \wedge \underline{y[f]y'} \wedge Ax' \wedge By' \\
x = y \wedge \underline{x' = y'} \wedge y[f]y' \wedge Ax' \wedge By' \\
x = y \wedge x' = y' \wedge y[f]y' \wedge \underline{Ay'} \wedge \underline{By'} \\
\perp
\end{array}$$

Proposition 2. *If $c \rightarrow^*_{CFT} d$ then $CFT \models c \Leftrightarrow d$.*

Proof. This is immediate by definition of the semantics of CFT. □

Proposition 3. *The system of Figure 1.5 is terminating.*

Proof. We show this using a lexicographic combination of measuring function ψ_1, ψ_2, ..., each mapping a constraint to some natural number. To show termination of the system it is sufficient to show that if c reduces to d then there is some i such that ψ_1 to ψ_{i-1} are unchanged and ψ_i strictly decreases.

More precisely, we define three measuring functions:

- $\psi_1(\phi)$ is the number of subtree constraints of ϕ,
- $\psi_2(\phi)$ is the number of atomic constraints of ϕ,
- $\psi_3(\phi)$ is the number of variables of ϕ that are *not* eliminated in ϕ.

The termination argument is now given by the following table:

Rule	ψ_1	ψ_2	ψ_3
Eq	$=$	$=$	$<$
Triv	$=$	$<$	
Fun	$<$		
{S,A,F}Clash	$\leq$	$<$	

□

Proposition 4. *Any constraint which is not reducible by $\rightarrow_{CFT}$ is either $\perp$ or of the form $e \wedge c$ where c is a solved form and e is a system of equations that are eliminated in $e \wedge c$.*

Theorem 1. *Satisfiability of CFT-constraints is decidable in quasi-linear time.*

Proof. Given a constraint c, we apply the transformation rules of Figure 1.5 in any order until we obtain a constraint d that is no longer reducible. This process terminates by Proposition 3. By Proposition 4, d is either $\perp$ and hence not satisfiable, or of the form $e \wedge c$, where c is a solved form and e is eliminated, and hence satisfiable. By Proposition 2, c is satisfiable iff d is.

It is quite easy to see that the number of possible reductions is linear in the number of atomic constraints of c, hence a normal form of c can be found in polynomial time. An algorithm that exhibits quasi-linear complexity can be found in the literature. □

Note that this complexity result is about the so-called off-line complexity, that is it applies to the situation where the input is the complete constraint. In case of the CLP calculus presented in Section 1.2.2, however, the constraint has to be checked for satisfiability at each stage of the computation. Hence, when we build a constraint consisting of n atomic constraint by successively adding a constraint and checking the conjunction for satisfiability then the total cost sums up to $O(n^2)$. Although this is so far not in the calculus (it will be realized in the calculus of Section 1.4), it is straightforward to extend the programming calculus in such a way that the constraint is stored in a logically equivalent *simplified* form. Then it makes sense to ask for the on-line complexity of the problem: Given a sequence of atomic constraints, $a_1, \dots, a_n$, what is the complexity of determining the smallest i such that $a_1, \dots, a_i$ is unsatisfiable? An algorithm solving this problem can store the constraints seen so far in a simplified form. The abstract machine presented in [ST94], based on this principle, has a quasi-linear complexity even in this on-line scenario.

Negated Constraints and Independence. We might now try to add negated constraints to the constraint system CFT, thus considering constraints like

$$Ax \wedge x[f]y \wedge \neg\exists z\Big(x[g]z \wedge Bz\Big) \wedge \neg\exists v\Big(x[g]v \wedge v\{f,g,h\}\Big)$$

Negative constraints provide additional expressivity. There is another important motivation, stemming from the entailment problem considered below, which we will encounter in Section 1.5.

The treatment of conjunctions of positive and negative constraints is simplified by an important property of CFT: The independence of negative constraints (or short: independence).

A constraint system is called *independent* if for all constraints $\gamma, \gamma_1, \dots, \gamma_n$ (remember that we require the set of constraints to be closed under existential quantification) the following holds:

$$\gamma \wedge \neg\gamma_1 \wedge \ldots \wedge \neg\gamma_n \text{ is satisfiable} \quad \Leftrightarrow \quad \text{for all } i\text{: } \gamma \wedge \neg\gamma_i \text{ is satisfiable}$$

This is equivalent to say that

$$\gamma \models \gamma_1 \vee \ldots \vee \gamma_n \quad \Leftrightarrow \quad \text{there is an } i \text{ such that } \gamma \models \gamma_i$$

Hence, independence allows us to reduce the problem of checking that a conjunction of positive and negative constraints $\gamma \wedge \neg\gamma_1 \ldots \wedge \neg\gamma_n$ is satisfiable, to the problem of checking that for all indices i the entailment judgment $\gamma \models \gamma_i$ does *not* hold.

An example of a constraint system that is not independent is (Presburger or real) arithmetic with multiplication: $x * y = 0 \models x = 0 \vee y = 0$, but $x * y = 0$ does not imply any of $x = 0$ or $y = 0$.

CFT enjoys the independence property. This is the case only because we required the set of label symbols and the set of feature symbols to be infinite. If the set of labels is finite, say $\{A_1, \ldots, A_n\}$, then independence does not hold since *true* $\models (A_1x \vee \ldots \vee A_nx)$ but *true* implies no particular A_ix. A similar counterexample can be found if the set of features is finite.

Constraint Entailment. Hence, in order to cope with positive and negative constraints we have to solve the so-called *entailment* problem: Given constraints γ and γ', does $\gamma \models \gamma'$ hold? In other words, is the set of data items described by constraint γ' contained in the set of data items described by constraint γ? Besides its use for testing satisfiability of positive and negative constraints, entailment is of paramount importance in the paradigm of constraint programming (see Section 1.5).

The first observation is that, in an entailment problem $\gamma \models \gamma'$, we can assume γ to contain no existential quantifier. This is justified by the following equivalences, where we can assume without loss of generality that the set of local variables of γ, $\bar{x}$, is disjoint with $V(\gamma')$ (otherwise we can rename the local variables of γ since CFT, as all constraint systems, is closed under renaming of bound variables):

$$\begin{aligned} & (\exists \bar{x}(a_1 \wedge \ldots \wedge a_n)) \models \gamma' \\ \Leftrightarrow\ & \forall \bar{x}((a_1 \wedge \ldots \wedge a_n) \Rightarrow \gamma') \text{ is valid} \\ \Leftrightarrow\ & (a_1 \wedge \ldots \wedge a_n) \models \gamma' \end{aligned}$$

The existential quantifier on the right-hand side of the entailment problem, however, is essential and can not be dropped: the entailment judgment $x\{f\} \models \exists y\, x[f]y$ holds, however $x\{f\} \models x[f]y$ does not hold.

We will demonstrate the test of entailment of CFT constraints with some examples. The exact criterion can be found in the literature. The judgment

$$x[f]y \wedge Ay \wedge x[f]z \wedge Bz \models \exists v\, x[g]v$$

holds since the left-hand side is not satisfiable, the rules of Figure 1.5 reduce the left-hand side to $\perp$. Hence the entailment judgment is vacuously true. The judgment

$$x[f]y \wedge Ay \models \exists z(x[f]z \wedge Bz)$$

does *not* hold since the left-hand side is satisfiable, but both sides together are not. Hence, there is a valuation that makes the left-hand side true and the right-hand side false, that is, implication does not hold. The judgment

$$x[f]y \wedge Ay \wedge x[g]z \wedge Bz \models \exists v(x[f]v \wedge x[g]z)$$

holds since the information on the right-hand side is directly contained in the left-hand side after proper instantiation of the local variable v. The judgment

$$x\{f, g\} \wedge Ax \wedge x[f]z \wedge y\{f, g\} \wedge Ay \wedge y[f]z \models x = y$$

does *not* hold since the values of x and y might differ at the feature g. The judgment

$$x\{f,g\} \wedge Ax \wedge x[f]x \wedge x[g]y \wedge y\{f,g\} \wedge Ay \wedge y[f]x \wedge y[g]y \models x = y$$

does hold. This case needs a more involved reasoning than the preceding cases:

We say that a variable x is *determined* by a constraint if the constraint contains a label constraint Ax, an arity constraint $x\{f_1,\ldots,f_n\}$ and selection constraints $x[f_i]y_i$ for each $i = 1,\ldots,n$. In the above example, x and y are both determined. The left-hand side does not contain any other variable, the consequence is that the left-hand-side has *exactly one solution.*

Furthermore, if we identify in the left-hand side x with y then we don't get a contradiction with the rules of Figure 1.5. Hence, there is a common solution of both sides. Since the left-hand side allows only one solution, we conclude that *all* solutions of the left-hand side also satisfy the right-hand side, that is that entailment holds. Furthermore, the judgment

$$x\{f,g\} \wedge Ax \wedge x[f]x \wedge x[g]z \wedge y\{f,g\} \wedge Ay \wedge y[f]x \wedge y[g]z \models x = y$$

also holds. This is a generalization of the preceding case since we now have a non-determined variable (that is, z) on the left-hand side. The argument is similar to the preceding case, except that now the left-hand side has for every value of z exactly one solution in the determined variables x and y, and that the left-hand side under the identification $x = y$ is satisfiable for any value of z.

The exact result is

Theorem 2. *Entailment of CFT constraints can be checked in quasi-linear time.*

To know more: A much more detailed exposition with an exact statement of the entailment criterion, proofs and the description of an abstract machine for testing satisfiability and entailment of CFT constraints can be found in [ST94].

1.3.3 First-Class Features

The feature constraint system CFT presented above has one serious limitation: feature symbols are fixed symbols. Variables denote feature trees but not feature symbols! It would be nice if variables could also denote feature symbols, that is if feature symbols would be given *first-class status.* This would allow much more expressivity, for instance we could just *program* an iteration over all root features of a feature tree.

We thus define the feature constraint system F: The logical language now contains two sorts: features and trees. Feature symbols are now constants of sort feature, and subtree selection $x[v]y$ is now a ternary predicate symbol of profile (trees, features, trees). Label and Arity constraints are as in CFT, and there is an equality predicate on both sorts. Here is an example of an F-constraint:

$$x\{f, g\} \land x[v]y \land v = f \land A(y)$$

Theorem 3. *Satisfiability of F-constraints is NP-complete.*

Proof. It is obvious that satisfiability of constraints can be checked in NP: If there is an atomic constraint $x[v]y$ and an arity constraint xF then we guess the value of v among the features in F; if there is no arity constraint for x then it is safe to take for v any new feature symbol.

To show that the problem is indeed NP-hard we encode 3-SAT in the satisfiability problem of F-constraints. Let $F = L_1 \land \ldots \land L_k$ be a propositional formula in conjunctive normal form, that is every L_i is a disjunction of three literals. We assume that the propositional variables are taken from the set $\{x_1, \ldots, x_n\}$. We construct an F-constraint ϕ_F such that F is satisfiable if and only if the constraints ϕ_F is satisfiable. The construction of F uses the variables $x_1^+, \ldots, x_n^+, x_1^-, \ldots, x_n^-, y_1, \ldots, y_k$. The formula ϕ_F is the conjunction of three subformulas $\phi_F^1 \land \phi_F^2 \land \phi_F^3$ that we will now describe:

1. For every i, either x_i^+ has root label $True$ and x_i^- has root label $False$, or the other way round.

$$\phi_F^1 = \bigwedge_{i=1}^{n} \exists z\Big(z\{p, n\} \land z[p]x_i^+ \land z[n]x_i^- \land \\ \exists i, v(z[i]v \land True(v)) \land \exists i, v(z[i]v \land False(v))\Big)$$

2. Encoding of the clause L_i:

$$\phi_F^2 = \bigwedge_{i=1}^{k} \Big(y_i\{1, 2, 3\} \land y_i[1]q_1 \land y_i[2]q_2 \land y_i[3]q_3\Big)$$

where L_i is $p_1 \lor p_2 \lor p_3$, and

$$q_j = \begin{cases} x_i^+ \text{ if } p_j = x_i \\ x_i^- \text{ if } p_j = \neg x_i \end{cases}$$

To give an example of this construction: If

$$F = \underbrace{(x_1 \lor \neg x_2 \lor \neg x_3)}_{L_1} \land \underbrace{(\neg x_2 \lor x_4 \lor \neg x_5)}_{L_2}$$

then we get

$$\phi_F^2 = y_1\{1, 2, 3\} \land y_1[1]x_1^+ \land y_1[2]x_2^- \land y_1[3]y_3^- \land \\ y_2\{1, 2, 3\} \land y_2[1]x_2^- \land y_2[2]x_4^+ \land y_2[3]x_5^-$$

3. All clauses L_i evaluate to true:

$$\phi_F^3 = \bigwedge_{i=1}^{k} \exists j, v(y_i[j]v \wedge True(v)) \qquad \Box$$

Does NP-completeness of the satisfiability problem imply that the constraint system is useless? There are two answers to this question:

1. It depends on the application. In applications like automatic theorem proving even NP might be an acceptable complexity since the search space is so huge that any means to restrict search by constraints is welcome. In programming, however, NP-complete constraint solving is much harder to accept. One can argue that the complexity of a program should stem from the algorithm, as expressed in the program, and not from the basic mechanisms that handles the data structures. Otherwise, it will be very difficult to obtain a useful complexity model. In other words, one should try to keep built-in constraint solving in programming simple (say linear, or at least polynomial of small degree), and have means to have more complex constraint solving programmed by the user.
2. The constraints might be hard to solve in the general case, however, they are easy to resolve once we have sufficient information about the values of the variables. In the case of F: Solving a constraint where all the feature variables are instantiated is nothing but CFT constraint solving, and this we know to do very efficiently. So one could attempt to modify our constraint-based formalism and allow constraints not to be solved immediately, that is to allow incomplete constraint solving methods.
   In the next section we will pursue this approach.

To know more: The constraint-system with first-class features has been introduced in [Tre93].

1.4 Programming with Incomplete Constraint Solvers

We will now refine the model of CLP presented in Section 1.2.2 to accommodate for incomplete constraint solving as discussed at the end of the preceding section. Furthermore, we would like to be more exact with the process of *constraint simplification* for reasons discussed after the proof of Theorem 1.

1.4.1 Constraint Systems (Second Version)

First we will refine our definition of a constraint system: A constraint system consists of

1. a language L of first-order logic,

2. a subset C of the set of first-order formulae of L , called the set of *constraints* of C, containing all atomic constraints, and closed under conjunction, existential quantification and renaming of bound variables,
3. a first-order structure A interpreting L,
4. a subset $S \subseteq C$ of *solved constraints* such that all constraints of S are satisfiable in A,
5. a computable partial *inference function*

   $$I\colon S \times C \rightsquigarrow \bot \cup (S \times 2^C)$$

   such that
   (a) If $I(c,d) = \bot$ then $c \wedge d$ is unsatisfiable in A
   (b) If $I(c,d) = (c', \{d_1, \dots, d_n\})$ then
      i. $A \models c \wedge d \Leftrightarrow c' \wedge d_1 \wedge \ldots \wedge d_n$
      ii. $A \models c' \Rightarrow c$

The solved constraints will play a prominent role in the calculus since they represent, in a given configuration, the knowledge about the values of the variables that can be freely used in the deductions. The role of the function I will be more clear with the following calculus.

1.4.2 A Refined CLP Calculus

Given a constraint system (L, C, A, S, I), a configuration is either FAIL or a pair $c \mid L$ where $c \in S$ (called the *constraint store*) and L is a list of atoms and constraints from C (the *agents*). The new calculus of constraint logic programming now comprises the rules depicted on Figure 1.6.

$$\text{(Unfold)} \quad \frac{c \mid L, A(\bar{x})}{c \mid L, K} \qquad A(\bar{x}) \leftarrow K \text{ is a program clause.}$$

$$\text{(Fail)} \quad \frac{c \mid L, d}{\textsc{Fail}} \qquad I(c,d) = \bot$$

$$\text{(Propagate)} \quad \frac{c \mid L, d}{c' \mid L, d_1, \dots, d_n} \quad I(c,d) = (c', \{d_1, \dots, d_n\})$$

Fig. 1.6. The rules of CLP with incomplete constraint solving.

Some remarks on this calculus:

- The second component of the inference function may be empty. In this case the application of rule (Propagate) makes the constraint disappear from the collection of agents and integrates it completely in the constraint store.

- It follows from the properties of the inference function of the constraint system that the constraint store increases monotonously during computation: If $c \mid L \rightarrow^* c' \mid L'$ then we have that $A \models c' \Rightarrow c$.
- The conjunction of constraints "$\wedge$" is not to be confused with the aggregation operator "," of agents. A single agent $c \wedge c'$ does not necessarily behave the same way as two separate agents c and c' since the inference function can see only one agent at a time.

Given a set of *initial agents* L, the computation starts on `True` $\mid L$. A computation *succeeds* if it terminates in a configuration of the form $c \mid \emptyset$, in this case c is the *answer constraint*, it fails if it terminates in FAIL, and it is *stuck* if it terminates in a configuration where no rule applies (remind that the inference function is partial!).

1.4.3 Some Constraint Systems with Partial Solvers

Feature Constraints. Our first example of a constraint system according to the new definition is CFT: The set of constraints and the structure are as in Section 1.3.2, and the the set of solved constraints is the set of constraints of the form $e \wedge c$ where c is a solved form and e is a set of equations that is eliminated in $e \wedge c$. The inference function is total in this case:

$$I_{CFT}(c,d) = \begin{cases} \bot & \text{if } c \wedge d \rightarrow^*_{CFT} \bot \\ c', \{\} & \text{if } c \wedge d \rightarrow^*_{CFT} c' \end{cases}$$

where $\rightarrow_{CFT}$ is the reduction relation introduced in Section 1.3.2. The fact that I_{CFT} is total and that, in case of non-failure, the second component is empty indicates that our constraint solver for CFT is complete.

The next example is the constraint system F that provided the motivation for incomplete constraint solving. We define the set of solved constraints as the set of constraints $e \wedge c$ where e is a set of equations eliminated in $e \wedge c$ (note that e may now give values to tree-variables *and* to feature-variables) and where c is a solved form of CFT (where we identify, slightly abusing notation, the F-constraint $x[f]y$ with the CFT-constraint $x[f]y$). The inference function is defined as follows:

$$I_F(c,d) = I_{CFT}(c,d) \qquad \text{if } d \text{ is a CFT constraint}$$

that is if d is one of $A(x)$, xF, $x[f]y$ with $f \in \mathcal{F}$ or an equation between tree-variables. The only other constraints that I_F can handle are equations for feature-variables:

$$I_F(c, v = t) = \begin{cases} \bot & \text{if } c\{v \mapsto t\} \text{ contains an equation between different feature symbols} \\ (v = t \wedge c\{v \mapsto t\}, \{\}) & \text{otherwise} \end{cases}$$

$$I_F(c, x[v]y) = \begin{cases} (c, \{x[f]y\}) & \text{if } c \text{ contains an equation } v = f \\ \text{undefined} & \text{otherwise} \end{cases}$$

where v is a feature variable. Note that feature variables may occur in c only in the equational part, hence the condition is very easy to decide.

The crucial point here is that $I(c, x[v]y)$ is not defined if c contains no definition for v. In this case we say that the constraint $x[v]y$ is *suspended* since it is "waiting" for its feature variable to get defined by some other agent.

Nonlinear Equations. Our last example is non-linear equations between integers. Constraints are conjunctions of one of the three forms $x = n$, $x = y + z$ and $x = y * z$ where n is an integer constant and x, y, z are variables. A conjunction containing only the first two forms of constraints is called *linear.* A constraint is solved if it is a linear system of the form

$$x_1 = t_1 \wedge \ldots \wedge x_n = t_n$$

where x_i is different from x_j for $i \neq j$, and x_i does not occur in $t_i, \ldots, t_n$. This is the well-known Gaussian triangular form. The equally well-known Gaussian elimination algorithm transforms any linear constraint c either into $\perp$ or into a solved form c', in this case we write $c \rightarrow^*_{lin} \perp$, resp. $c \rightarrow^*_{lin} c'$. Satisfiability of arbitrary constraints (involving products), however, is undecidable, as it has been shown by Matijacevič. We obtain an inference function for this constraint systems by suspending any non-linear equation $x = y * z$ until at least one of the two variables y and z is instantiated:

$$I_{NonLin}(c, d) = \begin{cases} \perp & \text{if } d \text{ linear, } c \wedge d \rightarrow^*_{lin} \perp \\ (c', \{\}) & \text{if } d \text{ linear, } c \wedge d \rightarrow^*_{lin} c' \end{cases}$$

$$I_{NonLin}(c, x = y * z) = \begin{cases} (c, \{x = n * z\}) & \text{if } c \text{ contains } y = n \\ (c, \{x = n * y\}) & \text{if } c \text{ contains } z = n \\ \text{undefined} & \text{otherwise} \end{cases}$$

The constraint $x = n * z$, for n an integer constant, can easily be expressed by linear constraints.

Is it possible to find a stronger inference function that can extract more linear constraints from a non-linear equation? The answer is yes, in fact there is an infinite growing chain of inference functions, since we can, for any $k = 1, 2, 3, \ldots$, add

$$I_{NonLin}(c, y = x * x) = c, \{y = x * x, y \geq 2kx - k^2\}$$

The inequation $y \geq 2kx - k^2$ can be seen as a linear constraint since it can be rewritten as

$$\exists z, z_1, z_2, z' \left(z = y + z_1 \wedge z_1 = k^2 \wedge z = 2kx + z' \right)$$

where $z = 2kx + z'$ itself is a linear equation as we have seen above. The reader verifies easily the soundness of these inference functions.

Hence, the approach presented in this section leaves two open questions:

- What is a appropriate formalism to define the inference function of a constraint system?
- What inference function should we choose when there is no way to extract the "maximal" information from some agent, as in the case $y = x * x$ seen above?

In the next section we will present a programming formalism that allows the user, to some extent, to program the constraint system itself in the programming language. This also yields one possible answer to the second question since it gives the programmer the means to program constraint propagation (which corresponds to the inference function of this section) to be as strong as he needs it for his application.

To know more: The material of this section owes much to the survey [JL87]. See also [CDJK99] for a methodological view of constraint solving algorithms.

1.5 Committed Choice: A More Realistic Approach

The CLP calculus presented in Section 1.2.2 is an non-deterministic calculus. To define the semantics of of a CLP program it is necessary to talk about *all* possible deductions in the calculus. This non-determinism is not reflected in the calculus itself. For a concrete implementation of CLP this means that the machine must be able to explore systematically all computation paths, this is usually implemented with some kind of backtracking.

In this section we want to present another constraint-based programming formalism that

- represents indeterminism *explicitly* in the calculus, by using disjunctions, and
- allows both deterministic and non-deterministic computation steps.

1.5.1 A Calculus for Constraint Programming

Constraint Programming (CP) in its pure form follows a paradigm that is fundamentally different from the philosophy of CLP as seen in Section 1.2.2. CP is based on *deterministic* computation while the latter follows the tradition of *declarative programming*, that is based on *non-deterministic* computation that can be realized using an additional enumeration procedure. In the following we give a simple calculus that reunites CLP and pure CP.

A *program* is a sequence of procedure definitions

$$\begin{aligned} P_1(x_1, \ldots, x_{n_1}) &\Leftrightarrow A_1 \\ &\vdots \\ P_m(x_1, \ldots, x_{n_m}) &\Leftrightarrow A_m \end{aligned}$$

where the A_i are *agent expressions* to be defined below. The variables x_i are considered *universally bound*. We will rename these bound variables whenever necessary. The set of agent expressions is defined by the following grammar:

$A \rightarrow P(x_1, \dots, x_n)$	*procedure call*
$\quad \rightarrow c$	*constraint*
$\quad \rightarrow A \wedge A$	*conjunction*
$\quad \rightarrow \exists \bar{x}\, A$	*scoped expression*
$\quad \rightarrow$ `if` D `fi`	*conditional*
$\quad \rightarrow$ `or` D `ro`	*disjunction*
$D \rightarrow$	*empty clause list*
$\quad \rightarrow C \mid D$	*non-empty clause list*
$C \rightarrow \exists \bar{x} c$ `then` A	*clause*

In a clause list we usually omit the last $|$. In a clause $\exists \bar{x} c$ `then` A, the scope of the existential quantifier extends over the constraint c (the *guard* of the clause) and the agent A (the *protected part* of the clause).

An important point of our version of the CP-calculus is that we adopt another solution than in Section 1.4 to cope with incomplete constraint solvers. Here, we assume that we dispose of a constraint system with a *complete* constraint solving algorithm, that is a constraint system in the sense of Section 1.2.3. Data descriptions that are not part of the constraint system in case we don't have an (efficient) constraint solver for them are now expressed by agent expressions called *constraint propagators*. Hence, what we treated in Section 1.4 as a constraint system with an incomplete inference function is now split up into one part which makes up the constraint system, and another part that is to be *programmed* (we will see soon how this can be accomplished).

A CP-configuration is either FAIL or a pair $c \mid As$ where c is a constraint and As is a set of agent expressions. The reduction rules of the calculus are given in Figure 1.7. In the Local-rule it is understood that $\bar{y}$ and $\bar{x}$ are both vectors of pairwise different variables and are of the same length. The notation $A\{\bar{x} \mapsto \bar{y}\}$ denotes the agent expression obtained by replacing in A every free occurrence of a free variable from $\bar{x}$ by its counterpart from $\bar{y}$.

A computation is called *successful* when it terminates in a configuration with an empty set of agents, it *fails* when it terminates in FAIL, and it is *stuck* when it terminates in a configuration where no further rule application is possible and where the agent space is non-empty.

The disjunctive and conditional agents are at the heart of our CP calculus. Roughly, disjunctive agents express a *don't-know* choice while conditional agents express a *don't-care* choice. More precisely:

There are three ways how a disjunctive agent can evolve:

1. A clause is dropped by the rule DisDrop if its guard is not consistent with the constraint store.

$$\text{(Unfold)} \quad \frac{c \mid P(x_1,\ldots,x_n),\psi}{c \mid A,\psi} \quad \begin{array}{l} P(x_1,\ldots,x_n) \;\Leftrightarrow\; A \\ \text{is a program clause} \end{array}$$

$$\text{(Failure)} \quad \frac{c \mid c',\psi}{\textsc{Fail}} \quad c \wedge c' \text{ unsatisfiable}$$

$$\text{(Simplify)} \quad \frac{c \mid c',\psi}{c'' \mid \psi} \quad c \wedge c' \text{ satisfiable and } c \wedge c' \Leftrightarrow c''$$

$$\text{(Unpack)} \quad \frac{c \mid A_1 \wedge A_2,\psi}{c \mid A_1, A_2,\psi}$$

$$\text{(Local)} \quad \frac{c \mid (\exists \bar{x}\, A),\psi}{c \mid A\{\bar{x} \mapsto \bar{y}\},\psi} \quad \bar{y} \text{ fresh variables}$$

$$\text{(DisDrop)} \quad \frac{c \mid \;\texttt{or}\; D \mid \exists \bar{x}\, c' \;\texttt{then}\; A \mid D' \;\texttt{ro}\;,\psi}{c \mid \;\texttt{or}\; D \mid D' \;\texttt{ro}\;,\psi} \quad c \wedge \exists \bar{x} c' \text{ unsatisfiable}$$

$$\text{(DisSingle)} \quad \frac{c \mid \;\texttt{or}\; \exists \bar{x}\, c \;\texttt{then}\; A \;\texttt{ro}\;,\psi}{c \mid (\exists \bar{x}\, (c \wedge A)),\psi}$$

$$\text{(DisEmpty)} \quad \frac{c \mid \;\texttt{or ro}}{\textsc{Fail}}$$

$$\text{(CondDrop)} \quad \frac{c \mid \;\texttt{if}\; D \mid \exists \bar{x}\, c' \;\texttt{then}\; A \mid D' \;\texttt{fi}\;,\psi}{c \mid \;\texttt{if}\; D \mid D' \;\texttt{fi}\;,\psi} \quad c \wedge \exists \bar{x} c' \text{ unsatisfiable}$$

$$\text{(CondFire)} \quad \frac{c \mid \;\texttt{if}\; D \mid \exists \bar{x}\, c' \;\texttt{then}\; A \mid D' \;\texttt{fi}\;,\psi}{c \mid (\exists \bar{x}(c' \wedge A)),\psi} \quad c \models \exists \bar{c}',$$

$$\text{(CondEmpty)} \quad \frac{c \mid \;\texttt{if fi}\;,\psi}{c \mid \psi}$$

Fig. 1.7. Computation rules for CP.

2. If the set of clauses is empty then the agent fails. This is consistent with the logical view of an empty disjunction being equivalent to $false$.
3. If there is only one clause in the agent then the disjunction disappears and sets free the guard together with its protected part. Application of the Local-rule will eliminate the existential quantifier, allowing for further consideration of the agent expressions.

Note that, in presence of several disjunctive clauses, we can not commit to a clause when its guard is entailed by the constraint of the current configuration (since that would amount to a *don't-care* choice whereas we want disjunctive agents to model *don't-know* choices).

This treatment of disjunction is very similar to the *Andorra* rule proposed by D. H. Warren for logic programming to allow determinate reduction steps (the names comes from And-Or).

The conditional agent behaves differently:

1. A conditional *fires* its clause if its guard is entailed by the constraint store. Entailment is necessary for firing the clause even when there is only one clause left in the conditional expression.
2. A conditional clause is dropped when its guard is inconsistent with the constraint of the current configuration.
3. The conditional vanishes when no clause is left; as an alternative to our definition, one could allow a conditional expression to have an *otherwise* expression which fires when all regular clauses are eliminated.

A dramatic consequence of the rules for the conditional expression is that, a priori, logical semantics is lost. For a given constraint store it might well be the case that the guards of several clauses are entailed at a time. In this case, our calculus makes a non-deterministic *don't-care* choice of the clause to be fired. It is hence the responsibility of the programmer to make the guards of the conditional clauses strong enough such that a *don't care* commitment to a clause causes no harm. This choice is fundamentally different from the choice seen in Section 1.2.2 when selecting a clause since now all non-determinism is explicit in the expressions. There are, however, ways to ensure by a careful analysis that a conditional agent still behaves in a logically meaningful way (see, for instance, [AFM96]).

The constraint store increases monotonically during the course of a computation, that is if $c \mid \psi \to^* c' \mid \psi'$ then $c \models c'$ holds. The consequence is that if in a certain configuration a conditional agent can fire one of its clauses, then he will still be able to fire this clause after further progress of the other agents (provided that the clause has not disappeared due to inconsistency). This is one key to consider the calculus as a calculus of *concurrent computations*.

The agents co-exist independently from each other in the agent space, the only way to have some "communication" among agents is by having them impose constraints on the constraint store. Formally, if $c \mid A, \psi \to c \mid \phi, \psi$, then for all sets ψ' of agent expressions we have as well that $c \mid A, \psi' \to c \mid \phi, \psi'$. This is another key to concurrency.

Let us now demonstrate the difference between disjunctions and conditionals with an example: the concatenation of lists. First we define a concatenation predicate as a disjunction (the binary constructor symbol ": " is here written in infix notation):

```
conc(X,Y,Z) ⇔
   or
       X=nil ∧ Y=Z
    |  ∃ H,X',Z' X=H:X' ∧ Z=H:Z' then conc(X',Y,Z')
```

```
ro
```

Here is a query leading to a successful computation:

$$X = a\colon \mathit{nil} \wedge Y = b\colon c\colon \mathit{nil} \ \mid \ \mathit{conc}(X, Y, Z)$$
$$X = a\colon \mathit{nil} \wedge Y = b\colon c\colon \mathit{nil} \ \mid \ \texttt{or}\ X = \mathit{nil} \wedge Y = Z \mid \exists H, X', Z' \ldots \ \texttt{ro}$$
$$X = a\colon \mathit{nil} \wedge Y = b\colon c\colon \mathit{nil} \ \mid$$
$$\quad \texttt{or}\ \exists\, H, X', Z'\ X = H\colon X' \wedge Z = H\colon Z'\ \texttt{then}\ \mathit{conc}(X', Y, Z')\ \texttt{ro}$$
$$X = a\colon \mathit{nil} \wedge Y = b\colon c\colon \mathit{nil} \ \mid$$
$$\quad \exists H, X', Z'\ (X = H\colon X' \wedge Z = H\colon Z' \wedge \mathit{conc}(X', Y, Z'))$$
$$X = a\colon \mathit{nil} \wedge Y = b\colon c\colon \mathit{nil} \ \mid$$
$$\quad X = H\colon X' \wedge Z = H\colon Z', \mathit{conc}(X', Y, Z')$$
$$X = a\colon \mathit{nil} \wedge Y = b\colon c\colon \mathit{nil} \wedge H = a \wedge X' = \mathit{nil} \wedge Z = a\colon Z' \ \mid$$
$$\quad \mathit{conc}(X', Y, Z')$$
$$\ldots$$
$$X = a\colon \mathit{nil} \wedge Y = b\colon c\colon \mathit{nil} \wedge H = a \wedge X' = \mathit{nil} \wedge Z = a\colon b\colon c\colon \mathit{nil} \wedge$$
$$\quad Z' = b\colon c\colon \mathit{nil} \ \mid \ \emptyset$$

The following computation, however, is not successful since it gets stuck with a disjunction that can not reduce any further:

$$Y = \mathit{nil} \ \mid \ \mathit{conc}(X, Y, Z)$$
$$Y = \mathit{nil} \ \mid \ \texttt{or}\ X = \mathit{nil} \wedge Y = Z \mid \exists H, X', Z'\ X = H\colon X' \wedge \ldots \ \texttt{ro}$$

This situation can always occur when we use disjunctive agents. One solution is to consider a *disjunction of configurations* and to have an additional enumeration rule in the calculus:

$$\text{(Enumerate)}\ \frac{c \ \mid \ \texttt{or}\ C \mid D\ \texttt{ro}\ , \psi}{\Big(c \ \mid \ C, \psi\Big) \vee \Big(c \ \mid \ \texttt{or}\ D\ \texttt{ro}\ , \psi\Big)}$$

This rule should only be used when no other reduction is possible. In some cases, in particular for constraint satisfaction problems as seen in Section 1.2.5, such an *enumeration rule* is necessary. In other cases, however, it is possible to overcome the weakness of disjunctions by using conditional expressions, as in our example. Here is a version of the concatenation predicate using a conditional agent expression:

```
conc(X,Y,Z) ⇔
  if X=nil then Y=Z
   | Y=nil then X=Z
   | Z=nil then X=nil ∧ Y=nil
   | ∃ H,X',Z' X=H:X'  then  Z=H:Z' ∧ conc(X',Y,Z')
  fi
```

With this conditional version of conc, both example queries seen above lead to successful computations:

$$
\begin{aligned}
&X = a\colon nil \wedge Y = b\colon c\colon nil \mid conc(X, Y, Z)\\
&X = a\colon nil \wedge Y = b\colon c\colon nil \mid\\
&\qquad \texttt{if } \ldots \mid \exists H, X', Z'\ X = H\colon X' \texttt{ then } Z = H\colon Z' \wedge\\
&\qquad\qquad\qquad\qquad\qquad\qquad conc(X', Y, Z') \texttt{ fi}\\
&X = a\colon nil \wedge Y = b\colon c\colon nil \mid\\
&\qquad \exists H, X', Z'(X = H\colon X' \wedge Z = H\colon Z' \wedge conc(X', Y, Z'))\\
&X = a\colon nil \wedge Y = b\colon c\colon nil \mid\\
&\qquad X = H\colon X' \wedge Z = H\colon Z', conc(X', Y, Z')\\
&X = a\colon nil \wedge Y = b\colon c\colon nil \wedge Z = a\colon Z' \wedge H = a \wedge X' = nil \mid\\
&\qquad conc(X', Y, Z')\\
&X = a\colon nil \wedge Y = b\colon c\colon nil \wedge Z = a\colon Z' \wedge H = a \wedge X' = nil \mid\\
&\qquad \texttt{if } X' = nil \texttt{ then } Y = Z' \texttt{ fi}\\
&X = a\colon nil \wedge Y = b\colon c\colon nil \wedge Z = a\colon b\colon c\colon nil \wedge H = a \wedge\\
&\qquad X' = nil \wedge Z' = b\colon c\colon nil \mid \emptyset
\end{aligned}
$$

$$
\begin{aligned}
&Y = nil \mid conc(X, Y, Z)\\
&Y = nil \mid \texttt{ if } \ldots \mid Y = nil \texttt{ then } X = Z \mid \ldots \texttt{ fi}\\
&Y = nil \mid Y = nil, X = Z\\
&Y = nil \wedge X = Z \mid \emptyset
\end{aligned}
$$

1.5.2 Constraint Propagators for Boolean Constraints

Boolean constraints are an example where one constraint propagator makes use of an other constraint propagator. We give here the propagators for exclusive-or and inequality:

```
XOr(x,y,z) ⇔
if x=T then NEq (y,z) | y=T then NEq (x,z) | z=T then NEq(y,z)
 | x=F then y=z | y=F then x=z | z=F then x=y
 | x=y then z=F | x=z then y=F | y=z then x=F
fi

NEq(x,y) ⇔
if x=F then y=T | x=T then y=F | y=T then x=F | y=F then x=T
 | x=y then FAIL
fi
```

1.5.3 Constraint Propagators for Feature Constraints

The constraint propagator for the subtree constraint can be realized by testing a variable to be determined:

```
Subtree(X,V,Y) ⇔
  if Det(V) then X[V]Y
  fi
```

In case of the feature constraint system considered in this section, a conditional clause guarded by $Det(x)$ can fire if the x gets bound to some value. More generally, one defines that a constraint ψ determines a variable x if

$$\forall \bar{y} \exists^{\leq 1} x\, \psi \qquad \text{where } \bar{y} = V(\psi) - \{x\}$$

holds in the interpretation of the constraint system. In this formula, $\exists^{\leq 1} x\, \psi$ means *there is at most one value of x that satisfies ψ*, this formula can be seen as an abbreviation for

$$\forall x, x' (\psi \wedge \psi\{x \mapsto x'\} \Rightarrow x = x')$$

In case of the CFT constraint system this notion of a determined variable coincides exactly with the definition given in Section 1.3.2.

An important remark is that the expression $Det(x)$ is not a constraint since this expression does not have a meaning in itself, it rather is an observation *about the constraint store*. Strictly speaking, this is a violation of the formation rules of our calculus, nevertheless, such a special guard $Det(x)$ is easily integrated in the implementation of a constraint programming system.

More generally, we can see *Det* as a *reflection predicate* that can observe the state of the constraint store, thus raising in some sense the constraints to first-class status. Reflection of the constraint store raises the question of how to define proper semantics, however, it seems to be unavoidable if one wants to express useful constraint propagators in the programming calculus.

1.5.4 Finite Domain Constraints

We have already briefly encountered finite domain constraints in Section 1.2.5. We now give a complete definition.

An *interval* is a set of natural numbers of the form $\{n, n+1, \ldots, m\}$ where $m > n$. A *finite domain* is a finite union of intervals.

The language of finite domain constraints contains, for every finite domain D, a unary predicate $x \in D$ and an equality predicate $=$. Domains are written by using the notion $[n, m]$ for intervals and the symbol $\cup$ for union, as in $x \in [2, 4] \cup [6, 8]$, furthermore we will use a notation like $\{1, 17, 42\}$ for sparse sets. Constraints are (probably existentially quantified) conjunctions of atomic constraints, and the interpretation is the obvious one. An algorithm for checking satisfiability of constraints is straightforward (see Figure 1.8).

$$\text{(Intersect)}\quad \frac{x \in D_1 \wedge x \in D_2 \wedge \psi}{x \in D_1 \cap D_2 \wedge \psi} \text{ if } D_1 \cap D_2 \neq \emptyset$$

$$\text{(Disjoint)}\quad \frac{x \in D_1 \wedge x \in D_2 \wedge \psi}{\bot} \text{ if } D_1 \cap D_2 = \emptyset$$

$$\text{(Eq)}\quad \frac{x = y \wedge \phi}{x = y \wedge \phi\{x \mapsto y\}} \quad x \text{ occurs in } \phi,\ x \neq y$$

$$\text{(Triv)}\quad \frac{x = x \wedge \phi}{\phi}$$

Fig. 1.8. Rules for Finite Domain Constraints

What is missing to make finite domain constraints useful is constraints that express some non-trivial relation between variables, as for instance $x \neq y, x = y + z, x = y * z, \ldots$. Unfortunately, deciding satisfiability of finite domain constraints together with addition $x = y + z$ is NP-complete even when only domains of the form $[0, 1]$ are involved, as the reader may easily prove as an exercise (Hint: look at the proof of Theorem 3). Hence, we have another case where we need to program constraint propagators. The question is, however, how strong a constraint propagation we want to program. We will look at two approaches: domain propagation and interval propagation. Both propagation strategies are defined by reference to a logical reading. To keep things simple we will just look at one propagator $Plus(x, y, z)$, having a logical reading $x = y + z$.

The propagation methods that we present here always consider one agent expression $Plus(x, y, z)$ at a time, according to the principles which underly our programming model (see the remarks on the calculus above). In the literature on finite domain constraints this is called *local consistency* checking. In many application, however, it is necessary to check several agents simultaneously, that is to perform *global consistency* checking. This is often expressed by agent expressions acting on an arbitrary number of variables, for instance $AllDifferent(x_1, \ldots, x_n)$ which expresses, for arbitrary n, that all variables $x_1, \ldots, x_n$ take different values from their respective domains. This could of course be expressed as $n * (n - 1)$ binary inequalities, however, treating all these inequalities at once in an *AllDifferent* expression is much more efficient.

Domain Propagation. The definition of *domain propagation* is that (with the example of *Plus*):

$$\begin{aligned} & x \in D_x \wedge y \in D_y \wedge z \in D_z \ \mid \ Plus(x, y, z) \\ \rightarrow\ & x \in D'_x \wedge y \in D'_y \wedge z \in D'_z \ \mid \ Plus(x, y, z) \end{aligned}$$

such that the tuple (D'_x, D'_y, D'_z) is maximal (with respect to component-wise set inclusion) with

1. $D'_v \subseteq D_v$ for all variables v, and
2. for every $k \in D'_x$ there are $l \in D'_y$, $m \in D'_z$ such that $k = l + m$, and analogously for the other variables.

If this definition makes one of the domains empty then propagation of course yields the configuration Fail. For example,

$$\begin{aligned} & x \in [2, 12] \land y \in [2, 12] \land z \in \{12\} \mid \mathit{Mult}(x, y, z) \\ \to\ & x \in \{2, 3, 4, 6\} \land y \in \{2, 3, 4, 6\} \land z \in \{12\} \mid \mathit{Mult}(x, y, z) \end{aligned}$$

The situation is depicted in the left part of Figure 1.9. Note that in the diagram, every row and every column contains at least one $\times$-mark, due to the second condition of the above definition of domain propagation.

Furthermore it should be observed that with finite domain constraints (independent of the particular propagation method) the point-wise dependencies between variables are lost. This kind of abstraction is very similar to the set constraints considered in constraint-based program analysis (see Section 1.2.7).

Interval Propagation. In comparison with domain propagation, *interval propagation* propagates less constraints (that is, leads in general to a weaker constraint store) but is computationally less expensive. For the sake of simplicity let us assume that all finite domain constraints in the store are interval constraints $x \in [n, m]$, it is obvious how to generalize to arbitrary finite domains.

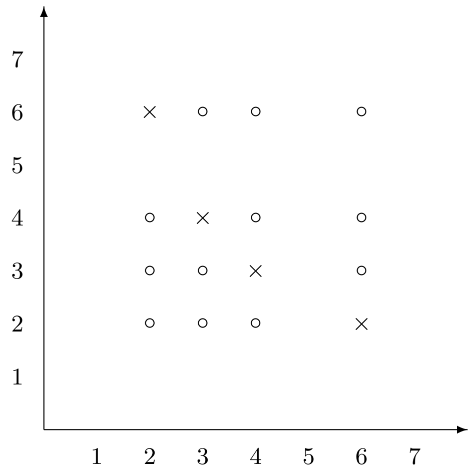
$$\begin{aligned} & x \in [x_l, x_u] \land y \in [y_l, y_u] \land z \in [z_l, z_u] \mid \mathit{Plus}(x, y, z) \\ \to\ & x \in [x'_l, x'_u] \land y \in [y'_l, y'_u] \land z \in [z'_l, z'_u] \mid \mathit{Plus}(x, y, z) \end{aligned}$$

where for all variables v the values v'_l are maximal and the values v'_u are minimal such that

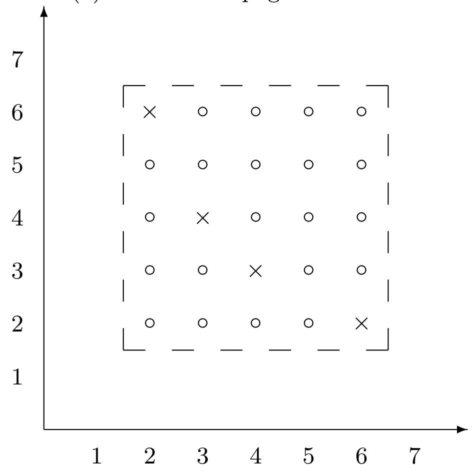
- for all variables v we have that $v_l \leq v'_l$ and $v'_u \leq v_u$, and
- There are $y_1, y_2 \in [y'_l, y'_u]$, $z_1, z_2 \in [z'_l, z'_u]$ such that $x'_l = y_1 + z_1$ and $x'_u = y_2 + z_2$, and analogously for the other variables.

If this definition makes one of the domains empty then propagation of course yields the configuration Fail. Alternatively, one could say that interval propagation yields the smallest intervals that subsume the sets found by domain propagation. We demonstrate this behavior with the same example that we used to demonstrate domain propagation:

$$\begin{aligned} & x \in [2, 12] \land y \in [2, 12] \land z \in \{12\} \mid \mathit{Mult}(x, y, z) \\ \to\ & x \in [2, 6] \land y \in [2, 6] \land z \in \{12\} \mid \mathit{Mult}(x, y, z) \end{aligned}$$



(a) Domain Propagation



(b) Interval Propagation

Fig. 1.9. Constraint store found by different propagation methods. Only the values for x and y are shown. Pairs that are actual solutions are marked by $\times$.

The situation is depicted in the right part of Figure 1.9. Note that the marked area is a rectangle, and that the rows and columns *along the border of the rectangle* each contain at least one ×-mark. Rows and columns in the interior of the rectangle, however, may not contain ×-marks.

1.5.5 Constraint Propagators for Interval Propagation

In order to express interval propagation in our calculus for constraint programming we need some way to know the actual bounds of a variable, that is to *reflect* the current state of the constraint store. For this purpose we will allow, for any variable x, in the guard of a conditional additional identifiers $x\uparrow$ and $x\downarrow$, denoting the current lower and upper bound of the domain of x. This is very similar to the predicate *Det* that we employed in Section 1.5.3, note that an expression using the notation $x\uparrow$ or $x\downarrow$ is not a constraint in the proper sense since it makes a statement about the current constraint store itself instead of about the value of the variable x.

Here is a possible definition for the propagator *Plus*:

```
Plus(x,y,z) ⇔
if x↑ > y↑ + z↑ then x ∈ [x↓,y↑+z↑] ∧ Plus(x,y,z)
 | x↓ < y↓ + z↓ then x ∈ [y↓+z↓,x↑] ∧ Plus(x,y,z)
 | y↑ > x↑ - z↓ then y ∈ [y↓,x↑-z↓] ∧ Plus(x,y,z)
 | y↓ < x↓ - z↑ then y ∈ [x↓-z↑,y↑] ∧ Plus(x,y,z)
 | z↑ > x↑ - y↓ then z ∈ [z↓,x↑-y↓] ∧ Plus(x,y,z)
 | z↓ < x↓ - y↑ then z ∈ [x↓-y↑,z↑] ∧ Plus(x,y,z)
 | x↓ = x↑ ∧ y↓ = y↑ then z = x↓ - y↓
 | x↓ = x↑ ∧ z↓ = z↑ then y = x↓ - z↑
 | y↓ = y↑ ∧ z↓ = z↑ then x = y↓ + z↓
fi
```

1.5.6 Concluding Remarks

In this section we have tried to demonstrate how constraint propagation can be expressed in a calculus based on constraint programming. However, there is no final answer to the question how to integrate constraints and programming. We have seen that already for quite simple constraint systems such as feature constraints with features as first-class values, or finite domain constraints with arithmetical relations between variables, we needed reflection of the constraint store as an extension of the programming calculus.

Programming constraint propagation in the programming calculus itself is not the only way to realize constraint propagators, constraint propagation might well be expressed in a dedicated formalism. Efficiency consideration will often lead to hardwire constraint propagation into a programming environment. Nevertheless, a programmer needs to know exactly the constraint propagation mechanism; a high-level description in the programming calculus might hence be a good way to specify the mechanism.

Last but not least, it is not obvious how constraint-based programming integrates with other programming paradigms as functional programming, object-oriented programming, or how constraint-based calculations can be distributed over different machines. The chapter *Functional and Constraint Logic Programming* will address programming calculi based on constraints.

To know more: Constraint calculi with committed choice go back to the family of cc-languages introduced by Saraswat, see for instance [SR91]. The material of this section reuses many ideas of the Oz project, the actual Oz-calculus, however, is much more expressive than the primitive calculus presented here, see [Smo95] and `http://www.mozart-oz.org`.

A good introduction to finite domain constraints is [Hen89]. More about real-life solutions by finite domain constraints will be told in the chapter *Building Industrial Applications with Constraint Programming.*

References

[AFM96] Slim Abdennadher, Thom Frühwirth, and Holger Meuss. On confluence of constraint handling rules. In Eugene Freuder, editor, *Principles and Practice of Constraint Programming*, volume 1118 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1996.

[Apt90] Krzysztof R. Apt. Introduction to logic programming. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B*, chapter 10, pages 493–574. Elsevier Science Publishers, 1990.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BVW94] Orna Bernholtz, Moshe Y. Vardi, and Pierre Wolper. An automata-theoretic approach to branching-time model checking. In David L. Dill, editor, *Computer Aided Verification*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155, Stanford, CA, USA, June 1994. Springer-Verlag.

[CDG$^+$99] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata. Techniques and Applications, April 1999. Available at `http://www.grappa.univ-lille3.fr/tata/`.

[CDJK99] Hubert Comon, Mehmet Dincbas, Jean-Pierre Jouannaud, and Claude Kirchner. A methodological view of constraint solving. *Constraints*, 4(4):337–361, December 1999.

[CJ99a] Hubert Comon and Yan Jurski. Counter automata, fixpoints and additive theories. December 1999.

[CJ99b] Hubert Comon and Yan Jurski. Timed automata and the theory of real numbers. In Jos C. M. Baeten and Sjouke Mauw, editors, *Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 242–257, Eindhoven, The Netherlands, August 1999. Springer-Verlag.

[Coh90] Jacques Cohen. Constraint logic programming languages. *Communications of the ACM*, 33(7):52–68, July 1990.

[CT94] Hubert Comon and Ralf Treinen. Ordering constraints on trees. In Sophie Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 1–14, Edinburgh, Scotland, 1994. Springer-Verlag. (Invited Lecture).

[DP00] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *Software Tools for Technology Transfer*, 2000. To appear.

[DSH90] Mehmet Dincbas, Helmut Simonis, and Pascal Van Hentenryck. Solving large combinatorial problems in logic programming. *Journal of Logic Programming*, 8:75–93, 1990.

[FO97] Laurent Fribourg and Hans Olsén. Proving safety properties of infinite state systems by compilation into presburger arithmetic. In Antoni W. Mazurkiewicz and Józef Winkowski, editors, *Concurrency Theory*, volume 1243 of *Lecture Notes in Computer Science*, pages 213–227, Warsaw, Poland, July 1997. Springer-Verlag.

[FP94] Laurent Fribourg and Morcos Veloso Peixoto. Automates concurrents à contraintes. *Technique et Science Informatiques*, 13(6), 1994.

[Hen89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[HJ90] Nevin Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17th ACM Conference on Principles of Programming Languages*, pages 197–209, San Francisco, CA, January 1990.

[JK91] Jean-Pierre Jouannaud and Claude Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In Jean-Louis Lassez and Gordon Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson.* MIT-Press, 1991.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Conference on Principles of Programming Languages*, pages 111–119, Munich, Germany, January 1987. ACM.

[Kel93] Bill Keller. *Feature Logics, Infinitary Descriptions and Grammar.* CSLI Lecture Notes No. 44. Center for the Study of Language and Information, 1993.

[Nie99] Robert Nieuwenhuis. Rewrite-based deduction and symbolic constraints. In Harald Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 302–313, Trento, Italy, July 1999. Springer-Verlag.

[Pod00] Andreas Podelski. Model checking as constraint solving. In Jens Palsberg, editor, *Static Analysis Symposium*, Santa Barbara, CA, USA, June/July 2000.

[PP97] Leszek Pacholski and Andreas Podelski. Set constraints: A pearl in research on constraints. In Gert Smolka, editor, *Principles and Practice of Constraint Programming*, volume 1330 of *Lecture Notes in Computer Science*, pages 549–561, Linz, Austria, October 1997. Springer-Verlag.

[Smo92] Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Current Trends in Computer Science*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, Heidelberg, New York, 1995.

[SR91] Vijay Saraswat and Martin Rinard. Semantic foundations of concurrent constraint programming. In *Proceedings of the 18th Symposium on Principles of Programming Languages*, pages 333–351, Orlando, FL, January 1991. ACM.

[ST94] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[Tre93] Ralf Treinen. Feature constraints with first-class features. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Proc. 18th Mathematical Foundations of Computer Science*, volume 711 of *Lecture Notes in Computer Science*, pages 734–743, Gdańsk, Poland, 30 August–3 September 1993. Springer-Verlag.

2 Constraint Solving on Terms

Hubert Comon[1] and Claude Kirchner[2]*

[1] LSV, Ecole Normale Supérieure de Cachan, Cachan, France
[2] LORIA and INRIA, Villers-lès-Nancy, France

2.1 Introduction

In this chapter, we focus on constraint solving on terms, also called Herbrand constraints in the introductory chapter, and we follow the main concepts introduced in that chapter.

The most popular constraint system on terms is probably *unification problems* which consist of (possibly existentially quantified) conjunctions of equations. Such formulas have to be interpreted in the set of terms or one of its quotients (see e.g. [JK91, BS99] for surveys on unification). Other constraint systems on terms proved to be very useful, for instance introducing negation as in PROLOG II [Col82] leads to study *disunification* (see [Com91] for a survey). *Ordering constraints* on terms have been introduced to express ordered strategies in automated deduction (see the introductory chapter and the chapter on "contraints and theorem proving", or the survey [CT94]). Membership constraints may also express typing information and *set constraints* deserved a lot of works recently (see e.g. [PP97] for a survey), ...

We will try here to describe very briefly most of the constraint systems on terms, but our main focus will be on the method we are using. We follow the distinction of [CDJK99] between *syntactic methods* and *semantic methods*. Syntactic methods consist simply of rewriting the constraint into an equivalent one until a solved form is reached, hence rewriting techniques are relevant there (and we refer the reader to [DJ90a, BN98, KK99] for basics on rewriting). Semantic methods are based on another representation of the set of solutions. In the case of constraints on terms, such a representation could be fruitfully given by means of automata.

Therefore, our chapter is divided into two parts: the first part will describe some constraint systems and their corresponding syntactic methods. In particular, we consider unification problems in section 2.3, dis-unification problems in section 2.4, ordering constraints in section 2.5 and matching constraints in section 2.6. The ELAN system [BKK$^+$98], designed to mechanize the use of inference rules, easily implements these ideas: its computation engine rewrites formulas under the control of strategies.

The second part of the lecture is dedicated to automata techniques. The relationship between logic and automata goes back to Büchi, Elgot and

* Both authors supported by the ESPRIT working group CCL-II, ref. WG # 22457.

Church in the early sixties [Büc60, Elg61, Chu62]. The basic idea is to associate with each atomic formula a device (an automaton) which accepts all the models of the formula. Then, using some closure properties of the recognized languages, we can build an automaton accepting all the models of an arbitrary given formula. This is also the basis of optimal decision techniques (resp. model-checking techniques) for propositional temporal logic (see e.g. [Var96, BVW94]). In this lecture, we illustrate the method with three main examples, using three different notions of automata: Presburger arithmetic and classical word automata in section 2.8, typing constraints and tree automata in section 2.9, set constraints and tree set automata in section 2.10. This second part heavily relies on the book [CDG$^+$97]. There was also a survey of these techniques at the CCL conference in 1994 [Dau94].

2.2 The Principle of Syntactic Methods

As explained in the introductory chapter, a constraint is simply a formula, together with its interpretation domain. The *syntactic methods* rely on an axiomatization of this domain, together with a strategy for the use of these axioms.

More precisely, a *constraint solving* method is defined by a (recursive) set of rewrite rules. Each rewrite rule $\phi \to \psi$ consists of a couple of constraint schemes ϕ, ψ, i.e. constraints with logical variables. For instance

$$x =^? t \wedge P \mapsto\!\!\!\!\twoheadrightarrow x =^? t \wedge P\{x \mapsto t\}$$

where x is a variable, t a term, P a formula and $\{x \mapsto t\}$ is the substitution of x with t, is the *replacement rule*, x, t, P being logical variables of an appropriate type.

Each rule is assumed to rewrite a constraint into an equivalent one, i.e. both sides of the rewrite rule should have the same set of solutions in the constraint domain. Let us emphasize two important consequences of this assumption:

- the rewrite rules *replace* a constraint with another one. This makes an important difference with deduction rules: the premises are destroyed.
- If several rules can be applied to the same constraint, i.e. when the system is not deterministic, then we *don't care* which rule is actually applied. This makes a difference with e.g. logic programming in which the non-determinism is "don't know". For constraint solving, we would like to never have to backtrack.

Often, the rewrite rules are simple combination of the domain axiomatization. (For instance, the replacement rule is a logical consequence of equality axioms).

Then, each rule comes with an additional condition, expressing a *strategy.* This is necessary for the termination of rewriting. For instance, the replacement rule does not terminate. Hence we restrict it using the following conditions:

x occurs in P and does not occur in t. The rule is applied at top position.

These conditions impose both restrictions to the formulas to which the rule can be applied and restrictions on the positions at which the rule can be applied. The above condition ensures termination of the replacement rule alone (this is left as an exercise).

If the conditions are too strong, then constraint solving may become trivial. For example we could prevent any application of a rule, which, of course, ensures termination but is not very useful. Hence the definition of a constraint solving method includes the key notion of *solved form.* Solved forms are particular constraints, defined by a syntactic restriction and for which the satisfiability test should be trivial. The *completeness* of the set of rewrite rules expresses that any irreducible constraint is a solved form. We will see several examples in what follows.

In summary, a syntactic method consists in

A set of rewrite rules which is *correct*
Conditions on these rewrite rules which ensure *termination*
Solved forms with respect to which the rewrite rules are *complete.*

There are several advantages of such a presentation. First, it allows to define general environments and tools for the design of constraint solving methods. This is precisely the purpose of the programming language ELAN whose first class objects are precisely rules and strategies [KKV95, Cas98, Rin97, KR98]. We use ELAN in this lecture in order to operationalize constraint solvers. The system, together with its environment, the manual and many examples, is available at: `www.loria.fr/ELAN`.

A second advantage is to separate clearly the logical part from the control, which allows easier proofs and helps a better understanding. Finally, note that algorithms which are designed using this method are automatically incremental since new constraints can always be added (using a conjunction) to the result of former simplifications.

2.3 Unification Problems

We now define equational unification. But to give such a definition is not so easy since it should be simple and allow convincing soundness and completeness proofs. This is not the case with many of the definitions of equational unification which are given in the literature. The reason is that there are two contradicting goals: as far as generality is concerned, the definition should not depend upon a particular case, such as syntactic unification. On the other hand, since our approach is based on transforming problems in simpler ones having the same set of solutions, the definition must allow to get a clear, simple and robust definition of equivalence of two unification problems but also to be able to define what simpler problem means.

2.3.1 Solutions and Unifiers

When defining equational problems and their solutions and unifiers, we should be aware that some variables may appear or go away when transforming a given unification problem into a simpler one. As a consequence, our definition of a solution to a unification problem should not only care about the variables occurring in the simplified problem, but also of the variables which have appeared at the intermediate steps. The idea that the so-called "new variables" are simply existentially quantified variables appeared first in Comon [Com88] although quantifiers had already been introduced by Kirchner and Lescanne [KL87] both in the more general context of disunification.

Definition 1. Let $\mathcal{F}$ be a set of function symbols, $\mathcal{X}$ be a set of variables, and $\mathcal{A}$ be an $\mathcal{F}$-algebra. A $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$*-unification problem* (*unification problem* for short) is a first-order formula without negations nor universal quantifiers whose atoms are $\mathbf{T}, \mathbf{F}$ and $s =^?_{\mathcal{A}} t$, where s and t are terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$. We call an *equation* on $\mathcal{A}$ any $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification problem $s =^?_{\mathcal{A}} t$ and *multiequation* any multiset of terms in $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

Equational problems will be written as a disjunction of existentially quantified conjunctions:

$$\bigvee_{j \in J} \exists \overrightarrow{w_j} \bigwedge_{i \in I_j} s_i =^?_{\mathcal{A}} t_i.$$

When $|J| = 1$ the problem is called a *system*. Variables $\overrightarrow{w}$ in a system $P = \exists \overrightarrow{w} \bigwedge_{i \in I} s_i =^?_{\mathcal{A}} t_i$ are called *bound*, while the other variables are called *free*. Their respective sets are denoted by $\mathcal{BV}ar(P)$ and $\mathcal{V}ar(P)$.

The superscripted question mark is used to make clear that we want to solve the corresponding equalities, rather than to prove them.

Example 1. With obvious sets $\mathcal{X}$ and $\mathcal{F}$ and for an $\mathcal{F}$-algebra $\mathcal{A}$,

$$\exists z \; f(x, a) =^?_{\mathcal{A}} g(f(x, y), g(z, a)) \wedge x =^?_{\mathcal{A}} z$$

is a system of equations where the only bound variable is z and the free variables are x and y.

A solution of an equational problem is a valuation of the variables that makes the formula valid:

Definition 2. A $\mathcal{A}$*-solution* (for short a solution when $\mathcal{A}$ is clear from the context) to a $\langle \mathcal{F}, \mathcal{X}, \mathcal{A}\rangle$-unification system $P = \exists \overrightarrow{w} \bigwedge_{i \in I} s_i =^?_{\mathcal{A}} t_i$ is a homomorphism h from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to $\mathcal{A}$ such that

$$h, \mathcal{A} \models \exists \overrightarrow{w} \bigwedge_{i \in I} s_i =^?_{\mathcal{A}} t_i.$$

$x + 0 \to x$	$0 + x \to x$
$x * 0 \to 0$	$0 * x \to 0$
$pred(succ(x)) \to x$	$succ(pred(x)) \to x$
$opp(0) \to 0$	$opp(opp(x)) \to x$
$x + opp(x) \to 0$	$opp(x) + x \to 0$
$opp(pred(x)) \to succ(opp(x))$	$opp(succ(x)) \to pred(opp(x))$
$succ(x) + y \to succ(x + y)$	$x + succ(y) \to succ(x + y)$
$x + pred(y) \to pred(x + y)$	$pred(x) + y \to pred(x + y)$
$opp(x + y) \to opp(y) + opp(x)$	$x * succ(y) \to (x * y) + x$
$succ(x) * y \to y + (x * y)$	$(x + y) + z \to x + (y + z)$
$opp(y) + (y + z) \to z$	$x + (opp(x) + z) \to z$
$pred(x) * y \to opp(y) + (x * y)$	$x * pred(y) \to (x * y) + opp(x)$

Fig. 2.1. BasicArithmetic: Basic arithmetic axioms.

An $\mathcal{A}$-solution to a $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification problem $D = \bigvee_{j \in J} P_j$, where all the P_j are unification systems, is a homomorphism h from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to $\mathcal{A}$ such that h is solution of at least one of the P_j.

We denote by $\mathcal{S}ol_{\mathcal{A}}(D)$ the set of solutions of D in the algebra $\mathcal{A}$. Two $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification problems D and D' are said to be *$\mathcal{A}$-equivalent* if $\mathcal{S}ol_{\mathcal{A}}(D) = \mathcal{S}ol_{\mathcal{A}}(D')$, this is denoted $D' \Leftrightarrow_{\mathcal{A}} D$.

Note that because equations are interpreted as equality, the equation symbol is commutative. Therefore, except if explicitly mentioned, we make no difference between $s =^? t$ and $t =^? s$.

Finding solutions to a unification problem in an arbitrary $\mathcal{F}$-algebra $\mathcal{A}$ is impossible in general and when it is possible it is often difficult. For example, solving equations in the algebra $\mathcal{T}(\mathcal{F})/E$, where E is the BasicArithmetic theory given by the set of equational axioms described in Figure 2.1 is actually the problem of finding integer solutions to polynomial equations with integer coefficients. This is known as Hilbert's tenth problem, shown to be undecidable by Matijasevič [Mat70, DMR76].

Fortunately, we will see that the existence of solutions is decidable for many algebras of practical interest. However, there are in general infinitely many solutions to a unification system P. A first step towards the construction of a finite representation of these solutions is the notion of a *unifier,* which is meant as describing sets of *solutions*:

Definition 3. A *$\mathcal{A}$-unifier* of an $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification system

$$P = \exists \overrightarrow{w} \bigwedge_{i \in I} s_i =^?_{\mathcal{A}} t_i$$

is a substitution (i.e. an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$) σ such that

$$\mathcal{A} \models \overline{\forall}\ \exists \overrightarrow{w} \bigwedge_{i \in I} \sigma_{|\mathcal{X} - \overrightarrow{w}}(s_i) = \sigma_{|\mathcal{X} - \overrightarrow{w}}(t_i)$$

where $\overline{\forall} P$ denotes the universal closure of the formula P.

A $\mathcal{A}$-unifier of a $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification problem $D = \bigvee_{j \in J} P_j$, where all the P_j are $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification systems, is a substitution σ such that σ unifies at least one of the P_j.

We denote by $\mathcal{U}_{\mathcal{A}}(D)$ the set of unifiers of D. This is abbreviated $\mathcal{U}(D)$ when $\mathcal{A}$ is clear from the context. Similarly when clear from the context, $\mathcal{A}$-unifiers are called unifiers and $\langle \mathcal{F}, \mathcal{X}, \mathcal{A} \rangle$-unification is called unification.

The above definition is important, since it allows to define unifiers for equations whose solutions range over an arbitrary $\mathcal{F}$-algebra $\mathcal{A}$. This schematization is in particular due to the ability for the image of unifiers to contain free variables. These free variables are understood as universally quantified.

The relationship between solutions and unifiers is not quite as strong as we would like it to be. Of course, interpreting a unifier in the algebra $\mathcal{A}$ by applying an arbitrary homomorphism yields an homomorphism that is a solution. That all solutions are actually homomorphic images of unifiers is not true in general, but happens to be true in term generated algebras (allowing in particular free algebras generated by a given set of variables).

To illustrate the difference between solutions and unifiers, let us consider the set of symbols $\mathcal{F} = \{0, \succ, *\}$ and the $\mathcal{F}$-algebra $\mathbf{R}$ whose domain is the set of real numbers. Then $h(x) = \sqrt{2}$ is a solution of the equation $x * x =^?_{\mathbf{R}} \succ (\succ (0))$, although no $\mathbf{R}$-unifier exists for this equation, since the square root cannot be expressed in the syntax allowed by $\mathcal{F}$.

Indeed there is a large class of algebras where unifiers could be used as a complete representation of solutions since a unification system P has $\mathcal{A}$-solutions iff it admits $\mathcal{A}$-unifiers, provided $\mathcal{A}$ is a term generated $\mathcal{F}$-algebra.

Example 2. In the BasicArithmetic example, if we consider the equation $x =^? \succ (y)$, then $(x \mapsto 1, y \mapsto 2)$ is one of its $\mathbf{N}$-solutions. Its corresponding $\mathbf{N}$-unifier is $(x \mapsto \mathit{succ}(0), y \mapsto \mathit{succ}(\mathit{succ}(0)))$. The $\mathbf{N}$-unifier $(x \mapsto \mathit{succ}(y))$ also represents the previous $\mathbf{N}$-solution by simply valuating x to $\mathit{succ}(0)$ and y to $\mathit{succ}(\mathit{succ}(0))$.

In the following, we will restrict our attention to the special but fundamental case of the free algebras, initial algebras and their quotients. For these algebras the above property is satisfied since they are term generated by construction. As an important consequence of this property, two unification problems are equivalent iff they have the same sets of unifiers, an alternative definition that we are adopting in the remainder.

Definition 4. For a set of equational axioms E built on terms, for any terms s, t in $\mathcal{T}(\mathcal{F}, \mathcal{X})$, an equation to be solved in $\mathcal{A} = \mathcal{T}(\mathcal{F}, \mathcal{X})/E$ is denoted by $s =^?_E t$. The equivalence of unification problems is denoted by $\Leftrightarrow_E$ and $\mathcal{A}$-unification is called *E-unification.*

Since a unification system is a term whose outermost symbol is the existential quantifier, its body part, i.e., the conjunction of equations occurring in the system, is actually a subterm. Rewriting this subterm to transform the unification system into a simpler one hides the existential quantifier away. As a consequence, one may simply forget about the existential quantifier for most practical purposes. This is true for syntactic unification, where existential quantification is not needed as long as the transformation rules are chosen among the set given in Section 2.3.6. This is not, however, true of all algorithms for syntactic unification: Huet [Hue76] for example uses an abstraction rule introducing new variables in a way similar to the one used for combination problems.

Example 3. The equation $x{+}y =^?_E x{+}a$ is equivalent to the equation $y =^?_E a$, since the value of x is not relevant: our definition allows dropping useless variables. Note that this is not the case if unifiers are substitutions whose domain is restricted to the variables in the problem, a definition sometimes used in the literature, since $\{x \mapsto a, y \mapsto a\}$ would be a unifier of the first problem but not of the second.

Example 4. The equation $x{+}(y{*}y) =^?_E x{+}x$ is equivalent to $P' = \exists z\ x{+}z =^?_E x{+}x \wedge z =^?_E y{*}y$ whereas it is not equivalent to $P'' = x{+}z =^?_E x{+}x \wedge z =^?_E y{*}y$. In P'', z does not get an arbitrary value, whereas it may get any value in P, and in P' as well, since the substitution cannot be applied to the bound variable z.

Exercice 1 — Assuming the symbol $+$ commutative, prove that $x + f(a, y) =^? g(y, b) + f(a, f(a, b))$ is equivalent to $(x =^? g(y, b) \wedge f(a, y) =^? f(a, f(a, b))) \vee (x =^? f(a, f(a, b)) \wedge f(a, y) =^? g(y, b))$.

2.3.2 Generating Sets

Unifiers schematize solutions; let us now consider schematizing sets of unifiers using the notion of complete set of unifiers.

Complete sets of unifiers. Unifiers are representations of solutions but are still infinitely many in general. We can take advantage of the fact that any instance of a unifier is itself a unifier to keep a set of unifiers minimal with respect to instantiation.

In order to express this in general, we need to introduce a slightly more abstract concept of equality and subsumption as follows.

Definition 5. Let $\mathcal{A}$ be an $\mathcal{F}$-algebra. We say that two terms s and t are $\mathcal{A}$-*equal*, written $s =_{\mathcal{A}} t$ if $h(s) = h(t)$ for all homomorphisms h from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ into $\mathcal{A}$. A term t is an $\mathcal{A}$-*instance* of a term s, or s is *more general* than t in the algebra $\mathcal{A}$ if $t =_{\mathcal{A}} \sigma(s)$ for some substitution σ; in that case we write $s \leq_{\mathcal{A}} t$ and σ is called a $\mathcal{A}$-match from s to t. The relation $\leq_{\mathcal{A}}$ is a quasi-ordering on terms called $\mathcal{A}$-*subsumption*.

Subsumption is easily lifted to substitutions:

Definition 6. We say that two substitutions σ and τ are $\mathcal{A}$-*equal* on the set of variables $V \subseteq \mathcal{X}$, written $\sigma =^V_{\mathcal{A}} \tau$ if $\sigma(x) =_{\mathcal{A}} \tau(x)$ for all variables x in V. A substitution σ is *more general* for the algebra $\mathcal{A}$ on the set of variable V than a substitution τ, written $\sigma \leq^V_{\mathcal{A}} \tau$, if there exists a substitution ρ such that $\rho\sigma =^V_{\mathcal{A}} \tau$. The relation $\leq^V_{\mathcal{A}}$ is a quasi-ordering on substitutions called $\mathcal{A}$-*subsumption.* V is omitted when equal to $\mathcal{X}$.

The definition of generating sets that we are now presenting is issued from the definition given first by G. Plotkin [Plo72] followed by G. Huet [Hue76] and J.-M. Hullot [Hul80b]. We denote the domain and the rank of a substitution σ respectively $\mathcal{D}om(\sigma)$ and $\mathcal{R}an(\sigma)$.

Definition 7. Given an equational problem P, $CSU_{\mathcal{A}}(P)$ is a *complete set of unifiers* of P for the algebra $\mathcal{A}$ if:

(i) $CSU_{\mathcal{A}}(P) \subseteq \mathcal{U}_{\mathcal{A}}(P)$, (correctness)
(ii) $\forall\theta \in \mathcal{U}_{\mathcal{A}}(P)$, $\exists\sigma \in CSU_{\mathcal{A}}(P)$ such that $\sigma \leq^{\mathcal{V}ar(P)}_{\mathcal{A}} \theta$, (completeness)
(iii) $\forall\sigma \in CSU_{\mathcal{A}}(P), \mathcal{R}an(\sigma) \cap \mathcal{D}om(\sigma) = \emptyset$. (idempotency)

$CSU_{\mathcal{A}}(P)$ is called a *complete set of most general unifiers* of P in $\mathcal{A}$, and written $CSMGU_{\mathcal{A}}(P)$, if:

(iv) $\forall\alpha, \beta \in CSMGU_{\mathcal{A}}(P), \alpha \leq^{\mathcal{V}ar(P)}_{\mathcal{A}} \beta$ implies $\alpha = \beta$. (minimality)

Furthermore $CSU_{\mathcal{A}}(P)$ is said *outside the set of variables* W such that $\mathcal{V}ar(P) \subseteq W$ when:

(v) $\forall\sigma \in CSU_{\mathcal{A}}(P), \mathcal{D}om(\sigma) \subseteq \mathcal{V}ar(P)$ and $\mathcal{R}an(\sigma) \cap W = \emptyset$. (protection)

Notice that unifiers are compared only on the problem variables (i.e., $\mathcal{V}ar(P)$), a fundamental restriction as pointed out in particular by F. Baader in [Baa91]. The conditions (idempotency) as well as (protection) in the above definition insure that the unifiers are idempotent.

Exercice 2 — Give a description of all the $\emptyset$-unifiers of the equation $x =^? y$. Then give a complete set of $\emptyset$-unifiers outside $\{x, y\}$. How does the elements of this set compare to the unifier $\alpha = \{x \mapsto y\}$? Is your complete set of unifiers minimal?

Minimal complete set of unifiers not always exist, as shown by [FH86]. For example, in the theory FH defined by:

$$\mathsf{FH} = \begin{cases} f(0, x) = x \\ g(f(x, y)) = g(y). \end{cases}$$

the equation $g(x) =^?_{\mathsf{FH}} g(0)$ has no minimal complete set of FH-unifiers. Indeed, with

$$\begin{array}{rcl} \sigma_0 & = & \{x \mapsto 0\} \text{ and} \\ \sigma_i & = & \{x \mapsto f(x_i, \sigma_{i-1}(x))\} \ (0 < i) \end{array}$$

$\Sigma = \{\sigma_i | i \in \mathbf{N}\}$ is a complete set of FH-unifiers for the equation $g(x) =^?_{\mathsf{FH}} g(0)$ and $\forall i \in \mathbf{N}, \sigma_{i+1} \leq^{\{x\}}_{\mathsf{FH}} \sigma_i$.

2.3.3 (Un)-Decidability of Unification

Equational unification and matching are in general undecidable since there exist equational theories that have undecidable word problems. What is more disturbing is that very simple theories have undecidable E-unification problem. Let us review some of them.

Proposition 1. Let DA be the theory built over the set of symbols $\mathcal{F} = \{a, *, +\}$ and consisting of the axioms:

$$\begin{cases} x + (y + z) &= (x + y) + z \\ x * (y + z) &= (x * y) + (y * z) \\ (x + y) * z &= (x * z) + (x * z). \end{cases}$$

Unification is undecidable in DA as shown in [Sza82] using a reduction to Hilbert's tenth problem.

Another simple theory with undecidable unification problem is $D_l A U_r$, consisting in the associativity of $+$, the left distributivity of $*$ with respect to $+$ and a right unit element 1 satisfying $x * 1 = x$ [TA87].

In fact, decidability of unification is even quite sensitive to "new" constants. H.-J. Bürckert shows it by encoding the previous DA theory using new constants. This shows in particular that there exists equational theories for which unification is decidable but matching is not [Bür89].

The decidability of unification for classes of theories is also a very challenging problem. For example, variable permutative theories have an undecidable unification problem, as shown in [NO90], refining a result of [SS90]. Even in theories represented by a canonical term rewriting system (which is a strong requirement) the unification problem is undecidable:

Proposition 2. [Boc87] In the equational theory $\mathsf{BasicArithmetic}$ presented by the canonical term rewriting system $\overrightarrow{\mathsf{BasicArithmetic}}$, the unification and matching problems are undecidable.

2.3.4 A Classification of Theories with Respect to Unification

Since we have seen that minimal complete sets of E-unifiers are isomorphic whenever they exist, a classification of theories based on their cardinality makes sense, as pioneered by Szabó and Siekmann [SS84,Sza82,SS82]. But in doing so, we should be careful with the fact that solving one single equation is not general enough a question, as shown by the following result:

Proposition 3. [BHSS89] There exists equational theories E such that all single equations have a minimal complete set of E-unifiers, but some systems of equations are of type zero i.e. have no minimal complete set of E-unifiers.

Thus it makes sense to define the type of an equational theory based on the cardinality of minimal complete sets of E-unifiers for equation systems, when they exist.

Let P be a system of equations in an equational theory E, and let $CSMGU_E(P)$ be a complete set of most general E-unifiers of P, whenever it exists. E-unification is said to be:

U-based if $CSMGU_E(P)$ exists for all problems P (the class of U-based theories is denoted by $\mathcal{U}$),

U-unitary if $E \in \mathcal{U}$ and $|CSMGU_E(P)| \leq 1$ for all P,

U-finitary if $E \in \mathcal{U}$ and $|CSMGU_E(P)|$ is finite for all P,

U-infinitary if E is U-based but not finitary,

U-nullary if E is not U-based,

U-undecidable if it is undecidable whether a given unification problem has unifiers.

Syntactic unification is unitary as we have seen in Section 2.3.6 and so is unification in boolean rings [MN89]. Commutative unification is finitary, as we will see next in Section 2.3.7. So is also associative-commutative unification. Associative unification is infinitary [Plo72], take for example the equation $x + a =^?_{\mathsf{A}(+)} a + x$ of which incomparable $\mathsf{A}(+)$-unifiers are $\{x \mapsto a\}, \{x \mapsto a + a\}, \{x \mapsto a + (a + a)\}, \cdots$. We have seen that the theory FH is nullary.

One can wonder if this classification can be enhanced by allowing U-finitary theories with only 2 most general elements and 3 and 4 ... , but this is hopeless due to the result of [BS86] showing that in any given U-finitary but non U-unitary theory, there exists an equation the complete set of unifiers of which has more that n elements for any given natural number n.

Finally, given a finite presentation of a theory E, its position in the unification hierarchy is undecidable, i.e. it is undecidable whether E is U-unitary, U-finitary, U-infinitary or U-nullary [Nut89].

A summary of the current knowledge on equational unification can be found in [KK99] or [BS99].

2.3.5 Transforming Equational Problems

Solved forms for Unification Problems. We now define the solved forms needed for equational unification. We assume the conjunction symbol ($\wedge$) to be associative and commutative.

Definition 8. A *tree solved form* is any conjunction of equations:

$$\exists \vec{z},\ x_1 =^? t_1 \ \wedge \ \cdots \ \wedge \ x_n =^? t_n$$

such that $\forall 1 \leq i \leq n, x_i \in \mathcal{X}$ and:

$$\begin{array}{ll} (i) \ \forall 1 \leq i < j \leq n & x_i \neq x_j, \\ (ii) \ \forall 1 \leq i, j \leq n & x_i \notin \mathcal{V}ar(t_j), \\ (iii) \ \forall 1 \leq i \leq n & x_i \notin \overrightarrow{z}, \\ (iv) \ \forall z \in \overrightarrow{z}, \exists 1 \leq j \leq n \ z \in \mathcal{V}ar(t_j). & \end{array}$$

Given a unification problem P, we say that $\exists \overrightarrow{z}, \ x_1 =^? t_1 \ \wedge \ \cdots \ \wedge \ x_n =^? t_n$ is a tree solved form for P if it is a tree solved form equivalent to P and all variables free in $\exists \overrightarrow{z}, \ x_1 =^? t_1 \ \wedge \ \cdots \ \wedge \ x_n =^? t_n$ are free variables of P.

In the above definition, the first condition checks that a variable is given only one value, while the second checks that this value is a finite term. The third and fourth conditions check that the existential variables are useful, i.e., that they contribute to the value of the other variables.

Tree solved forms have the property to be solvable:

Lemma 1. Let $\mathcal{A}$ be an $\mathcal{F}$-algebra. A unification problem P with tree solved form:

$$P = \exists \overrightarrow{z}, \ x_1 =^? t_1 \ \wedge \ \cdots \ \wedge \ x_n =^? t_n$$

has, up to $\mathcal{A}$-subsumption equivalence, a unique most general idempotent unifier $\{x_1 \mapsto t_1, \cdots, x_n \mapsto t_n\}$ in $\mathcal{A}$ which is denoted μ_P.

The notion of tree solved form could be extended to allow structure sharing, leading to the so-called dag solved form:

Definition 9. A *dag solved form* is any set of equations

$$\exists \overrightarrow{z} \ , x_1 =^? t_1 \ \wedge \ \cdots \ \wedge \ x_n =^? t_n$$

such that $\forall 1 \leq i \leq n, x_i \in \mathcal{X}$ and:

$$\begin{array}{ll} (i) \ \forall 1 \leq i < j \leq n & x_i \neq x_j, \\ (ii) \ \forall 1 \leq i \leq j \leq n & x_i \notin \mathcal{V}ar(t_j), \\ (iii) \ \forall 1 \leq i \leq n & t_i \in \mathcal{X} \Rightarrow x_i, \ t_i \notin \overrightarrow{z}, \\ (iv) \ \forall z \in \overrightarrow{z}, \exists 1 \leq j \leq n \ z \in \mathcal{V}ar(t_j). & \end{array}$$

Given a unification problem P, we say that $\exists \overrightarrow{z} \ , x_1 =^? t_1 \ \wedge \ \cdots \ \wedge \ x_n =^? t_n$ is a dag solved form for P if it is a dag solved form equivalent to P and all variables free in $\exists \overrightarrow{z} \ , x_1 =^? t_1 \ \wedge \ \cdots \ \wedge \ x_n =^? t_n$ are free variables of P.

Of course, a tree solved form for P is a dag solved form for P. Dag solved forms save space, since the value of the variable x_j need not be duplicated in the t_i for $i \leq j$. Conversely, a dag solved form yields a tree solved form by replacing x_i by its value t_i in all t_j such that $j < i$ and removing the remaining unnecessary existentially quantified variables. Formally the job is done by the following transformation rule:

Eliminate $P \wedge x =^? s$
$\mapsto\!\!\!\!\twoheadrightarrow \{x \mapsto s\}P \wedge x =^? s$ if $x \notin \mathcal{V}ar(s), s \notin \mathcal{X}, x \in \mathcal{V}ar(P)$

Dag2Tree: Transformation of dag to tree solved forms

together with the simplification rules described in figure 2.2.

As a consequence we get solvability of dag solved forms:

Lemma 2. A unification problem $P = \exists \overrightarrow{z}\ , x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n$ in dag solved form has, up to $\mathcal{A}$-subsumption equivalence, a unique most general idempotent unifier $\sigma = \sigma_n \cdots \sigma_2\sigma_1$, where $\sigma_i = \{x_i \mapsto t_i\}$.

Dag solved forms relate to the so-called occur-check ordering on $\mathcal{X}$:

Definition 10. Given a unification problem P, let $\sim_P$ be the equivalence on $\mathcal{X}$ generated by the pairs (x, y) such that $x =^? y \in P$. The *occur-check* relation $\prec^{oc}$ on $\mathcal{X}$ defined by P is the quasi-ordering generated by the pairs (x', y') such that $x' \sim_P x, x =^? f(s_1, ..., s_n) \in P, y \in \mathcal{V}ar(f(s_1, ..., s_n)), y \sim_P y'$.

Example 5. For the system $P = (x =^? f(u, a) \wedge u =^? g(f(a, x)) \wedge x =^? y \wedge x =^? z)$ we have $x \sim_P y \sim_P z$ and $y \prec^{oc} u \prec^{oc} x$.

In a dag solved form, any two variables are not in the equivalence of the occur-check ordering. Conversely, a system of equations of the form $x =^? t$ with $x \in \mathcal{X}$ and such that $\prec^{oc}$ is acyclic, can be ordered (using topological sort) so as to meet the above condition. Accordingly, such a set of equations will be considered in dag solved form.

In the following, we refer without further precision to the most general unifier associated to a particular solved form by either one of the above lemmas.

Equivalence. We now state some commonly used transformations preserving the set of E-unifiers.

Proposition 4. Let E be a set of equational axioms built on terms of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Then, one can replace any subterm t in an equational problem P with an E-equal term without changing the set of E-unifiers of P.

In particular rewriting by some term rewriting system that is a sub-theory of E preserves the set of E-unifiers.

One quite important set of rules preserve equivalence of equational problems; they are the rules that allow manipulating the connectors $\wedge$ and $\vee$. The most commonly used such rules are described in Figure 2.2. This set of rules can be checked to be confluent modulo associativity-commutativity of

Associativity- $\wedge$	$(P_1 \wedge P_2) \wedge P_3$	$=$	$P_1 \wedge (P_2 \wedge P_3)$
Associativity- $\vee$	$(P_1 \vee P_2) \vee P_3$	$=$	$P_1 \vee (P_2 \vee P_3)$
Commutativity- $\wedge$	$P_1 \wedge P_2$	$=$	$P_2 \wedge P_1$
Commutativity- $\vee$	$P_1 \vee P_2$	$=$	$P_2 \vee P_1$

Trivial	$P \wedge (s =^? s)$	$\to$	P
AndIdemp	$P \wedge (e \wedge e)$	$\to$	$P \wedge e$
OrIdemp	$P \vee (e \vee e)$	$\to$	$P \vee e$
SimplifAnd1	$P \wedge \mathbf{T}$	$\to$	P
SimplifAnd2	$P \wedge \mathbf{F}$	$\to$	$\mathbf{F}$
SimplifOr1	$P \vee \mathbf{T}$	$\to$	$\mathbf{T}$
SimplifOr2	$P \vee \mathbf{F}$	$\to$	P
DistribCoD	$P \wedge (Q \vee R)$	$\to$	$(P \wedge Q) \vee (P \wedge R)$
Propag	$\exists \overrightarrow{z} : (P \vee Q)$	$\to$	$(\exists \overrightarrow{z} : P) \vee (\exists \overrightarrow{z} : Q)$
EElimin0	$\exists z : P$	$\to$	P if $z \notin \mathcal{V}ar(P)$
EElimin1	$\exists z : z =^? t \wedge P$	$\to$	P if $z \notin \mathcal{V}ar(P) \cup \mathcal{V}ar(t)$

Fig. 2.2. *RAUP*: Rules and Axioms for Connectors Simplification in Unification Problems

conjunction and disjunction ($\vee$ and $\wedge$). Note that we choose to use distributivity (the rule *DistribCoD*) in such a way that we get disjunctive normal forms, a most convenient representation of *unification* problems. Remember also that we assume the equation symbol $=^?$ to be commutative.

Proposition 5. All the rules in *RAUP* preserve the set of $\mathcal{A}$-unifiers, for any $\mathcal{F}$-algebra $\mathcal{A}$.

2.3.6 Syntactic Unification

Syntactic unification or $\emptyset$-unification is the process of solving equations in the free algebra $\mathcal{T}(\mathcal{F}, \mathcal{X})$. In this section, unification problems are assumed to be unquantified conjunctions of equations. This is so because there is no need for new variables to express the (unique) most general unifier. The notions of solved forms are also used without quantifiers and we can now define the transformation rules.

Notation: Given a set of equations P, remember that $\{x \mapsto s\}P$ denotes the system obtained from P by replacing all the occurrences of the variable x by the term s. The size of a term s is denoted $|s|$.

Let *SyntacticUnification* be the set of transformation rules defined in Figure 2.3. The transformation rules *Conflict* and *Decompose* must be understood as schemas, f and g being quantified over the signature. This is handled

$$
\begin{array}{lll}
\textit{Delete} & P \wedge s =^? s & \\
 & \twoheadrightarrow P & \\
\textit{Decompose} & P \wedge f(s_1,\dots,s_n) =^? f(t_1,\dots,t_n) & \\
 & \twoheadrightarrow P \wedge s_1 =^? t_1 \wedge \dots \wedge s_n =^? t_n & \\
\textit{Conflict} & P \wedge f(s_1,\dots,s_n) =^? g(t_1,\dots,t_p) & \\
 & \twoheadrightarrow \mathbf{F} & \text{if } f \neq g \\
\textit{Coalesce} & P \wedge x =^? y & \\
 & \twoheadrightarrow \{x \mapsto y\}P \wedge x =^? y & \text{if } x, y \in \mathcal{V}ar(P) \textit{ and } x \neq y \\
\textit{Check*} & P \wedge x_1 =^? s_1[x_2] \wedge \dots & \\
 & \quad \dots \wedge x_n =^? s_n[x_1] & \\
 & \twoheadrightarrow \mathbf{F} & \text{if } s_i \notin \mathcal{X} \textit{ for some } i \in [1..n] \\
\textit{Merge} & P \wedge x =^? s \wedge x =^? t & \\
 & \twoheadrightarrow P \wedge x =^? s \wedge s =^? t & \text{if } 0 < |s| \leq |t| \\
\textit{Check} & P \wedge x =^? s & \\
 & \twoheadrightarrow \mathbf{F} & \text{if } x \in \mathcal{V}ar(s) \textit{ and } s \notin \mathcal{X} \\
\textit{Eliminate} & P \wedge x =^? s & \\
 & \twoheadrightarrow \{x \mapsto s\}P \wedge x =^? s & \text{if } x \notin \mathcal{V}ar(s), s \notin \mathcal{X}, \\
 & & \quad x \in \mathcal{V}ar(P)
\end{array}
$$

Fig. 2.3. *SyntacticUnification*: Rules for syntactic unification

in ELAN by using a specific construction called "For Each" and used in Figure 2.4. We avoid merging *Coalesce* and *Eliminate* into a single rule on purpose, because they do not play the same role. *Coalesce* takes care of variable renaming: this is the price to pay for alpha-conversion. *Eliminate* is quite different from *Coalesce* because it makes terms growing, thus we will see how to avoid applying it.

First, all these rules are sound i.e. preserve the set of unifiers:

Lemma 3. All the rules in *SyntacticUnification* are sound.

A strategy of application of the rules in *SyntacticUnification* determines a unification procedure. Some are complete, some are not, but a brute force fair strategy is complete:

Theorem 1. *Starting with a unification problem P and using the above rules repeatedly until none is applicable results in* $\mathbf{F}$ *iff P has no unifier, or else it results in a tree solved form of P:*

$$x_1 =^? t_1 \wedge \cdots \wedge x_n =^? t_n.$$

Moreover

$$\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$$

is a most general unifier of P.

Exercice 3 — In fact, the condition $0 < |s| \leq |t|$ is fundamental in the *Merge* rule. Give a unification problem P such that without that condition *Merge* does not terminate.

Now that we have proved that the whole set of rules terminates, we can envisage complete restrictions of it. Let us first define useful subsets of the rules in *SyntacticUnification*. We introduce the set of rules:

TreeUnify = {*Delete, Decompose, Conflict, Coalesce, Check, Eliminate*}
and
DagUnify = {*Delete, Decompose, Conflict, Coalesce, Check*, Merge*}.

Corollary 1. Starting with a unification problem P and using the rules *TreeUnify* repeatedly until none is applicable, results in **F** iff P has no unifier, or else in a tree solved form:

$$x_1 =^? t_1 \wedge \ldots \wedge x_n =^? t_n$$

such that $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ is a most general unifier of P.

Exercice 4 — Apply the set of rules *TreeUnify* to the following unification problems:

$$f(g(h(x), a), z, g(h(a), y)) =^? f(g(h(g(y, y)), a), x, z)$$
$$f(g(h(x), a), z, g(h(a), y)) =^? f(a, x, z)$$

We can also forbid the application of the *Eliminate* rule, in which case we get dag solved forms:

Corollary 2. Starting with a unification problem P and using the rules *DagUnify* repeatedly until none is applicable, results in **F** iff P has no unifier, or else in a dag solved form

$$x_1 =^? t_1 \wedge \ldots \wedge x_n =^? t_n$$

such that $\sigma = \{x_n \mapsto t_n\} \ldots \{x_1 \mapsto t_1\}$ is a most general unifier of P.

Exercice 5 — Apply the set of rules *DagUnify* to the following unification problem:

$$f(g(h(x), a), z, g(h(a), y)) =^? f(g(h(g(y, y)), a), x, z)$$

Compare with what you get using the set of rules *TreeUnify*.

The previous results permit to implement complete solving strategies in ELAN. The complete module performing syntactic unification is given in Figure 2.4 and is available with the standard library of ELAN [BCD+98]. Notice that a unification problem is built over the binary operator $\wedge$ (the conjunction) which satisfies $AC(\wedge)$-axioms. Therefore, the unification rules are applied modulo these axioms. The decomposition rule depends on the signature;

```
module unification[Vars,Fss]
import global termV[Vars] termF[Fss] unifPb ;
 local Fss int identifier bool pair[identifier,int]
 list[pair[identifier,int]] eq[variable] eq[term] eq[Fsymbol]
 occur[variable,term] occur[variable,unifPb] ;
end

stratop global
 unify         : <unifPb -> UnifPb> bs;  end

rules for unifPb
 P : unifPb; s,t : term; x,y : variable;
local
 [delete]      P ^ x=y => P
                    if eq_variable(x,y)                end
 [coalesce]    P ^ x=y => apply(x->y,P) ^ x=y
                    if occurs x in P
                    and occurs y in P and neq_variable(x,y) end
 [conflict]    P ^ s=t => false
                    if not(isvar(s)) and not(isvar(t))
                    and neq_Fsymbol(head(s),head(t))  end
 [occ_check]   P ^ x=s => false
                    if occurs x in s and not isvar(s)  end
 [occ_check]   P ^ s=x => false
                    if occurs x in s and not isvar(s)  end
 [eliminate]   P ^ x=s => apply(x->s,P) ^ x=s
                    if not(occurs x in s)
                    and occurs x in P and not isvar(s)  end
 [eliminate]   P ^ s=x => apply(x->s,P) ^ x=s
                    if not(occurs x in s)
                    and occurs x in P and not isvar(s)  end
 [trueadd] P => true ^ P                                end
 [trueelim] true ^ P=> P                                end
 [identity] P => P                                      end
end

 FOR EACH SS:pair[identifier,int]; F:identifier; N:int
 SUCH THAT SS:=(listExtract) elem(Fss) AND F:=()first(SS)
        AND N:=()second(SS) :{
 rules for unifPb
 s_1,...,s_N:term; t_1,...,t_N:term;
 local
 [decompose] P ^ F(s_1,...,s_N)=F(t_1,...,t_N)
 =>
 P { ^ s_I=t_I }_I=1...N
 end end }

strategies for unifPb
implicit
 [] unify => dc one(trueadd) ;
 repeat*(dc(dc one(delete), dc one(decompose),
 dc one(conflict), dc one(coalesce),
 dc one(occ_check), dc one(eliminate))) ;
 dc one(trueelim, identity)
end end end
```

Fig. 2.4. Syntactic Unification in ELAN

a nice feature allowed in ELAN is to allow a natural expression of such rules by the use of a very general pre-processor. Note also the way strategies are described: dc expresses a “dont care” choice, repeat* applies as much as

$$\begin{array}{ll}
\textit{Decompose} & P \wedge f(s_1,\dots,s_n) =^? f(t_1,\dots,t_n) \\
 & \mapsto\!\!\!\twoheadrightarrow \\
 & P \wedge s_1 =^? t_1 \wedge \dots \wedge s_n =^? t_n \\
 & \textsf{if } f \neq + \\
\textit{ComMutate} & P \wedge s_1 + s_2 =^?_{\mathsf{C}} t_1 + t_2 \\
 & \mapsto\!\!\!\twoheadrightarrow \\
 & P \wedge \begin{pmatrix} s_1 =^?_{\mathsf{C}} t_1 \wedge s_2 =^?_{\mathsf{C}} t_2 \\ \vee \\ s_1 =^?_{\mathsf{C}} t_2 \wedge s_2 =^?_{\mathsf{C}} t_1 \end{pmatrix}
\end{array}$$

Fig. 2.5. *CommutativeUnification*: The main rules for commutative unification

possible its argument and one allows to keep only one result out of all the possible ones.

Let us finally mention that syntactic unification has been proved to be linear in the size of the equation to be solved [PW78], provided that equality between two variable occurrences can be tested in constant time. But quasi-linear algorithms using dag-solved forms are often more useful in practice.

One may also ask whether syntactic unification can be speeded up by using massive parallelism. It is somewhat surprising that this is not the case, due to the non-linearity of terms. Dwork et al. [DKM84] show that unification of terms is logspace complete for P: unless $P \subset NC$, no parallel algorithm for unifying s and t will run in a time bounded by a polynomial in the logarithm of $|s| + |t|$ with a number of processors bounded by a polynomial in P.

2.3.7 Unification Modulo Commutativity

As in the free case, unification rules for unification modulo commutativity transform a unification problem into a set of equivalent unification problems in solved form. These rules are the same as for syntactic unification but should now embed the fact that some symbols are commutative. We give the modified rules for a commutative theory where we assume that $\mathcal{F} = \{a, b, \dots, f, g, \dots, +\}$ and $+$ is commutative, the others function symbols being free. Then a set of rules for unification in this theory can be easily obtained by adding a *mutation* rule to the set *Dag-Unify*, which describes the effect of the commutativity axiom. We call the resulting set of rules *CommutativeUnification*: the modified rules are described in Figure 2.5. It is very important to notice that a more modular and clever way to get a unification algorithm for this theory is indeed to *combine* unification algorithm for free symbols and for individual commutative symbols. This is detailed in the lecture by [Baader and Schultz].

$$
\begin{array}{llll}
\textit{Delete} & P \wedge s =^?_E s & \Vdash\!\!\twoheadrightarrow & P \\
\textit{Decompose} & P \wedge f(\overrightarrow{s}) =^?_E f(\overrightarrow{t}) & \Vdash\!\!\twoheadrightarrow & P \wedge s_1 =^?_E t_1 \wedge \ldots \wedge s_n =^?_E t_n \\
\textit{Coalesce} & P \wedge x =^?_E y & \Vdash\!\!\twoheadrightarrow & \{x \mapsto y\}P \wedge x =^?_E y \\
 & & & \text{if } x, y \in \mathcal{V}ar(P) \text{ and } x \neq y \\
\textit{Eliminate} & P \wedge x =^?_E s & \Vdash\!\!\twoheadrightarrow & \{x \mapsto s\}P \wedge x =^?_E s \\
 & & & \text{if } x \notin \mathcal{V}ar(s),\ s \notin \mathcal{X} \text{ and } x \in \mathcal{V}ar(P) \\
\textit{LazyPara} & P \wedge s =^?_E t & \Vdash\!\!\twoheadrightarrow & P \wedge s|_p =^?_E l \wedge s[r]_p =^?_E t \\
 & & & \text{if } s|_p \notin \mathcal{X} \text{ and } s|_p(\Lambda) = l(\Lambda) \\
 & & & \text{where } l = r \in E
\end{array}
$$

Fig. 2.6. *GS-Unify*: Gallier and Snyder's rules for E-unification

We see how easy it is here to obtain a set of rules for unification modulo commutativity from the set of rules for syntactic unification. Note that again, there is no need of using existential quantifiers here.

Theorem 2. *Starting with a unification problem P and using repeatedly the rules CommutativeUnification, given in figure 2.5, until none is applicable results in* **F** *iff P has no C-unifier, or else it results in a finite disjunction of tree solved form:*

$$\bigvee_{j \in J} x_1^j =^?_{\mathsf{C}} t_1^j \wedge \cdots \wedge x_n^j =^?_{\mathsf{C}} t_n^j$$

having the same set of C-unifiers than P. Moreover:

$$\Sigma = \{\sigma^j | j \in J \text{ and } \sigma^j = \{x_1^j \mapsto t_1^j, \ldots, x_n^j \mapsto t_n^j\}\}$$

is a complete set of C-unifiers of P.

As for syntactic unification, specific complete strategies can be designed. Note also that by removing *Eliminate*, we get a set of rules for solving equations over infinite trees, exactly as in the free case.

2.3.8 General E-Unification

As we have seen, equational unification is undecidable since unification of ground terms boils down to the word problem. It is indeed semi-decidable by interleaving production of substitutions with generation of equational proofs. Gallier and Snyder gave a complete set of rules for enumerating a complete set of unifiers to a unification problem P in an arbitrary theory E [GS87, GS89, Sny88]. This set of rules is given is Figure 2.6.

The rule *LazyPara* (for lazy paramodulation) implements a lazy (since the induced unification problem is not solved right-away) use of the equations in E. Every time such an equation is used in the rule set, the assumption is tacitly made that the variables of the equation are renamed to avoid possible captures.

Gallier and Snyder prove that for any E-unifier γ of a problem P, there exists a sequence of rules (where *Eliminate* and *Coalesce* are always applied immediately after *LazyPara*) whose result is a tree solved form yielding an idempotent unifier $\sigma \leq \gamma$. In this sense, the set of rules is complete. This result is improved in [DJ90b] where a restrictive version of *LazyPara* is proved to be complete. General E-unification transformations have also been given in [Höl89].

2.3.9 Narrowing

Narrowing is a relation on terms that generalizes rewriting in using unification instead of matching in order to apply a rewrite rule. This relation has been first introduced by M. Fay to perform unification in equational theories presented by a confluent and terminating term rewriting system, and this is our motivation for introducing it now. Another important application is its use as an operational semantics of logic and functional programming languages like BABEL [MNRA92], EQLOG [GM86] SLOG [Fri85], and this will be used in the chapter [[TOY]].

Narrowing a term t is finding an instantiation of t such that one rewrite step becomes applicable, and to apply it. This is achieved by replacing a non-variable subterm which unifies with a left-hand side of a rewrite rule by the right-hand side, and by instantiating the result with the computed unifier. In this process, is it enough to take the most general unifier. If this process is applied to an equation seen as a term with top symbol $=^?$, and is iterated until finding an equation whose both terms are syntactically unifiable, then the composition of the most general unifier with all the substitutions computed during the narrowing sequence yields a unifier of the initial equation in the equational theory. The narrowing process that builds all the possible narrowing derivations starting from the equation to be solved, is a general unification method that yields complete sets of unifiers, provided that the theory is presented by a terminating and confluent rewrite system [Fay79, Hul80a]. Furthermore, this method is incremental since it allows building, from a unification algorithm in a theory A, a unification procedure for a theory $R \cup A$, provided the class rewrite system defined by R and A is Church-Rosser and terminating modulo A [JKK83].

However, the drawback of such a general method is that it very often diverges and several attempts have been made to restrict the size of the narrowing derivation tree [Hul80a, NRS89, WBK94]. A successful method to solve this problem has been first proposed by J.-M. Hullot in restricting narrowing to *basic* narrowing. It has the main advantage to separate the solving of the syntactic unification constraint from the narrowing process itself. It is in fact a particular case of deduction with constraints, and the terminology "basic," indeed comes from this seminal work [Hul80a].

In this section we present the two relations of narrowing and basic (or constraint) narrowing and their application to the unification problem in equational theories presented by a terminating and confluent term rewrite system.

Narrowing relations.

Definition 11. (Narrowing) A term t is *narrowed* into t', at the non variable position $p \in \mathcal{D}om(t)$, using the rewrite rule $l \to r$ and the substitution σ, when σ is a most general unifier of $t|_p$ and l and $t' = \sigma(t[r]_p)$. This is denoted $t \leadsto_{[p,l\to r,\sigma]} t'$ and it is always assumed that there is no variable conflict between the rule and the term, i.e. that $\mathcal{V}ar(l,r) \cap \mathcal{V}ar(t) = \emptyset$.

For a given term rewriting system R, this generates a binary relation on terms called *narrowing* relation and denoted $\leadsto^R$.

Note that narrowing is a natural extension of rewriting since unification is used instead of matching. As a consequence the rewriting relation is always included in the narrowing one: $\longrightarrow^R \subseteq \leadsto^R$ since, for terms with disjoint sets of variables, a match is always a unifier.

Example 6. If we consider the rule $f(f(x)) \to x$ then the term $f(y)$ narrows at position Λ:

$$f(y) \leadsto_{[\Lambda, f(f(x))\to x, \{(x\mapsto z),(y\mapsto f(z))\}]} z.$$

On this example, we can notice that narrowing may introduce new variables, due to the unification step. Now, if we narrow the term $g(y, f(y))$, we get the following derivation:

$$\begin{aligned} g(y,f(y)) &\leadsto_{[2, f(f(x))\to x, \{(x\mapsto z),(y\mapsto f(z))\}]} g(f(z),z) \\ &\leadsto_{[1, f(f(x))\to x, \{(x\mapsto z'),(z\mapsto f(z'))\}]} g(z',f(z')) \\ &\ldots \end{aligned}$$

which shows that even if the term rewriting system terminates, the narrowing derivation may not be so.

Exercice 6 — Use the system BasicArithmetic on page 51 to narrow the terms $succ(succ(0)) + pred(0)$, $succ(succ(x)) + pred(0)$, $succ(succ(x)) + pred(y)$.

Definition 12. A *constrained term* $(\exists W, t \parallel c)$ is a couple made of a term t and a system of constraints c together with a set of existentially quantified variables W. It schematizes the set of all instances of t by a solution of c with no assumption on W, i.e.

$$\{\exists W, \sigma(t) \mid \sigma \in \mathcal{S}ol(\exists W, c)\}.$$

The set of free variables of a constrained term $(\exists W, t \parallel c)$ is the union of the set of free variables of t and the free variables set of c minus W: $\mathcal{V}ar((\exists W, t \parallel c)) = \mathcal{V}ar(t) \cup \mathcal{V}ar(c) \setminus W$.

We consider in this section only constraint consisting of system of syntactic equations. This can be extended to more general constraint languages and domains as proposed for example in [KK89, KKR90, Cha94].

Example 7. The formula

$$\tau = (f(x, f(a, x)) \parallel \exists z, f(x, z) =^? f(g(a, y), f(a, y)) \wedge y =^? g(u, b))$$

is a constrained term. It schematizes the terms:

$$\begin{array}{l} f(g(a, g(u, b)), f(a, g(a, g(u, b)))), \\ f(g(a, g(a, b)), f(a, g(a, g(a, b)))), \\ f(g(a, g(b, b)), f(a, g(a, g(b, b)))), \\ \dots \end{array}$$

and $\mathcal{V}ar(\tau) = \{x, y, u\}$.

Definition 13. (Constrained narrowing) A constrained term $(\exists W, t[u]_p \parallel c)$ *c-narrows* (narrows with constraints) into the constrained term

$$(\exists W \cup \mathcal{V}ar(l), t[r]_p \parallel c \wedge u =^?_\emptyset l)$$

at the non-variable position $p \in \mathcal{G}rd(t)$, using the rewrite rule $l \to r$ of the rewrite system R, if the system $c \wedge u =^?_\emptyset l$ is satisfiable and provided that the variables of the rule and the constrained terms are disjoint: $\mathcal{V}ar(l, r) \cap \mathcal{V}ar((t \parallel c)) = \emptyset$. This is denoted:

$$(\exists W, t[u]_p \parallel c) \overset{c}{\leadsto}^R_{[p, l \to r]} (\exists W \cup \mathcal{V}ar(l), t[r]_p \parallel c \wedge u =^?_\emptyset l).$$

Example 8. If we consider as previously the rule $f(f(x)) \to x$, then the term $(f(y) \parallel \mathbf{T})$ c-narrows at position Λ:

$$(\exists, f(y) \parallel \mathbf{T}) \overset{c}{\leadsto}_{[\Lambda, f(f(x)) \to x]} (\exists \{x\}, x \parallel f(y) =^?_\emptyset f(f(x))),$$

and similarly:

$$(\exists, g(y, f(y)) \parallel \mathbf{T}) \overset{c}{\leadsto}_{[2, f(f(x)) \to x,]} (\exists \{x\}, g(y, x) \parallel f(y) =^? f(f(x))).$$

The relation between rewriting and narrowing can be made more precise than just the trivial relationship $\longrightarrow^R \subseteq \leadsto^R$:

Lemma 4. For any term t and term rewriting system R, if $t \leadsto^R_{[m, g \to d, \sigma]} t'$ then $\sigma(t) \longrightarrow^R_{[m, g \to d]} t'$.

This can be pictured as follows:

$$\begin{tikzcd}
& \sigma(t) \arrow[dr, dashed, "{[m,g\to d]}"] & \\
t \arrow[ur, dashed, "\sigma"] \arrow[rr, squiggly, "{[m,g\to d,\sigma]}"] & & \sigma(t[d]_m)
\end{tikzcd}$$

The dual of this property, i.e. the rewriting to narrowing correspondence schema is more subtle and has been exhibited first by J.-M. Hullot [Hul80a, Hul80b].

Proposition 6. Let t_0 be a term and ρ be a R-normalized substitution such that $\rho(t_0) \longrightarrow^{R}_{[m,g\to d]} t'_1$. Then there exist substitutions σ et μ such that:

1. $t_0 \rightsquigarrow^{R}_{[m,g\to d,\sigma]} t_1$,
2. $\mu(t_1) = t'_1$,
3. $\rho =^{\mathcal{V}ar(t_0)} \mu\sigma$,
4. μ is R-normalized.

$$\begin{tikzcd}
\rho(t_0) \arrow[r, "g\to d"] & t'_1 \\
t_0 \arrow[u, mapsto, "\rho"] \arrow[r, squiggly, "{[m,g\to d,\sigma]}"] & t_1 \arrow[u, mapsto, dashed, "\mu"']
\end{tikzcd}$$

This result can be easily extended by induction on the number of steps to any rewriting derivation.

In order to apply narrowing to equational unification, it is convenient to introduce, for any rewrite system R on $\mathcal{T}(\mathcal{F},\mathcal{X})$, a new rule $x =^?_R x \to \mathbf{T}$ in which the symbols $=^?_R$ and $\mathbf{T}$ are considered as new symbols of the signature (i.e. $\mathbf{T}, =^?_R \;\notin \mathcal{F}$). This rule matches the equation $s =^?_R t$ if and only if s and t are identical; it narrows $s =^?_R t$ iff s and t are syntactically unifiable. Note that this could be also viewed as rewriting propositions instead of terms, an approach developed in [DHK98]. This leads to an easy characterization of R-unifiers:

Lemma 5. Let σ be a substitution, s and t be terms and R be any confluent rewrite system. Let $\bar{R} = R \cup \{x =^?_R x \to \mathbf{T}\}$. σ is a R-unifier of s and t if and only if $\sigma(s =^?_R t) \overset{\bar{R}}{\underset{}{\xrightarrow{*}}} \mathbf{T}$.

Since we also consider constrained equation systems, let us now define what a solution of such an entity is.

Definition 14. Let R be a term rewriting system on $\mathcal{T}(\mathcal{F},\mathcal{X})$. A *constrained system* is a constrained term $(\exists W, P \parallel c)$ where $P = \bigwedge_{i=1,\dots,n} s_i =^?_R t_i$ is a system of R-equations, i.e. a term in $\mathcal{T}(\mathcal{F} \cup \{\wedge, =^?_R\}, \mathcal{X})$ and c is a system of equations in the empty theory: $c = \exists W' \bigwedge_{i=1,\dots,n} s_i =^?_\emptyset t_i$. A *$R$-unifier* of a constrained system $(\exists W, P \parallel c)$ is a substitution σ which is a $\emptyset$-unifier of $(\exists W, c)$ and an R-unifier of $(\exists W, P)$.

For example $(x =^?_R y \parallel \mathbf{F})$ has no R-unifier and the R-solutions of $(s =^?_R t \parallel \mathbf{T})$ are the same as the R-unifiers of $s =^?_R t$.

Proposition 7. (Correctness) Let R be a term rewriting system. Let

$$\begin{aligned} G &= (\exists W, s =^?_R t \parallel c) \\ G' &= (\exists W \cup \mathcal{V}ar(l), s[r]_p =^?_R t \parallel c \wedge s|_p =^?_\emptyset l) \end{aligned}$$

If $G \rightsquigarrow^{R}_{[1.p,l\to r]} G'$ then $\mathcal{U}_R(G') \subseteq \mathcal{U}_R(G)$.

Proving completeness could be achieved using generalizations of Lemma 4 and Property 6:

Narrow $(\exists W, s =^?_R t \parallel c)$
$\mapsto\!\!\!\twoheadrightarrow$
$(\exists W \cup \mathcal{V}ar(l), s[r]_p =^?_R t \parallel c \wedge s|_p =^?_\emptyset l)$
if $(c \wedge (s_{|p} =^?_\emptyset l))$ is satisfiable and $s|_p$ is not a variable

Block $(\exists W, s =^?_R t \parallel c)$
$\mapsto\!\!\!\twoheadrightarrow$
$(\exists W, \mathbf{T} \parallel c \wedge s =^?_\emptyset t)$
if $(c \wedge (s =^?_\emptyset t))$ is satisfiable

Fig. 2.7. *Narrowing*: Unification via constrained narrowing

Theorem 3. *Let R be a terminating and confluent term rewriting system and $\bar{R} = R \cup \{x =^? x \to \mathbf{T}\}$. If the substitution σ is a R-unifier of the terms s and t, then there exists a constrained narrowing derivation:*

$$(\exists\emptyset, s =^?_R t \parallel \mathbf{T}) \overset{c}{\leadsto}^{\bar{R}} \dots \overset{c}{\leadsto}^{\bar{R}} (\exists W_n, \mathbf{T} \parallel c_n),$$

such that $\sigma \in \mathcal{U}_\emptyset(c_n)$.

Let us consider now the *constrained narrowing tree* whose root is labeled with the constrained term $(\exists\emptyset, s =^?_R t \parallel \mathbf{T})$ and whose edges are all possible constrained narrowing derivations issued from a given node. In this tree, which is in general infinite, a *successful leave* is by definition a node labeled by a constrained term of the form: $(\exists W, \mathbf{T} \parallel c)$ with c satisfiable. For a given equation $s =^?_R t$, we denote $\mathcal{SNT}(s =^?_R t)$ the set of all successful nodes of the constrained narrowing tree issued from $(\exists\emptyset, s =^?_R t \parallel \mathbf{T})$.

Thanks to Theorem 3, we have:

$$\mathcal{U}_R(s =^?_R t) \subseteq \bigcup_{(\exists W, \mathbf{T} \parallel c) \in \mathcal{SNT}(s =^?_R t)} \mathcal{U}_\emptyset(c),$$

and since constrained narrowing is correct (by Property 7) we get the equality:

$$\mathcal{U}_R(s =^?_R t) = \bigcup_{(\exists W, \mathbf{T} \parallel c) \in \mathcal{SNT}(s =^?_R t)} \mathcal{U}_\emptyset(c).$$

This justifies the following main result about constrained narrowing:

Corollary 3. The transformation rules described in Figure 2.7, applied in a non deterministic and fair way to the constrained equation $(\exists\emptyset, s =^?_R t \parallel \mathbf{T})$, yield constrained equations of the form $(\exists W, \mathbf{T} \parallel c)$ such that the most general -unifiers of the c's form altogether a complete set of R-unifiers of $s =^?_R t$.

Note that in the set of rules *Narrowing*, the *Block* rule mimics exactly the application of the rule $x =^?_R x \to \mathbf{T}$.

Example 9. If we consider the rewrite system R reduced to the rule $f(f(y)) \to y$, then the constrained equation $(\exists\emptyset, f(x) =^?_R x \parallel \mathbf{T})$ is rewritten, using the rules *Narrowing* as follows:

$$\begin{array}{ll} & (\exists\emptyset, f(x) =^?_R x \parallel \mathbf{T}) \\ \mapsto\!\!\!\!\twoheadrightarrow_{\mathbf{Narrow}} & (\exists\{y\}, y =^?_R x \parallel f(f(y)) =^?_\emptyset f(x)) \\ \mapsto\!\!\!\!\twoheadrightarrow_{\mathbf{Block}} & (\exists\{y\}, \mathbf{T} \parallel y =^?_\emptyset x \wedge f(f(y)) =^?_\emptyset f(x)) \\ \mapsto\!\!\!\!\twoheadrightarrow_{\mathbf{SyntacticUnification}} & (\exists\{y\}, \mathbf{T} \parallel \mathbf{F}) \end{array}$$

See exercise 8 to conclude about the solution set of this equation.

Exercice 7 — Use the system BasicArithmetic on page 51 to solve the equation $x * x =^? x + x$ using *constrained* narrowing.

Exercice 8 — Let $R = \{f(f(x)) \to x\}$. Show that the standard narrowing is not terminating on the equation $f(y) =^?_R y$, as on the contrary basic or constrained narrowing does. What is the complete set of R-unifiers of this equation?

Notice that the previous proof of completeness of constrained narrowing can be extended to equational unification. This allows dealing in particular with narrowing modulo associativity and commutativity. Using normalization during the narrowing process could be achieved in a complete way. The latest paper on the subject, linking the problematic to the redundancy notions used in automated theorem proving, is [Nie95].

2.4 Dis-Unification Problems

As the name suggests, dis-unification is concerned with the generalization of unification to formulas where negation and arbitrary quantification are allowed. Many problems can be formalized in this setting and dis-unification has many useful applications from logic to computer science and theorem proving. Solving arbitrary first-order formulas whose only symbol is equality in an algebra $\mathcal{A}$ shows the decidability of the theory of $\mathcal{A}$ and provides a complete axiomatization of $\mathcal{A}$ [Mah88]. Dealing with negation in logic programming or when automating inductive theorem proving leads to solve dis-unification problems.

For a detailed motivation and presentation of dis-unification, see the surveys [CL89, Com91].

2.4.1 Equational Formulas, Semantics and Solved Forms

We call *equational formula* any first-order formula built on the logical connectives $\vee, \wedge, \neg$, the quantifiers $\forall$ and $\exists$ and which atomic formulas are equations or the constant predicate $\mathbf{T}$.

The set of $\mathcal{A}$-solutions is defined as previously for constraint systems.

The most common domains that have been considered in the literature are the following:

$\mathcal{A} = \mathcal{T}(\mathcal{F})$ **or** $\mathcal{T}(\mathcal{F}, \mathcal{X})$ This is the interpretation used when interested for example in complement problems (see for example [Com86]). The main point when dealing with such interpretation is the finiteness of $\mathcal{F}$ as we will see later. The case of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ will be considered as a special case of $\mathcal{T}(\mathcal{F})$ where the (infinite many) variables are considered as (infinitely many) constants.

$\mathcal{A} = RF(\mathcal{F})$ the algebra of rational terms is considered for example in constrained logic programming [Mah88]. When dealing with $IT(\mathcal{F})$, the algebra of infinite terms (not necessarily rational), [Mah88] has proved that a formula holds in $IT(\mathcal{F})$ if and only if it holds in $RT(\mathcal{F})$.

$\mathcal{A} = NF_{\mathcal{R}}(\mathcal{F})$ the subset of terms in $\mathcal{T}(\mathcal{F})$ that are in normal form with respect to the term rewrite system $\mathcal{R}$. Such interpretation are considered in particular in [Com89].

$\mathcal{A} = \mathcal{T}(\mathcal{F})/E$ where E is a finite set of equational axioms. This has been studied for specific set of axioms (with associativity-commutativity as an important theory from the application point of view) as well as in the general case using narrowing based techniques.

2.4.2 Solving Equational Formulas in $\mathcal{T}(\mathcal{F})$

Following the approach proposed in the section 2.2, we now define the solved forms under consideration when dealing with equational formulas when the interpretation domain is $\mathcal{A} = \mathcal{T}(\mathcal{F})$.

Definition 15. A basic formula is either **F**, **T** or

$$\exists \overrightarrow{z} : x_1 = t_1 \wedge \ldots \wedge x_n = t_n \wedge z_1 \neq u_1 \wedge \ldots \wedge z_m \neq u_m$$

where

- $x_1, \ldots, x_n$ are free variables which occur only once
- $z_1, \ldots, z_m$ are variables s.t. $\forall i, z_i \notin Var(u_i)$

Lemma 6. [Independence of dis-equations] If $\mathcal{T}(\mathcal{F})$ is infinite, and if E is a finite conjunction of equations and D is a finite conjunction of dis-equations then $E \wedge D$ has a solution in $\mathcal{T}(\mathcal{F})$ iff for each $s \neq t \in D$, $E \wedge s \neq t$ has a solution in $\mathcal{T}(\mathcal{F})$.

Lemma 7. The basic formulas which are distinct from **F** are solvable.

And indeed the solved forms are finite disjunction of basic formulas:

Theorem 4. *[Com91] If $\mathcal{A}$ is either $\mathcal{T}(\mathcal{F})$, $IT(\mathcal{F})$ or $RT(\mathcal{F})$ then any equational formula is equivalent to a finite disjunction of basic formulas.*

Let us now show how to compute the solved form of any equational formula in $\mathcal{T}(\mathcal{F})$.

$$\mathsf{SUP} - \mathit{DistribDoC}$$
$$\cup$$

Dis-equation	$\neg(s =^? t)$	$\rightarrow s \neq^? t$
TrivialDis	$s \neq^? s$	$\rightarrow \mathbf{F}$
NegT	$\neg\mathbf{T}$	$\rightarrow \mathbf{F}$
NegF	$\neg\mathbf{F}$	$\rightarrow \mathbf{T}$
NegNeg	$\neg\neg e$	$\rightarrow e$
NegDis	$\neg(e \vee e')$	$\rightarrow \neg e \wedge \neg e'$
NegConj	$\neg(e \wedge e')$	$\rightarrow \neg e \vee \neg e'$
DistribDoC	$e \vee (e' \wedge e'')$	$\rightarrow (e \vee e') \wedge (e \vee e'')$

Fig. 2.8. SEF: Rules for connectors Simplification in Equational Formulas

2.4.3 Solving Dis-Equation on Finite Terms

Our goal is here to transform any equational problem into its solved form assuming that the solving domain is $\mathcal{A} = \mathcal{T}(\mathcal{F})$. Indeed it is enough to transform the so called elementary formulas into basic formulas since, by successive application of the transformation rules and the appropriate use of double negation, this will allow to solve any equational formula. This is explained in [CL89][Section 6.1], presented by transformation rules is [Com91].

Definition 16. We call *elementary formulas* any disjunction of formulas of the form

$$\exists \overrightarrow{z}\ \forall \overrightarrow{y}\ P$$

where P is a quantifier-free equational formula.

The rules for connectors simplification (SUP) described in Figure 2.2 are extended and modified in order to cope with dis-equations. In particular, we orient this time distributivity (*DistribCoD*) in order to get conjunctive normal forms.

We give now some of the rules reducing elementary formulas to basic ones. A full description of the rules could be find in [Com91].

The replacement rule of unification should be extended to deal with dis-equations. In particular we get:

$$\mathit{Replacement2}\ z \neq^? t \vee P[z] \mapsto\!\!\!\!\rightarrow z \neq^? t \vee \{z \mapsto t\}P[z]$$

if z is a free variable,
t does not contain any occurrence of a universally quantified variable,
$z \notin \mathcal{V}ar(t)$ and,
if t is a variable, then it occurs in P.

We have to deal directly with quantifiers that should be eliminated as far as possible either universal;

ElimUQ1	$\forall y : P$	$\mapsto\!\!\!\!\twoheadrightarrow P$ if y does not occur free in P
ElimUQ2	$\forall y : y \neq^? u \wedge P$	$\mapsto\!\!\!\!\twoheadrightarrow \mathbf{F}$ if $y \notin \mathcal{V}ar(u)$
ElimUQ3	$\forall y : y \neq^? u \vee d$	$\mapsto\!\!\!\!\twoheadrightarrow \{y \mapsto u\}d$ if $y \notin \mathcal{V}ar(u)$

or existential:

ElimEQ1	$\exists x : P$	$\mapsto\!\!\!\!\twoheadrightarrow P$ if x does not occur free in P
ElimEQ2	$\exists z : z =^? t \wedge P$	$\mapsto\!\!\!\!\twoheadrightarrow P$ if $z \notin \mathcal{V}ar(t, P)$

With respect to unification, a new clash rule appears:

Clash $f(t_1, \ldots, t_m) \neq^? g(u_1, \ldots, u_n) \mapsto\!\!\!\!\twoheadrightarrow \mathbf{T}$
if $f \neq g$

and the decomposition rule should take care of dis-equalities:

Decomp2 $f(t_1, \ldots, t_m) \neq^? f(u_1, \ldots, u_m)$
$\mapsto\!\!\!\!\twoheadrightarrow$
$t_1 \neq^? u_1 \vee \ldots \vee t_m \neq^? u_m$

Decomp3 $(f(t_1, \ldots, t_m) =^? f(u_1, \ldots, u_m) \vee d) \wedge P$
$\mapsto\!\!\!\!\twoheadrightarrow$
$(t_1 =^? u_1 \vee d) \wedge \ldots \wedge (t_m =^? u_m \vee d) \wedge P$
if one of the terms $t_1, \ldots, t_m, u_1, \ldots, u_m$ contains a universally quantified variable, or else d does not contain any universally quantified variable

Of course the occur check may now also generate positive information:

OccurC2 $s \neq^? u[s] \mapsto\!\!\!\!\twoheadrightarrow \mathbf{T}$
if s and $u[s]$ are not syntactically equal

When we assume that the domain consists in finite trees over a finite set of symbols, more quantifier eliminations can be performed like in:

ElimEQ3 $\exists \overrightarrow{w} : (d_1 \vee z_1 \neq^? u_1) \wedge \ldots \wedge (d_n \vee z_n \neq^? u_n) \wedge P$
$\mapsto\!\!\!\!\twoheadrightarrow$
$\exists \overrightarrow{w} : P$
if there exists a variable $w \in \overrightarrow{w} \cap Var(z_1, u_1) \cap \ldots \cap Var(z_n, u_n)$ which does not occur in P.

and finite search could be used, when everything else have failed:

Explosion $\exists \overrightarrow{w_1} : P$

$\mapsto\!\!\!\twoheadrightarrow$

$$\bigvee_{f \in F} \exists \overrightarrow{w}\, \exists \overrightarrow{w_1} : P \wedge z =^? f(\overrightarrow{w})$$

if $\overrightarrow{w} \cap \mathcal{V}ar(P) = \emptyset$,
no other rule can be applied and
there exists in P an equation or a dis-equation $z = u$ where u contains an occurrence of a universally quantified variable.

All these rules are correct and are consequences of the axiom system for finite trees over a finite alphabet. The full set of rules as given in [Com91] is terminating for any reduction strategy and the solved forms that are reached are basic formulas. As a consequence:

Theorem 5. *[Com88, Mah88] The first-order theory of finite trees is decidable.*

The above rules provide a way to reduce elementary formulas to basic ones. Of course obtaining solved form consisting only of equalities, and therefore giving a basis of the dis-unifiers set under the form of a complete set of substitutions is more appealing and as been addressed in [LM87, LMM88], [MS90, CF92], also in some equational cases [Fer98].

The complexity of dis-unification, even in the syntactic case where no equational theory is involved, can be high. In particular deciding satisfiability of elementary formulas is NP-complete [Pic99].

When extending dis-unification to equational theories, narrowing-based procedures could be designed [Fer92]. Unfortunately, even for theories as simple as associativity and commutativity, dis-unifiability becomes undecidable for general problems [Tre92] but is still decidable when considering the existential fragment i.e. elementary formulas without universal quantifiers [Com93]. It is also interesting to note that when restricting to shallow theories (where the equational axioms involve only variable at depth less than one), dis-unification is still decidable [CHJ94].

2.5 Ordering Constraints

We now consider the case where the symbolic constraints under consideration are based on an ordering predicate interpreted as a recursive path ordering on terms. We are presenting only the basic results and main references on the literature. A more detailed account of results could be found in the survey paper [CT94].

As this will be detailed in other lectures of this school, this kind of ordering constraints is extremely useful in theorem proving [KKR90, HR91, NR95], [BGLS95] as well as in programming language design since they allow to prune the search space by keeping into account simplification as well as deletion strategies.

This section is devoted to ordering constraints where the ordering predicate is interpreted as a path ordering, but several other possibilities have been already explored:

Subterm ordering are studied in [Ven87].
Encompassment ordering [CCD93] play a central role in the ground reducibility problem [CJ94].
Matching ordering is surveyed in Section 2.6.

A *reduction ordering* $>$ on $\mathcal{T}(\mathcal{F},\mathcal{X})$ is a well-founded ordering closed under context and substitution, that is such that for any context $C[\_]$ and any substitution σ, if $t > s$ then $C[t] > C[s]$ and $\sigma(t) > \sigma(s)$.

Reduction orderings are exactly what is needed when dealing with termination of term rewrite systems since a rewrite system R over the set of terms $\mathcal{T}(\mathcal{F},\mathcal{X})$ is terminating iff there exists a reduction ordering $>$ such that each rule $l \to r \in R$ satisfies $l > r$.

Two most commonly used methods for building reduction orderings are polynomial interpretations and path orderings. Even if a very general notion of general path ordering can be defined [DH95], we restrict here to the *recursive path ordering* possibly with status.

Definition 17. Let us assume that each symbol f in the signature has a status, $Stat(f)$ which can be either lexicographic (lex) or multiset ($mult$).

The equality up to multisets, $=^{mult}$, is defined on terms as equality up to permutation of direct arguments of function symbols with multiset status.

Let $>_{\mathcal{F}}$ be a precedence on $\mathcal{F}$. The *recursive path ordering with status* $>_{rpos}$ is defined on terms by $s = f(s_1,..,s_n) >_{rpos} t = g(t_1,\dots,t_m)$ if one at least of the following conditions holds:

1. $s_i >_{rpos} t$ or $s_i =^{mult} t$, for some i with $1 \le i \le n$, or
2. $f >_{\mathcal{F}} g$ and $\forall j \in \{1,\dots,m\}, s >_{rpos} t_j$, or
3. $f = g$, $Stat(f) = lex$, $(s_1,\dots,s_n) >^{lex}_{rpos} (t_1,\dots,t_m)$ and $\forall i \in \{1,\dots,n\}, s >_{rpos} t_i$, or
4. $f = g$, $Stat(f) = mult$ and $\{s_1,\dots,s_n\} >^{mult}_{rpos} \{t_1,\dots,t_m\}$.

where $>^{mult}_{rpos}$ and ${}^{lex}_{rpos}$ are the multiset and lexicographic extensions of $>_{rpos}$ respectively (the reader is referred to [KK99, BS99] for a definition of these extensions).

When all symbols have the lexicographic status, we speak of lexicographic path ordering (LPO for short). When conversely they all have the multiset status, we speak of the multiset or recursive path ordering (RPO for short). Such orderings are called generically *path orderings*.

A *term ordering constraint* is a quantifier-free formula built over the binary predicate symbols $>$ and $=$ which are interpreted respectively as a given path ordering $\succ$ and a congruence on terms. We denote an inequation by $s \succ^? t$. A solution to an ordering constraint c is a substitution σ such that $\sigma(c)$ evaluates to true.

When solving such ordering constraints, a first main concern is about the introduction of new constants in order to express the solutions. This rises the distinction between fixed signature semantics and extended signature ones [NR95]. The satisfiability problem for ordering constraints has been shown to be decidable for fixed signature either when $>$ is a total LPO [Com90], or when it is a recursive path ordering with status [JO91]. In the case of extended signature, the decidability has been shown for RPO by [Nie93] and for LPO in [NR95]. Concerning complexity, NP algorithms have been given for LPO (in both fixed and extended signatures), RPO (for extended signatures) [Nie93] and for RPO with fixed signatures [NRV98]. NP-hardness is known, in all the cases and even for a single inequation [CT94].

Following the latest results of [Nie99] to which we refer for the full details, the satisfiability of a path ordering constraint can be achieved by first reducing it to a solved form which is then mainly checked to be occur-check free. The reduction to solved form follows the principle we have already seen for other constraint systems. The set of rules are of course using the definition of RPOS in order to decompose problem in simpler ones. For example, right decomposition is achieved via the rule:

$$\begin{array}{lll} \textit{decompR} & S \wedge s \succ^? f(t_1, \dots, t_n) \mapsto\!\!\!\!\rightarrow & S \wedge s \succ^? t_1 \wedge \dots \wedge s \succ^? t_n \\ & & \text{if } \textit{top}(s) >_{\mathcal{F}} f \end{array}$$

Left decomposition is achieved with some nondeterminism handled here by a disjunction:

$$\begin{array}{lll} \textit{decompL} & S \wedge f(s_1, \dots, s_n) \succ^? t \mapsto\!\!\!\!\rightarrow & \bigvee_{1 \le i \le n} S \wedge s_i \succ^? t \vee \bigvee_{1 \le i \le n} S \\ & & \wedge s_i =^? t \\ & & \text{if } \textit{top}(t) >_{\mathcal{F}} f \end{array}$$

where the introduced equations are solved modulo multiset equality.

2.6 Matching Constraints

The matching process is a symbolic computation of main interest for programming languages and in automated deduction. For applying a rewrite rule $l \to r$ to a (ground) term s, we have to decide if the left hand-side l of the rule "matches" a *ground* subterm t of s. This matching problem, denoted by $l \le^? t$ consists to unify l with a ground term t. According to this definition, matching is a very specific case of unification (unification with a *ground* term) and can be solved by reusing a unification algorithm (or strategy) if such an algorithm exists like in the empty theory. But there exist also equational theories where matching is decidable but unification is not, and in many situations, matching is much less complex than unification.

2.6.1 Syntactic Matching

The matching substitution from t to t', when it exists, is unique and can be computed by a simple recursive algorithm given for example by G. Huet [Hue76] and that we are describing now.

Definition 18. A *match-equation* is any formula of the form $t \ll^? t'$, where t and t' are terms. A substitution σ is solution of the match-equation $t \ll^? t'$ if $\sigma t = t'$. A *matching system* is a conjunction of match-equations. A substitution is solution of a matching system P if it is solution of all the match-equations in P. We denote by $\mathbf{F}$ a matching system without solution.

We are now ready to describe the computation of matches by the following set of transformation rules *Match* where the symbol $\wedge$ is assumed to satisfy the rules and axioms of figure 2.2.

Delete	$t \ll^? t \wedge P$	$\mapsto\!\!\!\!\twoheadrightarrow$	P
Decomposition	$f(t_1, \dots, t_n) \ll^? f(t'_1, \dots, t'_n) \wedge P$	$\mapsto\!\!\!\!\twoheadrightarrow$	$\bigwedge_{i=1,\dots,n} t_i \ll^? t'_i \wedge P$
SymbolClash	$f(t_1, \dots, t_n) \ll^? g(t'_1, \dots, t'_m) \wedge P$	$\mapsto\!\!\!\!\twoheadrightarrow$	$\mathbf{F}$ if $f \neq g$
MergingClash	$x \ll^? t \wedge x \ll^? t' \wedge P$	$\mapsto\!\!\!\!\twoheadrightarrow$	$\mathbf{F}$ if $t \neq t'$
SymbVarClash	$f(t_1, \dots, t_n) \ll^? x \wedge P$	$\mapsto\!\!\!\!\twoheadrightarrow$	$\mathbf{F}$ if $x \in \mathcal{X}$

Match: Rules for syntactic matching

Theorem 6. *[KK99] The normal form by the rules in Match, of any matching problem $t \ll^? t'$ such that $\mathcal{V}ar(t) \cap \mathcal{V}ar(t') = \emptyset$, exists and is unique.*

1. *If it is $\mathbf{F}$, then there is no match from t to t'.*
2. *If it is of the form $\bigwedge_{i \in I} x_i \ll^? t_i$ with $I \neq \emptyset$, the substitution $\sigma = \{x_i \mapsto t_i\}_{i \in I}$ is the unique match from t to t'.*
3. *If it is $\mathbf{T}$ then t and t' are identical: $t = t'$.*

Exercice 9 — Write a program, in the language of your choice, implementing a matching algorithm derived from the *Match* set of rules.

Exercice 10 — Compute the match from the term $f(g(z), f(y, z))$ to the term $f(g(f(a, x)), f(g(c), f(a, x)))$.

It should be noted that the rule *Delete* in the previous set of rules is not correct when the two members of the match equation $t \ll^? t'$ share variables. This can be seen on the following example. The matching problem $f(x, x) \ll^? f(x, a)$ has no solution but the unrestricted use of the transformations leads to: $f(x,x) \ll^? f(x,a) \mapsto\!\!\!\!\rightarrow_{\mathbf{Decomposition}} \{x \ll^? x, x \ll^? a\} \mapsto\!\!\!\!\rightarrow_{\mathbf{Delete}} \{x \ll^? a\}$ which has an obvious solution.

The above transformation rules could be easily generalized to deal with commutative matching. But the extension to associative *and* commutative matching is not straitforward and needs either to use semantical arguments like solving systems of linear Diophantine equations [Hul79, Mza86, Eke95] or to use the concept of syntactic theories [Kir85, KK90] to find the appropriate transformation rules [AK92, KR98].

2.7 Principles of Automata Based Constraint Solving

In the *syntactic methods* which have been presented previously, the aim of the constraint solving procedure is to reach a *solved form* which is an equivalent (yet simpler) representation of the set of solutions of the original constraint. Here, we consider another representation of the set of solutions: an automaton.

The relationship between logic and automata goes back to Büchi, Elgot and Church in the early sixties [Büc60, Elg61, Chu62]. The basic idea is to associate with each atomic formula a device (an automaton) which accepts all the models of the formula. Then, using the closure properties of the recognized languages, we can build an automaton accepting all the models of an arbitrary given formula. This is also the basis of optimal decision techniques (resp. model-checking techniques) for propositional temporal logic (see e.g. [Var96, BVW94]).

In this lecture, we illustrate the method with three main examples, using three different notions of automata:

Classical word automata. They allow to represent the set of solutions for arbitrary Presburger formulas, yielding a decision procedure for Presburger arithmetic. This technique is known since Büchi, but has been re-discovered recently with applications in constraint solving. This constraint solving method has been implemented in several places and competes with other arithmetic solvers as shown in [BC96, SKR98, Kla99]. We survey this very simple method in section 2.8.

Automata on finite trees. They allow to represent sets of terms, hence solutions of typing (membership) constraints on terms. We introduce such constraints as well as tree automata in section 2.9. There are several other applications of tree automata in constraint solving. For instance we will sketch applications in unification theory (sections 2.11.1, 2.11.2). They are also used e.g. in type reconstruction, in which case the interpretation domain is a set of trees representing types themselves (see e.g. [TW93]).

Tree set automata. They recognize sets of sets of terms. We define such a device in section 2.10 and show how it can be used in solving the so-called *set constraints* which were already surveyed at the CCL conference in 1994 [Koz94].

A significant part of these lecture notes is developed in [CDG$^+$97]. There was already a survey of these techniques at the CCL conference in 1994 [Dau94].

2.8 Presburger Arithmetic and Classical Word Automata

We recall briefly the standard definitions in sections 2.8.1 and 2.8.2. Then we show how automata recognize sets of integers and we give a construction for the decision of arithmetic constraints.

2.8.1 Presburger Arithmetic

The *basic terms* in Presburger arithmetic consist of first-order variables, which we write $x, x_1, y, z'...$, the constants 0 and 1 and sums of basic terms. For instance $x + x + 1 + 1 + 1$ is a basic term, which we also write $2x + 3$.

The *atomic formulas* are equalities and inequalities between basic terms. For instance $x + 2y = 3z + 1$ is an atomic formula.

The *formulas* of the logic are first-order formulas built on the atomic formulas. We use the connectives $\wedge$ (conjunction), $\vee$ (disjunction), $\neg$ (negation), $\exists x$ (existential quantification), $\forall x$ (universal quantification). For instance, $\forall x, \exists y.(x = 2y \vee x = 2y + 1)$ is a formula. $\top$ is the empty conjunction (valid formula) and $\perp$ is the empty disjunction (unsatisfiable formula).

The *free variables* of a formula ϕ are defined as usual: for instance $FV(\phi_1 \vee \phi_2) = FV(\phi_1) \cup FV(\phi_2)$ and $FV(\exists x.\phi) = FV(\phi) \setminus \{x\}$.

The interpretation domain of formulas is the set of natural numbers $\mathbb{N}$, in which $0, 1, +, =, \leq$ have their usual meaning. A *solution* of a formula $\phi(x_1, \ldots, x_n)$ is an assignment of $x_1, \ldots, x_n$ in $\mathbb{N}$ which satisfies the formula. For instance $\{x \mapsto 0; y \mapsto 2; z \mapsto 1\}$ is a solution of $x + 2y = 3z + 1$ and every assignment $\{x \mapsto n\}$ is a solution of $\exists y.(x = 2y \vee x = 2y + 1)$.

2.8.2 Finite Automata

For every word $w \in A^*$, if i is a natural number which is smaller or equal to the length of w, we write $w(i)$ the ith letter of w.

A *finite automaton* on the alphabet A is a tuple (Q, Q_f, q_0, δ) where Q is a finite set of *states*, $Q_f \subseteq Q$ is the set of *final states*, $q_0 \in Q$ is the *initial state* and $\delta \subseteq Q \times Q \times Q$ is the *transition relation*.

Given a word $w \in A^*$ a *run* of the automaton (Q, Q_f, q_0, δ) on w is a word $\rho \in Q^*$ such that $\rho(1) = q_0$ and, if $\rho(i) = q$, $\rho(i+1) = q'$, then $(q, w(i), q') \in \delta$. A *successful run* ρ on w is a run such that $\rho(n+1) \in Q_f$ where n is the length of w. w is *accepted by the automaton* if there is a successful run on w. The *language* accepted (or recognized) by an automaton is the set of words on which there is a successful run.

The class of languages which are accepted by some finite automaton is closed by union, intersection and complement. Such constructions can be found in most textbooks. (See e.g. [Per90] for references).

2.8.3 Sets of Integers Recognized by a Finite Automaton

Every natural number can be seen as a word written in base k ($k \geq 1$) over the alphabet $A = \{0, ..., k-1\}$. For convenience, we write here its representation in a somehow unusual way from right to left. For instance thirteen can be written in base two as: 1011. There are more than one representation for each number, since we may complete the word with any number of 0's on the right: 101100 is another representation of thirteen in base two. Conversely, there is a mapping denoted $\widetilde{\ }^k$ from $\{0, ..., k-1\}^*$ into $\mathbb{N}$ which associates with each word a natural number: $\widetilde{10110}^2$ is thirteen.

n-uples of natural numbers can be represented in base k as a single word over the alphabet $\{0, ..., k-1\}^n$ by stacking the representations of each individual numbers. For instance the pair $(13, 6)$ (in base 10) can be represented in base 2 as a word over the alphabet $\{0,1\}^2$ as: $^{1011}_{0110}$. Again there are more than one representation since the word can be completed by any number of $^0_0$'s on the right.

2.8.4 A Translation from Presburger Formulas to Finite Automata

In the logic versus automata spirit we start by associating an automaton with each atomic formula.

Automata associated with equalities. For every basic formula

$$a_1x_1 + \ldots + a_nx_n = b \text{ (where } a_1, \ldots, a_n, b \in \mathbf{Z})$$

we construct the automaton by saturating the set of rules and the set of states, originally set to $\{q_b\}$ using the inference rule:

$$\frac{q_c \in Q}{q_d \in Q, \quad (q_c, \theta, q_d) \in \delta} \quad \text{if} \quad \begin{cases} a_1\theta_1 + \ldots + a_n\theta_n =_2 c \\ d = \frac{c - a_1\theta_1 \ldots - a_n\theta_n}{2} \\ \theta \in \{0,1\}^n \text{encodes } (\theta_1, \ldots, \theta_n) \end{cases}$$

in other words: for every state $q_c \in Q$, compute the solutions $(\theta_1, \ldots, \theta_n)$ of $a_1x_1 + \ldots + a_nx_n = c$ modulo 2 and add the state q_d and the rule $q_c \xrightarrow{\theta} q_d$ where $d = \frac{c - a_1\theta_1 \ldots - a_n\theta_n}{2}$.

The initial state is q_b and, if $q_0 \in Q$, then this is the only final state. Otherwise there is no final state.

Example 10. Consider the equation $x + 2y = 3z + 1$.

Since $b = 1$, we have $q_1 \in Q$. We compute the solutions modulo 2 of $x+2y = 3z+1$: we get $\{(0,0,1),(0,1,1),(1,0,0),(1,1,0)\}$. Then we compute the new states: $\frac{1-0-0+3}{2} = 2$, $\frac{1-0-2+3}{2} = 1$, $\frac{1-1-0+0}{2} = 0$, $\frac{1-1-2+0}{2} = -1$, yielding $q_2, q_0, q_{-1} \in Q$, and the new transitions:

$$q_1 \xrightarrow{\substack{0\\0\\1}} q_2, q_1 \xrightarrow{\substack{0\\1\\1}} q_1, q_1 \xrightarrow{\substack{1\\0\\0}} q_0, q_1 \xrightarrow{\substack{1\\1\\0}} q_{-1}.$$

Going on with states q_2, q_0, q_{-1}, we get the automaton of figure 2.9; we use circles for the states and double circles for final states.

For instance, consider the path from the initial state q_1 to the final state q_0 going successively through $q_{-1}q_{-2}q_0q_1q_2$. The word $\begin{smallmatrix}111100\\110001\\001110\end{smallmatrix}$ is accepted by the automaton, which corresponds to the triple (in base ten): $x = 15, y = 35, z = 28$ and one can verify $15 + 2 \times 35 = 3 \times 28 + 1$.

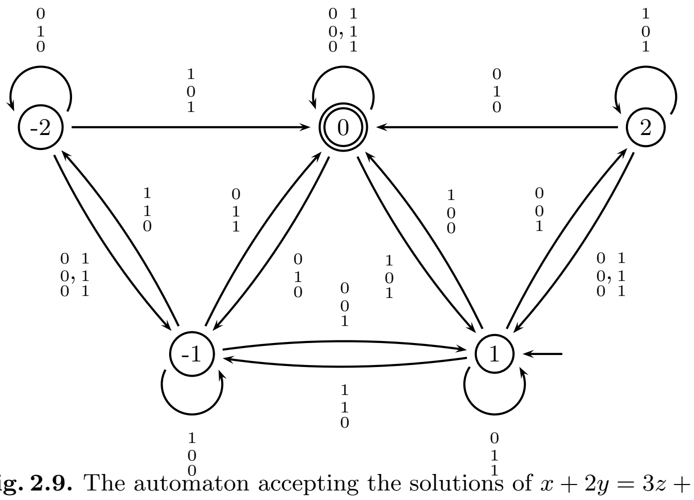


Fig. 2.9. The automaton accepting the solutions of $x + 2y = 3z + 1$

Proposition 8. *The saturation terminates yielding an (deterministic) automaton whose number of states is bounded by* $\max(|b|, |a_1| + \ldots + |a_n|)$ *and which accepts the solutions of* $a_1x_1 + \ldots + a_nx_n = b$.

Indeed, it is not difficult to check that the absolute values of the states are bounded by $\max(|b|, |a_1| + \ldots + |a_n|)$ (this is an invariant of the inference rule, the proof is left to the reader). The correctness and completeness are also easy: it suffices to show by induction on the length of the word (resp. on the length of the transition sequence) that, any word accepted starting from any state q_c is a solution of $a_1x_1 + \ldots + a_nx_n = c$.

The algorithm can be optimized in several respects, for instance pre-computing the solutions modulo 2 or using Binary Decision Diagrams (BDDs) to represent the transitions of the automaton.

Exercise 1. If we fix $a_1, \dots a_n$, what is the size of the automaton, w.r.t. b?

Automata associated with inequalities. Next, we compute an automaton for inequalities $a_1x_1 + \ldots + a_nx_n \leq b$. The computation is similar: we start with q_b and, from a state q_c we compute the transitions and states as follows. For every bit vector $(\theta_1, \dots \theta_n)$, $q_c \xrightarrow{\theta_1 \dots \theta_n} q_d$ with

$$d = \lfloor \frac{c - \sum_{i=1}^{n} a_i\theta_i}{2} \rfloor$$

$Q_f = \{q_c \mid c \geq 0\}$.

Example 11. Consider the inequality $2x - y \leq -1$. The automaton which we get using the algorithm is displayed on figure 2.10.

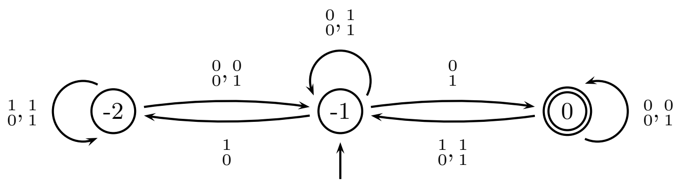


Fig. 2.10. The automaton accepting the solutions of $2x - y \leq -1$

Exercise 2. What is the largest number of reachable states in the automaton for $a_1x_1 + \ldots + a_nx_n \leq b$?

Closure properties and automata for arbitrary formulas.
Let $\mathcal{A}_\phi(x_1, \dots, x_n)$ be an automaton over the alphabet $\{0,1\}^n$ which accepts the solutions of the formula ϕ whose free variables are contained in $\{x_1, \dots, x_n\}$. If a variable x_i does not occur free in ϕ, then accepting/not accepting a word will not depend on what we have on the ith track.

Exercise 3. Show how to compute $\mathcal{A}_\phi(x_1, \dots, x_n, y)$, given $\mathcal{A}_\phi(x_1, \dots, x_n)$.

Given two automata $\mathcal{A}_{\phi_1}(\overrightarrow{x})$ and $\mathcal{A}_{\phi_2}(\overrightarrow{x})$ we can compute $\mathcal{A}_{\phi_1 \wedge \phi_2}(\overrightarrow{x})$ and $\mathcal{A}_{\phi_1 \vee \phi_2}(\overrightarrow{x})$ by the classical constructions for intersection and union of finite automata. We only have to be sure that $\overrightarrow{x}$ contains the free variables of both ϕ_1 and ϕ_2, which can be easily be satisfied, thanks to the above exercise.

Then negation corresponds to the complement, a classical construction which is linear when the automaton is deterministic (and may be exponential otherwise). Since the automata which were constructed for atomic formulas are deterministic, computing the automaton for $a_1x_1 + \ldots + a_nx_n \neq b$ is easy.

Now, it remains to handle the quantifiers. Since $\forall x.\phi$ is logically equivalent to $\neg\exists x.\neg\phi$, we only have to consider existential quantification. $\mathcal{A}_{\exists x.\phi}$ is computed from $\mathcal{A}_\phi$ by *projection*: the states, final states, initial state and transitions of $\mathcal{A}_{\exists x.\phi}$ are identical to those of $\mathcal{A}_\phi$, except that we forget the track corresponding to x in the labels of the transitions.

Example 12. Consider the formula $\exists z.x + 2y = 3z + 1$. The automaton is obtained from the automaton of figure 2.9, forgetting about z. This yields the automaton of figure 2.11. By chance, this automaton is still deterministic.

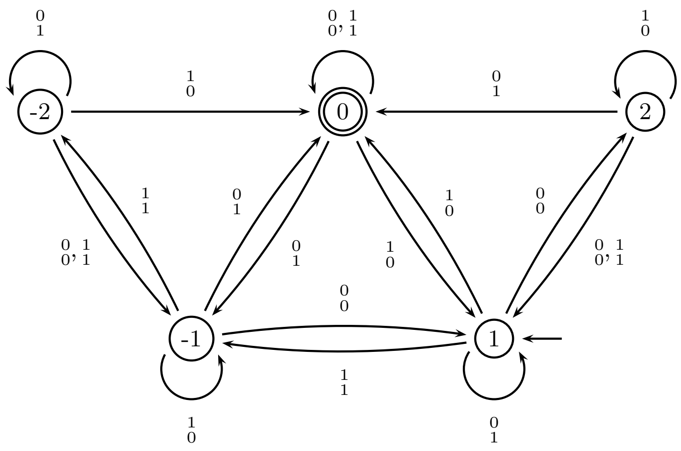


Fig. 2.11. The automaton accepting the solutions of $\exists z.x + 2y = 3z + 1$

But this is not the case in general.

Finally, by induction on the Presburger formula ϕ, we can compute an automaton which accepts the set of solutions of ϕ.

Deciding the satisfiability of a formula ϕ then reduces to decide the emptiness of $\mathcal{A}_\phi$, which can be done in linear time w.r.t. the number of states.

Consider however that $\mathcal{A}_\phi$ could have, in principle, a number of states which is a tower of exponentials whose height is the number of quantifier alternations in ϕ. This is because each quantifier alternation may require a projection followed by a complement and the projection may yield a non-deterministic automaton, requiring an exponential blow-up for the complement. Whether this multi-exponential blow-up may occur in the above construction is an open question. (Though the complexity of Presburger arithmetic is known and elementary, using other methods. See [FR79].)

2.9 Typing Constraints and Tree Automata

If we want to solve constraints on terms, we need another device, which accepts not only words, but also trees. We introduce tree automata on a very simple constraint system. Some other applications are sketched in sections 2.11.1, 2.11.2, 2.11.4. We refer the interested reader to [CDG$^+$97], a book freely available on the web, which contains much more material, both on theory and on applications to tree automata to constraint solving.

2.9.1 Tree Automata

Recognizing a tree is not very different from recognizing a word: a tree t is accepted if there is a successful run of the automaton on t. The main difference is that transitions send some (possibly different) states to all the successors of a node instead of sending a single state to the next position.

Here, we will only consider automata on finite trees, though there are very interesting developments and applications in the case of infinite trees (see e.g. [Tho90]).

We assume that $\mathcal{F}$ is a finite ranked alphabet (each symbol has a fixed arity) and $\mathcal{T}(\mathcal{F})$ is the set of (ground) terms built on $\mathcal{F}$. A finite (bottom-up) tree automaton is a triple (Q, Q_f, δ) where Q is a finite set of states, Q_f is a subset of final states and δ is a set of rules of the form $f(q_1, \dots, q_n) \to q$ where $q_1, \dots, q_n, q \in Q$ and $f \in \mathcal{F}$ has arity n. A *run* ρ of $\mathcal{A}$ on a term (or tree) $t \in \mathcal{T}(\mathcal{F})$ is a tree which has the same set of positions as t, whose labels are in Q and such that, for every position p of t, if $t(p) = f$ has arity n, $\rho(p) = q$ and $\rho(p \cdot 1) = q_1, \dots \rho(p \cdot n) = q_n$, then there is a transition $f(q_1, \dots, q_n) \to q$ in δ. A run is *successful* if its root is labeled with a finite state. A term t is *recognized* by a tree automaton $\mathcal{A}$ if there is a successful run of $\mathcal{A}$ on t.

Example 13. Using a notation which is more common in algebraic specifications, we define below an order-sorted signature:

SORTS:$\mathsf{nat}, \mathsf{int}$
SUBSORTS : $\mathsf{nat} \le \mathsf{int}$
FUNCTION DECLARATIONS:

$$\begin{array}{lr}
0 : & \to \mathsf{nat} \\
+ : & \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat} \\
s : & \mathsf{nat} \to \mathsf{nat} \\
p : & \mathsf{nat} \to \mathsf{int} \\
+ : & \mathsf{int} \times \mathsf{int} \to \mathsf{int} \\
abs : & \mathsf{int} \to \mathsf{nat} \\
fact : & \mathsf{nat} \to \mathsf{nat} \\
\dots &
\end{array}$$

This can be seen as a finite tree automaton whose states are the sorts $\{\mathsf{nat}, \mathsf{int}\}$. Function declarations can easily be translated into automata rules, e.g. $+(\mathsf{nat}, \mathsf{nat}) \rightarrow \mathsf{nat}$. The subsort ordering corresponds to ϵ-transitions which can easily be eliminated, yielding

$$\begin{array}{rlrl}
0 & \rightarrow \mathsf{nat} & 0 & \rightarrow \mathsf{int} \\
+(\mathsf{nat}, \mathsf{nat}) & \rightarrow \mathsf{nat} & +(\mathsf{nat}, \mathsf{nat}) & \rightarrow \mathsf{int} \\
s(\mathsf{nat}) & \rightarrow \mathsf{nat} & s(\mathsf{nat}) & \rightarrow \mathsf{int} \\
p(\mathsf{nat}) & \rightarrow \mathsf{int} & +(\mathsf{int}, \mathsf{int}) & \rightarrow \mathsf{int} \\
abs(\mathsf{int}) & \rightarrow \mathsf{nat} & abs(\mathsf{int}) & \rightarrow \mathsf{int} \\
fact(\mathsf{nat}) & \rightarrow \mathsf{nat} & fact(\mathsf{nat}) & \rightarrow \mathsf{int} \\
& \dots & &
\end{array}$$

The set of terms accepted by the automaton in state nat (resp int) is the set of terms of *sort* nat.

Basically, recognizable tree languages have the same properties as recognizable word languages: closure by Boolean operations, decidability of emptiness, finiteness etc... See [CDG⁺97] for more details.

2.9.2 Membership Constraints

The set of *sort expressions* $\mathcal{SE}$ is the least set such that

- $\mathcal{SE}$ contains a finite set of *sort symbols* S, including the two particular symbols $\top_S$ and $\bot_S$.
- If $s_1, s_2 \in \mathcal{SE}$, then $s_1 \vee s_2$, $s_1 \wedge s_2$, $\neg s_1$ are in $\mathcal{SE}$
- If $s_1, \dots, s_n$ are in $\mathcal{SE}$ and f is a function symbol of arity n, then $f(s_1, \dots, s_n) \in \mathcal{SE}$.

The atomic formulas are expressions $t \in s$ where $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $s \in \mathcal{SE}$. The formulas are arbitrary first-order formulas built on these atomic formulas.

These formulas are interpreted as follows: we are giving an order-sorted signature (or a tree automaton) whose set of sorts is S. We define the interpretation$[\![\cdot]\!]_S$ of sort expressions as follows:

- if $s \in S$, $[\![s]\!]_S$ is the set of terms in $\mathcal{T}(\mathcal{F})$ that are accepted in state s.
- $[\![\top_S]\!]_S = \mathcal{T}(\mathcal{F})$ and $[\![\bot_S]\!]_S = \emptyset$
- $[\![s_1 \vee s_2]\!]_S = [\![s_1]\!]_S \cup [\![s_2]\!]_S$, $[\![s_1 \wedge s_2]\!]_S = [\![s_1]\!]_S \cap [\![s_2]\!]_S$, $[\![\neg s]\!]_S = \mathcal{T}(\mathcal{F}) \setminus [\![s]\!]_S$
- $[\![f(s_1, \dots, s_n)]\!]_S = \{f(t_1, \dots, t_n) \mid t_1 \in [\![s_1]\!]_S, \dots t_n \in [\![s_n]\!]_S\}$

Example 14. Consider the specification:

$$\begin{array}{rlrl}
0 & \rightarrow \mathsf{even} & s(\mathsf{odd}) & \rightarrow \mathsf{even} \\
s(\mathsf{even}) & \rightarrow \mathsf{odd} & \mathsf{even} + \mathsf{odd} & \rightarrow \mathsf{odd} \\
\mathsf{odd} + \mathsf{even} & \rightarrow \mathsf{odd} & \mathsf{even} + \mathsf{even} & \rightarrow \mathsf{even} \\
\mathsf{odd} + \mathsf{odd} & \rightarrow \mathsf{even} & &
\end{array}$$

$[\![\mathsf{odd} \wedge \mathsf{even}]\!]$ is empty: this can be shown by constructing the intersection automaton whose rules only yielding $\mathsf{odd} \wedge \mathsf{even}$ consist of

$$\begin{array}{rr} s(\mathsf{odd} \wedge \mathsf{even}) \rightarrow \mathsf{odd} \wedge \mathsf{even} & \mathsf{odd} \wedge \mathsf{even} + \mathsf{even} \rightarrow \mathsf{odd} \wedge \mathsf{even} \\ \mathsf{odd} \wedge \mathsf{even} + \mathsf{odd} \rightarrow \mathsf{odd} \wedge \mathsf{even} & \mathsf{odd} + \mathsf{odd} \wedge \mathsf{even} \rightarrow \mathsf{odd} \wedge \mathsf{even} \\ \mathsf{even} + \mathsf{odd} \wedge \mathsf{even} \rightarrow \mathsf{odd} \wedge \mathsf{even} & \end{array}$$

which shows that $\mathsf{odd} \wedge \mathsf{even}$ is unreachable.

2.9.3 From the Constraints to the Automata

It is easy to associate with each membership constraint $x \in \zeta$ a tree automaton which accepts the solutions of the constraint: simply consider an automaton for $[\![\zeta]\!]$. Now, for arbitrary constraints, we need a decomposition lemma showing how $f(t_1, \ldots, t_n) \in \zeta$ can be decomposed into a finite disjunction of constraints $t_1 \in \zeta_1 \wedge \ldots \wedge t_n \in \zeta_n$. This is not difficult (though a bit long, see [CD94, Com98] for more details and extensions). Then every atomic constraint is a membership constraint $x \in \zeta$. And we use the closure properties of tree automata to get an automaton which accepts the solutions of an arbitrary constraint.

Note that, in this very simple example of application, we do not need to stack (or, more realistically, to overlap) the different tracks, since variables are "independent". In other words, the resulting automaton, which accepts tuples of terms, is actually a product of tree automata (each working on a single track). This is not the case for more complicated constraints such as rewriting constraints (see [CDG$^+$97, DT90]). For instance the solutions of $x = y$ are recognized by a finite automaton on pairs (simply check that all labels are pairs of identical symbols) but is not recognized by a product of two automata.

2.9.4 The Monadic Second Order Logics

The constraint system of the previous section can be embedded in a well-known more powerful system: the weak monadic second-order logic. It is beyond the scope of these notes to study these logics. Let us simply briefly recall the constraint systems and the main results.

Atomic formulas are $x = y \cdot i$ where $i \in \{1, ...n\}$, $x = y$, $x = \epsilon$ or membership constraints $x \in X$ where x, y, X are variables. Upper case letters are used here for monadic second order variables, while lower case letters are first-order variables. The formulas are built over these atomic constraints using Boolean connectives and quantifications, both over the first-order and the second-order variables.

The formulas are interpreted as follows: first-order variables range over words in $\{1, ..., n\}^*$ (ϵ is the empty word) and second-order variables range over finite (resp. finite or infinite) subsets of $\{1, ..., n\}^*$. In the finite (resp.

finite or infinite) case, the constraint system is called *weak monadic second order logic with n successors* (resp. *monadic second order logic with n successors*), $WSnS$ for short (resp. SnS).

The main results state on one hand that the solutions of atomic formulas are recognized by appropriate tree automata and on the other hand that the class of automata is closed under Boolean operations and projections and emptiness is decidable. This yields the decidability of the constraint system.

More precisely, we use an encoding similar to the encoding which was presented in section 2.8: with each k-uple of subsets of $\{1, ..., n\}^*$, we associate an infinite tree of degree n labeled with vectors in $\{0,1\}^k$. Assume for instance that there are three sets S_1, S_2, S_3. If a path $w \in \{1, ..., n\}^*$ of the tree yields a node labeled with e.g. $\begin{smallmatrix}1\\0\\1\end{smallmatrix}$, this means that $w \in S_1$, $w \notin S_2$ and $w \in S_3$. Hence solutions can be seen as trees labeled with $\{0,1\}^k$. Then

Theorem 7 ([TW68]). *The set of solutions of a constraint in WSnS is accepted by a finite tree automaton.*

Theorem 8 ([Rab69]). *The set of solution of a constraint in SnS is accepted by a Rabin tree automaton.*

Both results are shown by first proving that the solutions of atomic constraints are recognized by an appropriate automaton and then proving the closure properties of the class of automata.

Such closure properties are easy, except in one case which is really non trivial: the complement for Rabin automata. We did not define so far Rabin automata (see e.g. [Tho97, Rab77] for surveys). They work on infinite trees, hence top-down. On the other hand, it is not possible to determinize top-down tree automata (this is left as an exercise). Hence the classical complementation construction does not work for automata on infinite trees.

Such general results can be applied to several constraint systems which can be translated into monadic second-order logics. A typical example is rewriting constraints (see [DHLT88]). A more detailed description of applications can be found in [CDG+97].

2.9.5 Extensions of the Constraint System and Further Results

The simple constraint system of section 2.9.2 can be extended in several directions. For instance, it is possible to add equations between first-order terms [CD94] or to consider (in some restricted way) second-order terms [Com98].

Solving constraints which combine equations and membership predicates is also known as *order-sorted unification* and deserved several works which use syntactic constraint solving methods [MGS90, HKK98].

The membership constraints have also been extended to other models with applications to automated deduction (see e.g. [GMW97, JMW98]). The device (finite tree automata) has however to be extended, along the lines described in section 2.11.4.

Parametric specifications can also be handled, but we need then the set constraints of the next section, since we have also to find which values of the parametric types are solutions of the constraint.

Tree automata (on infinite trees) are used in [TW93] in the context of type reconstruction. Here constraints are interpreted over the type structures, not over the term structure.

2.10 Set Constraints and Tree Set Automata

There has been recently a lot of work on set constraints and, in particular on applications of automata techniques. Let us cite among others [BGW93, GTT93b, GTT93a, GTT94, Tom94].

We do not intend to survey the results. Our purpose is to show how automata may help to understand and solve such constraints.

2.10.1 Set Constraints

We consider only the simplest version of set constraints. The *set expressions* are built from set variables, intersection, union, complement and application of a function symbol (out of a finite ranked alphabet) to set expressions. Then the constraints are conjunctions of inclusion constraints $e \subseteq e'$ where e, e' are set expressions.

Set expressions are interpreted as subsets of $\mathcal{T}(\mathcal{F})$: intersection, union and complement are interpreted as expected. If f is a function symbol, then $f(e_1, \dots, e_n)$ is interpreted as the set of terms $f(t_1, \dots, t_n)$ where, for every i, t_i is in the interpretation of e_i.

Example 15. In this example, we define the lists L_S of elements in S as a solution of the classical fixed point equations. Since S is arbitrary, we consider another instance of the definition, where S is replaced with L_S itself. Then we are looking for the sets of trees S such that the lists over S are also lists of lists of elements in S.

$$\begin{aligned} L_S &= nil \cup cons(S, L_S) \\ L_{L_S} &= nil \cup cons(L_S, L_{L_S}) \\ N &= 0 \cup s(N) \\ L_S &= L_{L_S} \end{aligned}$$

A solution of this constraint is the set $S = \{nil\}$ since L_S then consists of lists of any number of *nil*'s which are also lists of lists of *nil*'s. What are all the solutions?

2.10.2 Tree Set Automata

We follow here the definitions of [Tom94]. Extensions can be found in e.g. [CDG$^+$97].

A *tree set automaton* (on the ranked alphabet $\mathcal{F}$) is a tuple (Q, A, Ω, δ) where Q is a finite set of states, A is a tuple $(A_1, \dots, A_m)$ of final states, $\Omega \subseteq 2^Q$ is a set of accepting sets of states and δ is a set of rules $f(q_1, \dots, q_n) \to q$ where $f \in \mathcal{F}$ has arity n and $q_1, \dots, q_n, q$ are states.

A *run* of a tree set automaton is a mapping from $\mathcal{T}(\mathcal{F})$ into Q such that, if $f(t_1, \dots, t_n) \in \mathcal{T}(\mathcal{F})$, then there is a rule $f(\rho(t_1), \dots, \rho(t_n)) \to \rho(f(t_1, \dots, t_n))$ in δ. A run is *successful* if $\rho(\mathcal{T}(\mathcal{F})) \in \Omega$.

The *language recognized* by a tree set automaton is the set of tuples of sets $(L_1, \dots, L_m)$ such that there is a successful run ρ such that, for every i, $L_i = \{t \in \mathcal{T}(\mathcal{F}) \mid \rho(t) \in F_i\}$.

We will see an example below.

2.10.3 From Constraints to Automata

We consider a single constraint $e = e'$ (the case $e \subseteq e'$ is similar). Let Q be the set of mappings from the set of subexpressions (with top symbol in $\mathcal{F}$ or $\mathcal{X}$) occurring in c into $\{0, 1\}$. Such mappings are extended to all subexpressions using the usual Boolean rules. They are represented as bit-vectors. For convenience (in the BDD style) we use an x instead of $0, 1$ when the value is not relevant (i.e. both rules are present, for each Boolean value). If there are n free variables $X_1, \dots, X_n$, then $A = (A_1, \dots, A_n)$ with $A_i = \{\phi \in Q \mid \phi(X_i) = 1\}$. δ consists of the rules $f(\phi_1, \dots, \phi_k) \to \phi$ such that, for every subexpression e which is not a variable, $\phi(e) = 1$ iff $(e = f(e_1, \dots, e_k)$ and $\forall i$, $\phi_i(e_i) = 1)$. Ω is the set of $\omega \subseteq Q$ such that, for every $\phi \in \omega$, $\phi(e) = 1$ if and only if $\phi(e') = 1$.

Example 16. Consider first the equation $L_S = nil \cup cons(S, L_S)$. We have $Q = 2^{\{S, L_S, cons(S, L_S), nil\}}$. The transitions are

$$\begin{aligned} nil &\to (\mathsf{x}, \mathsf{x}, 0, 1) \\ cons((1, \mathsf{x}, \mathsf{x}, \mathsf{x}), (\mathsf{x}, 1, \mathsf{x}, \mathsf{x})) &\to (\mathsf{x}, \mathsf{x}, 1, 0) \\ f(...) &\to (\mathsf{x}, \mathsf{x}, 0, 0) \text{ for any } f \notin \{cons, nil\} \end{aligned}$$

Ω consists of the subsets of

$$\bigcup_{x_1, x_2, x_3, x_4 \in \{0,1\}} \{(x_1, 0, 0, 0), (x_2, 1, 0, 1), (x_3, 1, 1, 0), (x_4, 1, 1, 1)\}.$$

Consider for instance the following mappings from $\mathcal{T}(\mathcal{F})$ to Q:

$\rho_1(t) = (0, 0, 0, 0)$ for all t
$\rho_2(t) = (0, 0, 0, 0)$ for all $t \neq nil$ and $\rho_2(nil) = (0, 0, 0, 1)$
$\rho_3(t) = (0, 0, 0, 0)$ for all $t \neq nil$ and $\rho_2(nil) = (0, 1, 0, 1)$

ρ_1 is not a run since it is not compatible with the first transition rule. ρ_2 is a run which is not successful since $(0,0,0,1) \in \rho_2(\mathcal{T}(\mathcal{F}))$ and $(0,0,0,1) \notin \omega$ for any $\omega \in \Omega$. ρ_3 is a successful run.

Let $A_1 = \{(1,\mathsf{x},\mathsf{x},\mathsf{x})\}$, $A_2 = \{(\mathsf{x},1,\mathsf{x},\mathsf{x})\}$. For instance $(\emptyset, \{nil\})$ is accepted by the automaton, as well as $(T(\{cons, nil\}), T(\{cons, nil\}))$.

Exercise 4. In the previous example, what are all the successful runs such that $\rho(\mathcal{T}(\mathcal{F}))$ is minimal (w.r.t. inclusion)?

Note that we may simplify the construction: we may only consider states which belong to some $\omega \in \Omega$: otherwise, we will not get a successful run. For instance in our example, the set of rules may be restricted to

$$\begin{aligned} nil &\to (\mathsf{x},1,0,1) \\ cons((1,\mathsf{x},\mathsf{x},\mathsf{x}),(\mathsf{x},1,\mathsf{x},\mathsf{x})) &\to (\mathsf{x},1,1,0) \\ f(...) &\to (\mathsf{x},0,0,0) \text{ for every } f \notin \{cons, nil\} \end{aligned}$$

Another remark is that Ω is subset closed, when it is derived from a (positive) set constraint: if $\omega \in \Omega$ and $\omega' \subseteq \omega$, then $\omega' \in \Omega$. Tree set automata which have this property are called *simple*. There is yet another particular property of tree set automata which can be derived from the construction: once we fix the components of the state which correspond to set variables, the rules becomes deterministic. In other words, if two rules applied to t yield states q_1, q_2, then q_1, q_2 differ only in the components corresponding to set variables. Such automata belong to a restricted class called *deterministic tree set automata*. The above construction shows that:

Proposition 9. *The solutions of an atomic set constraints are recognized by a deterministic simple tree set automaton.*

Now we may take advantage of closure of tree set automata:

Theorem 9 ([Tom94, GTT94]). *The class of languages recognized by tree set automata is closed by intersection, union, projection, inverse projection.*

The class of languages recognized by simple deterministic tree set automata is closed under all Boolean operations.

as well as decision properties:

Theorem 10 ([Tom94, GTT94]). *Emptiness decision is NP-complete for the class of (simple) tree set automata.*

Finally, if there is a solution (tuples of sets of terms) to a set constraint, then there is one which consists of regular sets only.

Example 17. Let us come back to example 15.
$Q = 2^{\{S, L_S, L_{L_S}, nil, cons(S, L_S), cons(L_S, L_{L_S})\}}$ and
$\Omega = 2^{\{(\mathsf{X},0,0,0,0,0),(\mathsf{X},1,1,0,1,1),(\mathsf{X},1,1,1,0,0)\}}$.

Since we may only consider rules which belong to some member of Ω, there are actually few rules:

$$\begin{array}{rl}
nil & \rightarrow (\mathsf{x}, 1, 1, 1, 0, 0) \\
cons((1, 1, \mathsf{x}, \mathsf{x}, \mathsf{x}, \mathsf{x}), (\mathsf{x}, 1, 1, \mathsf{x}, \mathsf{x}, \mathsf{x})) & \rightarrow (\mathsf{x}, 1, 1, 0, 1, 1) \\
cons((0, 0, 0, 0, 0, 0), (\mathsf{x}, \mathsf{x}, \mathsf{x}, \mathsf{x}, \mathsf{x}, \mathsf{x})) & \rightarrow (\mathsf{x}, 0, 0, 0, 0, 0) \\
cons((\mathsf{x}, \mathsf{x}, \mathsf{x}, \mathsf{x}, \mathsf{x}, \mathsf{x}), (\mathsf{x}, 0, 0, 0, 0, 0)) & \rightarrow (\mathsf{x}, 0, 0, 0, 0, 0) \\
f(...) & \rightarrow (\mathsf{x}, 0, 0, 0, 0, 0)
\end{array}$$

Let ρ be the mapping such that $\rho(t) = (1, 1, 1, 0, 1, 1)$ if $t \in T(\{nil, cons\})$, $\rho(nil) = (0, 1, 1, 1, 0, 0)$ and $\rho(t) = (0, 0, 0, 0, 0, 0)$ otherwise. ρ is a successful run.

Exercise 5. What is the solution corresponding to the run ρ? Are there any other successful runs?

2.11 Examples of Other Constraint Systems Using Tree Automata

2.11.1 Rigid E-Unification

A unification problem is a conjunction of equations between terms (as previously seen). Given a set of equations E, a solution of the rigid E-unification problem P is a (ground) substitution σ such that $E\sigma \vdash P\sigma$. Note that, in contrast with E-unification, only one instance of each equation in E can be used in the proof. Such problems play crucial roles in automated deduction (see e.g. [GRS87, DMV96]).

The crucial lemma in [Vea97] is that the set of solutions of a single rigid E-unification problem with one variable is recognized by a finite tree automaton. The following result is then a consequence (at least the EXPTIME membership), computing the intersection of n tree automata:

Theorem 11 ([Vea97]). *The simultaneous rigid E-unification problem with one variable is EXPTIME-complete.*

There are other applications of tree automata and tree grammars to unification problems, see for instance [LR97] and there are also other applications of tree automata to related problems (e.g. [CGJV99])

2.11.2 Higher-Order Matching Problems

We consider here simply typed λ terms. A *higher-order matching problem* is a (conjunction of) equation(s) $s = t$ where s, t are simply-typed λ-terms built over an alphabet of typed constants and variables, such that t does not contain any free variables. A solution consists in an assignment σ to the free variables of s such that $s\sigma$ and t are β-equivalent. This decidability of higher-order matching is still an open question. It has been shown decidable at order 3 [Dow93] and at order 4 [Pad96]; the *order* of a type is defined by $o(a) = 1$ if a is an atomic type and $o(\tau_1, \ldots, \tau_n \to \tau) = \max(o(\tau), 1+o(\tau_1), ..., 1+o(\tau_n))$. The order of a problem is the maximal order of the type of a free variable occurring in it.

An *interpolation equation* is a higher-order matching problem of the form $x(t_1, \ldots, t_n) = u$ where $t_1, \ldots, t_n, u$ do not contain any free variables. As shown in [Pad96], a matching problem of order n is equivalent to a Boolean combination of interpolation equations of order n.

The crucial property in [CJ97b], as before, is that the set of solutions of an interpolation equation $x(t_1, \ldots, t_n) = u$ where x is the only free variable, is recognized by a (slight variant of a) finite tree automaton, provided that the order of x is at most 4. It follows, thanks to the closure properties of tree automata that:

Theorem 12 ([CJ97b]). *The set of solutions of a 4th order matching problem is recognized by a (effectively computable) finite tree automaton.*

This implies of course the decidability of 4th order matching.

2.11.3 Feature Constraints

We mention here an application to feature constraints [MN98]. Since feature constraints are interpreted over infinite trees, we need a device which works on infinite trees. It is beyond the scope of this course to introduce automata on infinite objects (see e.g. [Tho90]).

The constraints consist of conjunctions of atomic constraints which are of the form $x \leq x'$, $x[f]x'$ or $a(x)$.

A *feature tree* is a finite or infinite tree in which the order of the sons is irrelevant, but each son of a node is labeled with a *feature* and each feature labels at most one son of each node (see e.g. [Tre97] for more details). Feature trees have been introduced as a record-like data structure in constraint logic programming (see e.g. [ST94]) and are also used as models in computational linguistics. The reader is referred to [Tre97] for more references of the subject.

The semantics of the constraints is given by an interpretation over feature trees, inequalities mean roughly that a tree is "more defined" than another tree. For instance it may include new features. $x[f]x'$ is satisfied by a pair of trees (t, t') if t' is a son of t, at feature f. $a(x)$ is satisfied by a tree t if the root of t is labeled with a.

The crucial part of the result in [MN98] is the following:

Theorem 13 ([MN98]). *The set of solutions of an constraint over sufficiently labeled feature trees is recognized by a finite tree automaton.*

Actually, the authors show a translation into some monadic second order logic; this is an equivalent formulation thanks to Rabin's theorem [Rab69] (see also [Tho97] for a comprehensive proof), or Thatcher and Wright's theorem if one considers finite trees only [TW68] (see also [CDG$^+$97]).

It follows for instance from this theorem that the entailment of existentially quantified feature constraints is decidable.

2.11.4 Encompassment Constraints

Encompassment constraints are built on arbitrarily many atomic predicates of the form $\mathit{encomp}_t(x)$. Where t is a term. Such a predicate is interpreted as the set of terms which *encompass* t. A term u encompasses v when an instance of v is a subterm of u.

It is known for a long time that, when t is a linear term (i.e. each variable occurs at most once in t), then the solutions $\mathit{encomp}_t(x)$ are recognized by a finite tree automaton, hence the first-order theory of such predicates is decidable, thanks to closure properties of tree automata.

When t is not linear, we need an extension of tree automata. Such an extension is defined and studied in [DCC95]: a *reduction automaton* is defined as a finite tree automaton, except that each transition may check equalities and/or disequalities between subtrees at fixed positions. For instance a rule $f(q_1, q_2) \xrightarrow{12=21 \wedge 11 \neq 22} q$ states that the transition can be applied to $f(t_1, t_2)$ only if t_1 can reach q_1, t_2 can reach q_2 and, moreover, the subterm of t_1 at position 2 equals the subterm of t_2 at position 1 and the subterm of t_1 at position 1 is distinct from the subterm of t_2 at position 2. Such a computation model is, in general, too expressive. Hence reduction automata also impose an ordering on the states such that, if there is at least one equality checked by the transition, then the state is decreasing. For instance in the above rule, we must have $q_1 > q$ and $q_2 > q$.

We follow our leitmotiv: first it is possible to associate each atomic constraint with a reduction automaton which accepts all its solutions, then we have the closure and decision properties for these automata:

Theorem 14 ([DCC95]).

1. *There is a (effectively computable) deterministic and complete reduction automaton which accepts the solutions of* $\mathit{encomp}_t(x)$.
2. *The class of languages accepted by deterministic reduction automata is effectively closed under Boolean operations.*
3. *The emptiness problem for reduction automata is decidable.*

It follows in particular that the first-order theory of encompassment predicates is decidable. This implies in particular the decidability of a well-known problem in rewriting theory: the ground reducibility, which can be expressed as a formula in this theory.

There are a lot of further works on automata with equality and disequality tests, e.g. [BT92,LM94,CCC+94,CJ97a,JMW98]. See [CDG+97] for a survey of the main results and applications.

Acknowledgments

This document beneficiated of works and papers written with several colleagues, in particular Jean-Pierre Jouannaud, Hélène Kirchner and Christophe Ringeissen.

References

[AK92] M. Adi and Claude Kirchner. Associative commutative matching based on the syntacticity of the ac theory. In Franz Baader, J. Siekmann, and W. Snyder, editors, *Proceedings 6th International Workshop on Unification, Dagstuhl (Germany)*. Dagstuhl seminar, 1992.

[Baa91] F. Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. In R. V. Book, editor, *Proceedings 4th Conference on Rewriting Techniques and Applications, Como (Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 86–97. Springer-Verlag, April 1991.

[Büc60] J.R. Büchi. On a decision method in restricted second-order arithmetic. In E. Nagel et al., editor, *Proc. Int. Congr. on logic, methodology and philosophy of science*, Standford, 1960. Stanford Univ. Press.

[BC96] Alexandre Boudet and Hubert Comon. Diophantine equations, Presburger arithmetic and finite automata. In H. Kirchner, editor, *Proc. Coll. on Trees in Algebra and Programming (CAAP'96)*, Lecture Notes in Computer Science, 1996.

[BCD+98] Peter Borovanský, Horatiu Cirstea, Hubert Dubois, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. ELAN *V 3.3 User Manual.* LORIA, Nancy (France), third edition, December 1998.

[BGLS95] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[BGW93] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Set constraints are the monadic class. In *Proc. 8th IEEE Symp. Logic in Computer Science, Montréal*, 1993.

[BHSS89] H.-J. Bürckert, A. Herold, and M. Schmidt-Schauß. On equational theories, unification and decidability. *Journal of Symbolic Computation*, 8(1 & 2):3–50, 1989. Special issue on unification. Part two.

[BKK+98] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the second International Workshop on Rewriting Logic and Applications*, volume 15, `http://www.elsevier.nl/locate/entcs/volume16.html`, Pont-à-Mousson (France), September 1998. Electronic Notes in Theoretical Computer Science.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and all That*. Cambridge University Press, 1998.

[Boc87] A. Bockmayr. A note on a canonical theory with undecidable unification and matching problem. *Journal of Automated Reasoning*, 3(1):379–381, 1987.

[BS86] R. V. Book and J. Siekmann. On unification: Equational theories are not bounded. *Journal of Symbolic Computation*, 2:317–324, 1986.

[BS99] F. Baader and W. Snyder. Unification theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*. Elsevier Science Publishers, 1999. To appear.

[BT92] B. Bogaert and Sophie Tison. Equality and disequality constraints on brother terms in tree automata. In A. Finkel, editor, *Proc. 9th Symp. on Theoretical Aspects of Computer Science*, Paris, 1992. Springer-Verlag.

[Bür89] H.-J. Bürckert. Matching — A special case of unification? *Journal of Symbolic Computation*, 8(5):523–536, 1989.

[BVW94] O. Bernholtz, M. Vardi, and P. Wolper. Au automata-theoretic approach to branching-time model checking. In *Proc. 6th Int. Conf. on Computer Aided verification*, volume 818 of *Lecture Notes in Computer Science*. Springer Verlag, 1994.

[Cas98] Carlos Castro. Building Constraint Satisfaction Problem Solvers Using Rewrite Rules and Strategies. *Fundamenta Informaticae*, 34:263–293, September 1998.

[CCC$^+$94] A.-C. Caron, H. Comon, J.-L. Coquidé, M. Dauchet, and F. Jacquemard. Pumping, cleaning and symbolic constraints solving. In *Proc. Int. Conference on Algorithms, Languages and Programming*, Lecture Notes in Computer Science, vol. 820, Jerusalem, July 1994. Springer-Verlag.

[CCD93] Anne-Cécile Caron, Jean-Luc Coquide, and Max Dauchet. Encompassment properties and automata with constraints. In Claude Kirchner, editor, *Rewriting Techniques and Applications, 5th International Conference, RTA-93*, LNCS 690, pages 328–342, Montreal, Canada, June 16–18, 1993. Springer-Verlag.

[CD94] Hubert Comon and Catherine Delor. Equational formulae with membership constraints. *Information and Computation*, 112(2):167–216, August 1994.

[CDG$^+$97] H. Comon, M. Dauchet, R. Gilleron, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. A preliminary version of this unpublished book is available on `http://l3ux02.univ-lille3.fr/tata` , 1997.

[CDJK99] Hubert Comon, Mehmet Dincbas, Jean-Pierre Jouannaud, and Claude Kirchner. A methodological view of constraint solving. *Constraints*, 4(4):337–361, December 1999.

[CF92] H. Comon and M. Fernández. Negation elimination in equational formulae. In *Proc. 17th International Symposium on Mathematical Foundations of Computer Science (Praha)*, Lecture Notes in Computer Science, 1992.

[CGJV99] V. Cortier, H. Ganzinger, F. Jacquemard, and V Veanes. Decidable fragments of simultaneous rigid reachability. In *Proc. ICALP'99*, 1999. To appear in ICALP'99.

[Cha94] Jacques Chabin. *Unification Générale par Surréduction Ordonnée Contrainte et Surréduction Dirigée.* Thèse de Doctorat d'Université, Université d'Orléans, January 1994.

[CHJ94] Hubert Comon, Marianne Haberstrau, and Jean-Pierre Jouannaud. Syntacticness, cycle-syntacticness and shallow theories. *Information and Computation*, 111(1):154–191, May 1994.

[Chu62] A. Church. Logic, arithmetic, automata. In *Proc. International Mathematical Congress*, 1962.

[CJ94] H. Comon and F. Jacquemard. Ground reducibility and automata with disequality constraints. In P. Enjalbert, E. W. Mayr, and K. W. Wagner, editors, *Proceedings 11th Annual Symposium on Theoretical Aspects of Computer Science, Caen (France)*, volume 775 of *Lecture Notes in Computer Science*, pages 151–162. Springer-Verlag, February 1994.

[CJ97a] Hubert Comon and Florent Jacquemard. Ground reducibility is exptime-complete. In *Proc. IEEE Symp. on Logic in Computer Science*, Varsaw, June 1997. IEEE Comp. Soc. Press.

[CJ97b] Hubert Comon and Yan Jurski. Higher-order matching and tree automata. In M. Nielsen and W. Thomas, editors, *Proc. Conf. on Computer Science Logic*, volume 1414 of *LNCS*, pages 157–176, Aarhus, August 1997. Springer-Verlag.

[CL89] H. Comon and P. Lescanne. Equational problems and disunification. *Journal of Symbolic Computation*, 7(3 & 4):371–426, 1989. Special issue on unification. Part one.

[Col82] A. Colmerauer. PROLOG II, manuel de référence et modèle théorique. Technical report, GIA, Université Aix-Marseille II, 1982.

[Com86] H. Comon. Sufficient completeness, term rewriting system and anti-unification. In J. Siekmann, editor, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, volume 230 of *Lecture Notes in Computer Science*, pages 128–140. Springer-Verlag, 1986.

[Com88] H. Comon. *Unification et disunification. Théories et applications.* Thèse de Doctorat d'Université, Institut Polytechnique de Grenoble (France), 1988.

[Com89] Hubert Comon. Inductive proofs by specification transformations. In Nachum Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, pages 76–91, Chapel Hill, NC, April 1989. Vol. 355 of *Lecture Notes in Computer Science*, Springer, Berlin.

[Com90] H. Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Sciences*, 1(4):387–411, 1990.

[Com91] H. Comon. Disunification: a survey. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 9, pages 322–359. The MIT press, Cambridge (MA, USA), 1991.

[Com93] Hubert Comon. Complete axiomatizations of some quotient term algebras. *Theoretical Computer Science*, 118(2):167–191, September 1993.

[Com98] H. Comon. Completion of rewrite systems with membership constraints. Part II: Constraint solving. *Journal of Symb. Computation*, 25:421–453, 1998. This is the second part of a paper whose abstract appeared in Proc. ICALP 92, Vienna.

[CT94] Hubert Comon and Ralf Treinen. Ordering constraints on trees. In Sophie Tison, editor, *Colloquium on Trees in Algebra and Programming*, volume 787 of *Lecture Notes in Computer Science*, pages 1–14, Edinburgh, Scotland, April 1994. Springer Verlag.

[Dau94] M. Dauchet. Symbolic constraints and tree automata. In Jouannaud [Jou94], pages 217–218.

[DCC95] Dauchet, Caron, and Coquidé. Automata for reduction properties solving. *JSCOMP: Journal of Symbolic Computation*, 20, 1995.

[DH95] N. Dershowitz and C. Hoot. Natural termination. *Theoretical Computer Science*, 142(2):179–207, May 1995.

[DHK98] Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. Rapport de Recherche 3400, Institut National de Recherche en Informatique et en Automatique, April 1998. `ftp://ftp.inria.fr/INRIA/publication/RR/RR-3400.ps.gz`.

[DHLT88] Max Dauchet, Thierry Heuillard, Pierre Lescanne, and Sophie Tison. The confluence of ground term rewriting systems is decidable. In *Proc. 3rd IEEE Symp. Logic in Computer Science, Edinburgh*, 1988.

[DJ90a] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 6, pages 244–320. Elsevier Science Publishers B. V. (North-Holland), 1990.

[DJ90b] D. Dougherty and P. Johann. An improved general E-unification method. In M. E. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449 of *Lecture Notes in Computer Science*, pages 261–275. Springer-Verlag, July 1990.

[DKM84] C. Dwork, P. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[DMR76] M. Davis, Y. Matijasevič, and J. A. Robinson. Hilbert's tenth problem: Positive aspects of a negative solution. In *F. E. Browder, Editor, Mathematical Developments Arising from Hilbert Problems, American Mathematical Society*, pages 323–378, 1976.

[DMV96] Anatoli Degtyarev, Yuri Matiyasevich, and Andrei Voronkov. Simultaneous reigid e-unification and related algorithmic problems. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 494–502. IEEE Comp. Soc. Press, 1996.

[Dow93] Gilles Dowek. Third order matching is decidable. *Annals of Pure and Applied Logic*, 1993.

[DT90] Max Dauchet and Sophie Tison. The theory of ground rewrite systems is decidable. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, 1990.

[Eke95] Steven Eker. Associative-commutative matching via bipartite graph matching. *Computer Journal*, 38(5):381–399, 1995.

[Elg61] C.C. Elgot. Decision problems of finite automata design and related arithmetics. *Trans. Amer. Math. Soc.*, 98:21–52, 1961.

[Fay79] M. Fay. First order unification in equational theories. In *Proceedings 4th Workshop on Automated Deduction, Austin (Tex., USA)*, pages 161–167, 1979.

[Fer92] M. Fernández. Narrowing based procedures for equational disunification. *Applicable Algebra in Engineering, Communication and Computation*, 3:1–26, 1992.

[Fer98] M. Fernández. Negation elimination in empty or permutative theories. *J. Symbolic Computation*, 26(1):97–133, July 1998.

[FH86] F. Fages and G. Huet. Complete sets of unifiers and matchers in equational theories. *Theoretical Computer Science*, 43(1):189–200, 1986.

[FR79] Jeanne Ferrante and Charles W. Rackoff. *The computational complexity of logical theories.* Number 718 in Lecture Notes in Mathematics. Springer Verlag, 1979.

[Fri85] L. Fribourg. SLOG: A logic programming language interpreter based on clausal superposition and rewriting. In *IEEE Symposium on Logic Programming*, Boston (MA), 1985.

[GM86] J. A. Goguen and J. Meseguer. EQLOG: Equality, types, and generic modules for logic programming. In Douglas De Groot and Gary Lindstrom, editors, *Functional and Logic Programming*, pages 295–363. Prentice Hall, Inc., 1986. An earlier version appears in *Journal of Logic Programming*, Volume 1, Number 2, pages 179–210, September 1984.

[GMW97] H. Ganzinger, C. Meyer, and C. Weidenbach. Soft typing for ordered resolution. In W. McCune, editor, *Proc. 14th Conference on Automated Deduction*, volume 1249 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1997.

[GRS87] Jean Gallier, S. Raatz, and Wayne Snyder. Theorem proving using rigid E-unification: Equational matings. In *Proc. 2nd IEEE Symp. Logic in Computer Science, Ithaca, NY*, June 1987.

[GS87] J. Gallier and W. Snyder. A general complete E-unification procedure. In P. Lescanne, editor, *Proceedings 2nd Conference on Rewriting Techniques and Applications, Bordeaux (France)*, volume 256 of *Lecture Notes in Computer Science*, pages 216–227, Bordeaux (France), 1987. Springer-Verlag.

[GS89] J. Gallier and W. Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, 67(2-3):203–260, October 1989.

[GTT93a] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated subset relationships. In *Proc. 34th Symposium on Foundations of Computer Science*, pages 372–380, Palo Alto, CA, November 1993. IEEE Computer society press.

[GTT93b] Rémy Gilleron, Sophie Tison, and Marc Tommasi. Solving systems of set constraints using tree automata. In *Proc. 10th Symposium on Theoretical Aspects of Computer Science, Würzburg, LNCS*, 1993.

[GTT94] R. Gilleron, S. Tison, and M. Tommasi. Some new decidability results on positive and negative set constraints. In Jouannaud [Jou94], pages 336–351.

[HKK98] Claus Hintermeier, Claude Kirchner, and Hélène Kirchner. Dynamically-typed computations for order-sorted equational presentations. *Journal of Symbolic Computation*, 25(4):455–526, 98. Also report LORIA 98-R-157.

[Höl89] S. Hölldobler. *Foundations of Equational Logic Programming*, volume 353 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1989.

[HR91] J. Hsiang and M. Rusinowitch. Proving refutational completeness of theorem proving strategies: The transfinite semantic tree method. *Journal of the ACM*, 38(3):559–587, July 1991.

[Hue76] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω.* Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

[Hul79] J.-M. Hullot. Associative-commutative pattern matching. In *Proceedings 9th International Joint Conference on Artificial Intelligence*, 1979.

[Hul80a] J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings 5th International Conference on Automated Deduction, Les Arcs (France)*, volume 87 of *Lecture Notes in Computer Science*, pages 318–334. Springer-Verlag, July 1980.

[Hul80b] J.-M. Hullot. *Compilation de Formes Canoniques dans les Théories équationelles.* Thèse de Doctorat de Troisième Cycle, Université de Paris Sud, Orsay (France), 1980.

[JK91] J.-P. Jouannaud and Claude Kirchner. Solving equations in abstract algebras: a rule-based survey of unification. In Jean-Louis Lassez and G. Plotkin, editors, *Computational Logic. Essays in honor of Alan Robinson*, chapter 8, pages 257–321. The MIT press, Cambridge (MA, USA), 1991.

[JKK83] J.-P. Jouannaud, Claude Kirchner, and Hélène Kirchner. Incremental construction of unification algorithms in equational theories. In *Proceedings International Colloquium on Automata, Languages and Programming, Barcelona (Spain)*, volume 154 of *Lecture Notes in Computer Science*, pages 361–373. Springer-Verlag, 1983.

[JMW98] Florent Jacquemard, Christoph Meyer, and Christoph Weidenbach. Unification in extensions of shallow equational theories. In T. Nipkow, editor, *Porceedings of the 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 76–90, Tsukuba, Japan, 1998. Springer-Verlag.

[JO91] J.-P. Jouannaud and M. Okada. Satisfiability of systems of ordinal notations with the subterm property is decidable. In J. Leach Albert, B. Monien, and M. Rodríguez Artalejo, editors, *Proceedings 18th ICALP Conference, Madrid (Spain)*, volume 510 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[Jou94] Jean-Pierre Jouannaud, editor. *First International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, München, Germany, September 1994. Springer Verlag.

[Kir85] Claude Kirchner. *Méthodes et outils de conception systématique d'algorithmes d'unification dans les théories équationnelles.* Thèse de Doctorat d'Etat, Université Henri Poincaré – Nancy 1, 1985.

[KK89] Claude Kirchner and Hélène Kirchner. Constrained equational reasoning. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, Portland (Oregon)*, pages 382–389. ACM Press, July 1989. Report CRIN 89-R-220.

[KK90] Claude Kirchner and F. Klay. Syntactic theories and unification. In *Proceedings 5th IEEE Symposium on Logic in Computer Science, Philadelphia (Pa., USA)*, pages 270–277, June 1990.

[KK99] Claude Kirchner and Hélène Kirchner. Rewriting, solving, proving. A preliminary version of a book available at `www.loria.fr/~ckirchne/rsp.ps.gz`, 1999.

[KKR90] Claude Kirchner, Hélène Kirchner, and M. Rusinowitch. Deduction with symbolic constraints. *Revue d'Intelligence Artificielle*, 4(3):9–52, 1990. Special issue on Automatic Deduction.

[KKV95] Claude Kirchner, Hélène Kirchner, and Marian Vittek. Designing constraint logic programming languages using computational systems. In P. Van Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming. The Newport Papers.*, chapter 8, pages 131–158. The MIT press, 1995.

[KL87] Claude Kirchner and P. Lescanne. Solving disequations. In D. Gries, editor, *Proceedings 2nd IEEE Symposium on Logic in Computer Science, Ithaca (N.Y., USA)*, pages 347–352. IEEE, 1987.

[Kla99] Nils Klarlund. A theory of restrictions for logics and automata. In *Proc. Computer Aided Verification (CAV)*, volume 1633 of *Lecture Notes in Computer Science*, pages 406–417, 1999.

[Koz94] D. Kozen. Set constraints and logic programming. In Jouannaud [Jou94]. To appear in Information and Computation.

[KR98] Claude Kirchner and Christophe Ringeissen. Rule-Based Constraint Programming. *Fundamenta Informaticae*, 34(3):225–262, September 1998.

[LM87] Jean-Louis Lassez and K. Marriot. Explicit representation of terms defined by counter examples. *Journal of Automated Reasoning*, 3(3):301–318, 1987.

[LM94] D. Lugiez and J.L. Moysset. Tree automata help one to solve equational formulae in ac-theories. *Journal of symbolic computation*, 11(1), 1994.

[LMM88] Jean-Louis Lassez, M. J. Maher, and K. Marriot. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*. Morgan-Kaufman, 1988.

[LR97] S. Limet and P. Réty. E-unification by means of tree tuple synchronized grammars:. *Discrete Mathematics and Theoretical Computer Science*, 1(1):69–98, 1997.

[Mah88] M. J. Maher. Complete axiomatization of the algebra of finite, rational trees and infinite trees. In *Proceedings 3rd IEEE Symposium on Logic in Computer Science, Edinburgh (UK)*. COMPUTER SOCIETY PRESS, 1988.

[Mat70] Y. Matijasevič. Diophantine representation of recursively enumerable predicates. In *Actes du Congrès International des Mathématiciens*, volume 1, pages 235–238, Nice (France), 1970.

[MGS90] J. Meseguer, J. A. Goguen, and G. Smolka. Order-sorted unification. In Claude Kirchner, editor, *Unification*, pages 457–488. Academic Press, London, 1990.

[MN89] U. Martin and T. Nipkow. Boolean unification — the story so far. *Journal of Symbolic Computation*, 7(3 & 4):275–294, 1989. Special issue on unification. Part one.

[MN98] Martin Müller and Joachim Niehren. Ordering constraints over feature trees expressed in second-order monadic logic. In *Proc. Int. Conf. on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 196–210, Tsukuba, Japan, 1998.

[MNRA92] J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: the language BABEL. *Journal of Logic Programming*, 12(3):191–223, February 1992.

[MS90] Michael J. Maher and Peter J. Stuckey. On inductive inference of cyclic structures. In F. Hoffman, editor, *Annals of Mathematics and Artificial*

Intelligence, volume F. J.C. Baltzer Scientific Pub. Company, 1990. To appear.

[Mza86] J. Mzali. *Méthodes de filtrage équationnel et de preuve automatique de théorèmes.* Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, 1986.

[Nie93] R. Nieuwenhuis. Simple lpo constraint solving methods. *Information Processing Letters*, 47(2), 1993.

[Nie95] Robert Nieuwenhuis. On narrowing, refutation proofs and constraints. In Jieh Hsiang, editor, *Rewriting Techniques and Applications, 6th International Conference, RTA-95*, LNCS 914, pages 56–70, Kaiserslautern, Germany, April 5–7, 1995. Springer-Verlag.

[Nie99] Roberto Nieuwenhuis. Solved forms for path ordering constraints. In Paliath Narendran and Michael Rusinowitch, editors, *Porceedings of RTA'99*, volume 1631 of *Lecture Notes in Computer Science*, pages 1–15. Springer Verlag, July 1999.

[NO90] P. Narendran and F. Otto. Some results on equational unification. In M. E. Stickel, editor, *Proceedings 10th International Conference on Automated Deduction, Kaiserslautern (Germany)*, volume 449 of *Lecture Notes in Computer Science*, pages 276–291, July 1990.

[NR95] R. Nieuwenhuis and A. Rubio. Theorem Proving with Ordering and Equality Constrained Clauses. *J. Symbolic Computation*, 19(4):321–352, April 1995.

[NRS89] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7(3 & 4):295–318, 1989. Special issue on unification. Part one.

[NRV98] P. Narendran, M. Rusinowitch, and R. Verma. RPO constraint solving is in NP. In *Annual Conference of the European Association for Computer Science Logic*, Brno (Czech Republic), August 1998. Available as Technical Report 98-R-023, LORIA, Nancy (France).

[Nut89] W. Nutt. The unification hierarchy is undecidable. In H.-J. Bürckert and W. Nutt, editors, *Proceedings 3rd International Workshop on Unification, Lambrecht (Germany)*, June 1989.

[Pad96] Vincent Padovani. *Filtrage d'ordre supérieur.* PhD thesis, Université de Paris VII, 1996.

[Per90] Dominique Perrin. Finite automata. In *Handbook of Theoretical Computer Science*, volume Formal Models and Semantics, pages 1–58. Elsevier, 1990.

[Pic99] Reinhard Pichler. Solving equational problems efficiently. In Harald Ganzinger, editor, *Automated Deduction – CADE-16, 16th International Conference on Automated Deduction*, LNAI 1632, pages 97–111, Trento, Italy, July 7–10, 1999. Springer-Verlag.

[Plo72] G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[PP97] L. Pacholski and A. Podelski. Set constraints - a pearl in research on constraints. In Gert Smolka, editor, *Proc. 3rd Conference on Principles and Practice of Constraint Programming - CP97*, volume 1330 of *Lecture Notes in Computer Science*, pages 549–561, Linz, Austria, 1997. Springer Verlag. Invited Tutorial.

[PW78] M. S. Paterson and M. N. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.

[Rab69] M.O. Rabin. Decidability of second-order theories and automata on infinite trees. *Trans. Amer. Math. Soc.*, 141:1–35, 1969.

[Rab77] M. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 595–629. North-Holland, 1977.

[Rin97] Christophe Ringeissen. Prototyping Combination of Unification Algorithms with the ELAN Rule-Based Programming Language. In *Proceedings 8th Conference on Rewriting Techniques and Applications, Sitges (Spain)*, volume 1232 of *Lecture Notes in Computer Science*, pages 323–326. Springer-Verlag, 1997.

[SKR98] T. Shiple, J. Kukula, and R. Ranjan. A comparison of presburger engines for EFSM reachability. In *Proc. Computer Aided verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 280–292, 1998.

[Sny88] W. Snyder. *Complete sets of transformations for general unification.* PhD thesis, University of Pennsylvania, 1988.

[SS82] J. Siekmann and P. Szabó. Universal unification and classification of equational theories. In *Proceedings 6th International Conference on Automated Deduction, New York (N.Y., USA)*, volume 138 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[SS84] J. Siekmann and P. Szabó. Universal unification. In R. Shostak, editor, *Proceedings 7th International Conference on Automated Deduction, Napa Valley (Calif., USA)*, volume 170 of *Lecture Notes in Computer Science*, pages 1–42, Napa Valley (California, USA), 1984. Springer-Verlag.

[SS90] M. Schmidt-Schauß. Unification in permutative equational theories is undecidable. In Claude Kirchner, editor, *Unification*, pages 117–124. Academic Press inc., London, 1990.

[ST94] Gert Smolka and Ralf Treinen. Records for logic programming. *Journal of Logic Programming*, 18(3):229–258, April 1994.

[Sza82] P. Szabó. *Unifikationstheorie erster Ordnung.* PhD thesis, Universität Karlsruhe, 1982.

[TA87] E. Tiden and S. Arnborg. Unification problems with one-sided distributivity. *Journal of Symbolic Computation*, 3(1 & 2):183–202, April 1987.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 134–191. Elsevier, 1990.

[Tho97] Wolfgang Thomas. Languages, automata and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 3, pages 389–456. Springer-Verlag, 1997.

[Tom94] Marc Tommasi. Automates et contraintes ensemblistes. Thèse de l'Univ. de Lille, February 1994.

[Tre92] R. Treinen. A new method for undecidability proofs of first order theories. *J. Symbolic Computation*, 14(5):437–458, November 1992.

[Tre97] Ralf Treinen. Feature trees over arbitrary structures. In Patrick Blackburn and Maarten de Rijke, editors, *Specifying Syntactic Structures*, chapter 7, pages 185–211. CSLI Publications and FoLLI, 1997.

[TW68] J.W. Thatcher and J.B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Math. Systems Theory*, 2:57–82, 1968.

[TW93] Jerzy Tiuryn and Mitchell Wand. Type reconstruction with recursive types and atomic subtyping. In *CAAP '93: 18th Colloquium on Trees in Algebra and Programming*, July 1993.

[Var96] M. Vardi. An automata-theoretic approach to linear time logic. In *Logic for concurrency: structure versus automata*, volume 1043 of *Lecture Notes in Comp. Science*. Springer Verlag, 1996.

[Vea97] Margus Veanes. *On simultaneous rigid E-unification*. PhD thesis, Computing Science Department, Uppsala University, Uppsala, Sweden, 1997.

[Ven87] K. N. Venkataraman. Decidability of the purely existential fragment of the theory of term algebras. *Journal of the ACM*, 34(2):492–510, 1987.

[WBK94] A. Werner, A. Bockmayr, and S. Krischer. How to realize LSE narrowing. In *Proceedings Fourth International Conference on Algebraic and Logic Programming, Madrid (Spain)*, Lecture Notes in Computer Science. Springer-Verlag, September 1994.

3 Combining Constraint Solving

Franz Baader[1] and Klaus U. Schulz[2*]

[1] Theoretical Computer Science, RWTH Aachen, Germany
[2] CIS, University of Munich, Germany

3.1 Introduction

In many areas of Logic, Computer Science, and Artificial Intelligence, there is a need for specialized formalisms and inference mechanisms to solve domain-specific tasks. For this reason, various methods and systems have been developed that allow for an efficient and adequate treatment of such restricted problems. In most realistic applications, however, one is faced with a complex combination of different problems, which means that a system tailored to solving a single problem can only be applied if it is possible to combine it both with other specialized systems and with general purpose systems.

This general problem of combining systems can be addressed on various conceptual levels. At one end, combination of logical systems is studied with an emphasis on formal properties, using tools from mathematics and logics. Examples of results obtained on this level are transfer results for modal logics [KW91, Hem94] and modularity results for term rewriting systems [Toy87, Ohl94]. On the other end of the spectrum, the combination of software tools necessitates considering physical connections and appropriate communication languages [CH96,LB96]. Between these two extremes lies the combination of constraint systems and the respective solvers, which is the topic of this paper. On the one hand, defining a semantics for the combined system may depend on methods and results from formal logic and universal algebra. On the other hand, an efficient combination of the actual constraint solvers often requires the possibility of communication and cooperation between the solvers.

Since there is a great variety of constraint systems and of approaches for combining them, we will start with a short classification of the different approaches. Subsequently, we will describe two of the most prominent combination approaches in this area:

- the Nelson-Oppen scheme [NO79, Opp80] for combining decision procedures for the validity of quantifier-free formulae in first-order theories, which was originally motivated by program verification;
- methods for combining E-unification algorithms [SS89a,Bou93,BS96] and more general constraint solvers [BS98], which are of interest in theorem proving, term rewriting, and constraint programming.

* Both authors supported by the ESPRIT working group CCL-II, ref. WG # 22457.

Our treatment of the Nelson-Oppen method can be seen as a warm-up exercise for the second approach since it is simpler both w.r.t. the actual combination algorithm and w.r.t. the (algebraic) tools required for proving its correctness. The problem of combining unification algorithms will be treated in more detail. First, we will describe the combination algorithm as introduced in [BS92, BS96] and briefly sketch how to prove its correctness. We will then give a logical reformulation of the combination algorithm, and describe an algebraic approach for proving its correctness. This approach has the advantage that it can be generalized to constraints more general than the equational constraints of unification problems and to solution structures more general than the E-free algebras of unification problems. Finally, we will sketch approaches for optimizing the combination algorithm, and comment on principal limitations for optimizations.

3.2 Classification of Constraint Systems and Combination Approaches

Before classifying different approaches for combining constraint system, we must explain what we mean by constraint systems and constraint solvers. We will also consider two examples, which already introduce the different forms of constraint systems whose combination will be considered in more detail in this paper.

3.2.1 Constraint Systems and Solvers

Informally, a *constraint system* is given by a constraint language and a "semantics" for this language. The *constraint language* determines which formal expressions are considered to be admissible constraints, and the *semantics* provides us with a notion of constraint satisfaction: given an admissible constraint, it is uniquely determined by the semantics whether this constraint is satisfiable or not. In general, this does not mean that this question is also effectively decidable. In its most basic form, a *constraint solver* for a given constraint system is a procedure that decides satisfiability.

The constraint language. To put these notions on a more formal footing, we assume that constraints are formulae of first-order predicate logic, and that the semantics is provided by the usual semantics of first-order logic. To be more precise, we consider an (at most countable) *signature* Σ consisting of function symbols and predicate symbols, and a (countably infinite) set V of (individual) variables, and build first-order terms, called *Σ-terms*, and first-order formulae, called *Σ-formulae*, from these ingredients in the usual way. Usually, a *Σ-constraint* is a Σ-formula $\varphi(v_1, \dots, v_n)$ with free variables $v_1, \dots, v_n$, and a solution of the constraint replaces the variables such that the resulting expression is "true" under the given semantics. If we are interested only in solvability of the constraints and not in actually computing a solution, we may also use closed formulae (e.g., the existential closure of the open formula) as constraints. It should also be noted that a constraint language usually does not allow for all Σ-formulae, but only for a certain subclass of formulae, to be used as constraints. Thus, a constraint language is characterized by a signature Σ and a class of Σ-formulae, which may be open or closed.

The semantics. Given such a constraint language, its semantics can be defined in two different ways: by a Σ-theory T or a Σ-structure $\mathcal{A}$. A Σ-theory is given by a set T of closed Σ-formulae, and a Σ-structure $\mathcal{A}$ is a Σ-interpretation, i.e., a nonempty set A, the domain of $\mathcal{A}$, together with an interpretation of the n-ary predicate (function) symbols as n-ary relations (functions) on A.

For a given Σ-structure $\mathcal{A}$, a *solution* of the Σ-constraint $\varphi(v_1, \dots, v_n)$ in $\mathcal{A}$ is a mapping $\{v_1 \mapsto a_1, \dots, v_n \mapsto a_n\}$ of the free variables of the constraint to elements of A such that $\mathcal{A} \models \varphi(a_1, \dots, a_n)$, i.e., $\varphi(v_1, \dots, v_n)$ is true in $\mathcal{A}$ under the evaluation $\{v_1 \mapsto a_1, \dots, v_n \mapsto a_n\}$. The constraint $\varphi(v_1, \dots, v_n)$ is *satisfiable* in $\mathcal{A}$ iff it has a solution in $\mathcal{A}$. This is equivalent to saying that its existential closure $\exists v_1. \cdots \exists v_n.\varphi(v_1, \dots, v_n)$ is valid in $\mathcal{A}$.

For a given Σ-theory T, there are two different ways of defining satisfiability of constraints, depending on whether we want the constraints to be satisfiable (in the sense introduced above) in all models or in some model of the theory T. In the first case, which is the more usual case in constraint solving, the Σ-constraint $\varphi(v_1, \dots, v_n)$ is *satisfiable* in (all models of) T iff its existential closure $\exists v_1. \cdots \exists v_n.\varphi(v_1, \dots, v_n)$ is valid in T. The second case coincides with the usual definition of satisfiability of (open) formulae in predicate logic: the Σ-constraint $\varphi(v_1, \dots, v_n)$ is *satisfiable* in (some model of) T iff its existential closure $\exists v_1. \cdots \exists v_n.\varphi(v_1, \dots, v_n)$ is satisfiable in T, i.e., valid in some model of T. In both cases, one does not have a natural notion of solution since there is more than one solution structure involved, though there may be specific instances where such a notion can be defined.

To sum up, we can define satisfiability of a constraint in three different ways: as validity of its existential closure in (i) a fixed solution structure $\mathcal{A}$; (ii) all models of a fixed theory T; (iii) some model of a fixed theory T.

Note that (i) is a special case of (ii) since we can take as theory T the theory of $\mathcal{A}$, i.e., the set of all Σ-formulae valid in $\mathcal{A}$. In general, (ii) is not a special case of (i). This is the case, however, if there exists a Σ-structure $\mathcal{A}$ that is *canonical* for T and the constraint language in the sense that a constraint is satisfiable in T iff it is satisfiable in $\mathcal{A}$.

The constraint solver. Given a constraint language and a semantics, a constraint solver is a procedure that is able to decide satisfiability of the constraints. In this paper, we will mostly restrict our attention to the combination of such decision procedures. It should be noted, however, that in many cases constraint solvers produce more than just the answer "yes" or "no".

If there is the notion of a solution available, one may also want to have a solver that not only decides satisfiability, but also computes a solution, if one exists. Since a given constraint may have more than one solution one may even be interested in obtaining a complete set of solutions, i.e., a set of solutions from which all solutions can be generated in a simple way. A prominent example of such a complete representation of all solutions is Robinson's most general unifier, from which all unifiers of a syntactic unification problem can be generated by instantiation.

Instead of actually computing a solution, the solver may transform the constraint into an equivalent one in "solved form." Such a solved form should, in some way, be simpler than the original constraint; in particular, the existence of a solved form should indicate satisfiability of the constraint, and it should be "easy" to read off an actual solution. The advantage of using solved forms is twofold. On the one hand, computing the solved form may be less complex than computing a solution. An example of this phenomenon is the so-called dag solved form of syntactic unification problems [JK91], which is linear in the size of the problem, whereas the most general unifier may be exponential in the size of the problem. On the other hand, a solver that computes a solved form is usually incremental: if the constraint is strengthened, then not all the work done during the satisfiability test needs to be re-done. To be more precise, assume that we have already tested φ for satisfiability, and then need to test $\varphi \wedge \psi$, a situation that frequently occurs, e.g., in constraint logic programming and theorem proving. If all we know after the satisfiability test is that φ is satisfiable, then we must start from scratch when testing $\varphi \wedge \psi$. However, if we have computed a solved form φ' of φ, then we can test $\varphi' \wedge \psi$ instead, which hopefully is easier.

3.2.2 More Notation and Two Examples

As examples that illustrate the notions introduced above, we consider quantifier-free formulae and E-unification problems. Before introducing these types of constraint systems more formally, we define some subclasses of first-order formulae, which will be of interest later on.

We consider logic *with equality*, i.e., the binary predicate symbol $=$, which is interpreted as equality on the domain, is always available, without being explicitly contained in the signature. A Σ-atom is of the form $P(t_1, \ldots, t_n)$, where P is an n-ary predicate symbol of $\Sigma \cup \{=\}$ and $t_1, \ldots, t_n$ are Σ-terms. If the atom is of the form $t_1 = t_2$, i.e., P is the equality symbol $=$, then it is called an *equational atom*; otherwise, it is called a *relational atom*. Negated equational atoms are written $t_1 \neq t_2$ rather than $\neg(t_1 = t_2)$. A Σ-matrix is a Boolean combination of Σ-atoms, and a positive Σ-matrix is built from Σ-atoms using conjunction and disjunction only. A *positive Σ-formula* is a quantifier prefix followed by a positive Σ-matrix. The formula is called *positive existential (positive AE)* iff the quantifier prefix consists of existential quantifiers (universal quantifiers followed by existential quantifiers). A *universal formula* is given by a universal quantifier prefix followed by a quantifier-free formula. A universal formula is called *conjunctive universal* iff its matrix is a conjunctions of Σ-atoms and negated Σ-atoms. *Σ-sentences* (of either type) are Σ-formulae without free variables. Given a Σ-structure $\mathcal{A}$ (a Σ-theory T), the *positive theory* of $\mathcal{A}$ (T) consists of the set of all positive Σ-sentences valid in $\mathcal{A}$ (T). The *positive existential, positive AE, universal, and conjunctive universal theories* of $\mathcal{A}$ (T) are defined analogously.

Quantifier-free formulae. The Nelson-Oppen [NO79,Opp80] combination method applies to constraint systems of the following form:

- For a given signature Σ, the constraint language consists of all quantifier-free Σ-formulae, i.e., all Σ-matrices.
- The semantics is defined by an arbitrary Σ-theory T.
- One is interested in satisfiability in *some* model of T.

Thus, the constraint solver must be able to decide whether the existential closure of a quantifier-free Σ-formula is valid in some model of T. Since a formula is valid in some model of T iff its negation is not valid in all models of T, a decision procedure for the universal theory of T can be used as a constraint solver for this type of constraint systems.

Unification problems. Unification modulo equational theories is a subfield of automated deduction which is well-investigated (see [JK91,BS94,BS00] for survey papers of the area).

An *equational theory* is given by a set E of identities $s = t$ between terms s, t. The *signature of* E is the set of all function symbols occurring in E. With $=_E$ we denote the least congruence relation on the term algebra $\mathcal{T}(\Sigma, V)$ that is closed under substitutions and contains E. Equivalently, $=_E$ can be introduced as the reflexive, transitive, and symmetric closure $\overset{*}{\leftrightarrow}_E$ of the rewrite relation $\rightarrow_E$ induced by E, or as the set of equational consequences of E, i.e., $s =_E t$ iff the universal closure of the atom $s = t$ is valid in all models of E. An equational theory E is *trivial* iff $x =_E y$ holds for two distinct variables x, y. In the following, we consider only non-trivial equational theories.

Given an equational theory E with signature Σ, an *elementary E-unification problem* is a finite system $\Gamma := \{s_1 =^? t_1, \ldots, s_n =^? t_n\}$ of equations between Σ-terms. In E-unification problems *with constants*, these terms may contain additional "free" constant symbols, i.e., constant symbols not contained in the signature Σ of E, and in *general* E-unifications problems, these terms may contain additional "free" function symbols, i.e., function symbols not contained in the signature Σ of E.

A *solution* (*E-unifier*) of the E-unification problem $\{s_1 =^? t_1, \ldots , s_n =^? t_n\}$ is a substitution σ such that $\sigma(s_i) =_E \sigma(t_i)$ for $i = 1, \ldots , n$. If there exists such a solution, then Γ is called *unifiable*. Recall that a *substitution* is a mapping from variables to terms which is almost everywhere equal to the identity. It can be extended to a mapping from terms to terms in the obvious way. Substitutions will be written in the form $\sigma = \{x_1 \mapsto u_1, \ldots , x_k \mapsto u_k\}$, where $x_1, \ldots , x_k$ are the finitely many variables that are changed by the substitution, and $u_1, \ldots , u_k$ are their respective σ-images. The set $\{x_1, \ldots , x_k\}$ is called the *domain* of the substitution σ, and $\{u_1, \ldots , u_k\}$ is called its *range*.

Let X be the set of all variables occurring in Γ. The E-unifier θ is an instance of the E-unifier σ iff there exists a substitution λ such that $\theta(x) =_E \lambda(\sigma(x))$ for all $x \in X$. A *complete set of E-unifiers of Γ* is a set C of E-unifiers of Γ such that every E-unifier of Γ is an instance of some unifier in C. Finite complete sets of E-unifiers yield a finite representation of the usually infinite sets of solutions of E-unification problems. Since this representation should be as small as possible, one is usually interested in *minimal complete sets of E-unifiers*, i.e., complete sets in which different elements are not instances of each other.

For a given equational theory E, elementary E-unification problems can be seen as constraints of the following constraint system:

- The constraint language consists of all conjunctions of equational atoms $s = t$. We call such a conjunction an *E-unification constraint*.
- The semantics is defined by the E-free algebra in countably many generators, i.e., the quotient term algebra $\mathcal{T}(\Sigma, V)/{=_E}$ for the countably infinite set of variables V.

Since the E-unification problem $\{s_1 =^? t_1, \ldots , s_n =^? t_n\}$ has a solution iff the existential closure of $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$ is valid in $\mathcal{T}(\Sigma, V)/{=_E}$, the notion of satisfiability of E-unification constraints induced by this semantics coincide with the notion of unifiability of E-unification problems introduced above.

Alternatively, the semantics of E-unification constraints can also be defined w.r.t. the equational theory E. In fact, the E-free algebra in countably many generators is a canonical solution structure for E-unification constraints: the existential closure of $s_1 = t_1 \wedge \ldots \wedge s_n = t_n$ is valid in $\mathcal{T}(\Sigma, V)/{=_E}$ iff it is valid in *all* models of E.

If we are interested only in decidability and not in complexity, then we can also consider general positive existential sentences instead of the conjunctive sentences obtained as existential closures of unification constraints. In fact, we can simply write the positive matrix in disjunctive normal form, and then distribute the existential quantifiers over disjunctions. This yields a disjunction of conjunctive sentences, which is valid iff one of its disjuncts is valid.

These observations are summed up in the following theorem:

Theorem 1. *Let E be an equational theory with signature Σ, and V a countably infinite set of variables. Then the following are equivalent:*

1. *Elementary E-unification is decidable.*
2. *The positive existential theory of E is decidable.*
3. *The positive existential theory of $\mathcal{T}(\Sigma, V)/{=_E}$ is decidable.*

In order to obtain a class of first-order formulae that correspond to E-unification problems with constants, but are built over the signature of E, we note that free constants can be generated via Skolemization. Since we are interested in validity of the formulae, we must Skolemize universal quantifiers, and since we want to obtain Skolem constants, these universal quantifiers should not be in the scope of an existential quantifier.

Theorem 2. *Let E be an equational theory with signature Σ, and V a countably infinite set of variables. Then the following are equivalent:*

1. *E-unification with constants is decidable.*
2. *The positive AE theory of E is decidable.*
3. *The positive AE theory of $\mathcal{T}(\Sigma, V)/{=_E}$ is decidable.*

These theorems can be seen as folk theorems in the unification community. Explicit proofs can be found in [Boc92]. For general E-unification, a similar characterization is possible. However, the proof of this result (which is not as straightforward as the ones for the above theorems) depends on results concerning the combination of E-unification algorithms. For this reason, we defer presenting the characterization to Section 3.5.1.

3.2.3 Combining Constraint Systems and Solvers

Given two constraint systems, it is not a priori clear what their combination is supposed to be, and in some cases there are several sensible candidates. Since constraint systems consist of a constraint language and a corresponding semantics, one must first define the combined language, and then introduce a combined semantics for this language.

Let us first consider the problem of defining the combined constraint language. This is quite simple if we restrict ourselves to the case where the two constraint languages consist of formulae of the same type, but over different signatures. In this case, the most natural candidate for the combined constraint language appears to be the language consisting of formulae of the given type, but over the union of the signatures. For example, if the constraint languages allow for quantifier-free formulae over the signatures Σ_1 and Σ_2, respectively, then the combined constraint language consists of all quantifier-free formulae over the signature $\Sigma_1 \cup \Sigma_2$. Similarly, combined unification problems consist of equations between terms over the union of the signatures of the component constraint systems. In this paper, we will consider only this simple case.

Once the combined constraint language is defined, it must be equipped with an appropriate semantics. If the semantics is defined via a theory, this is again quite simple: the natural candidate is the union of the component theories. Thus, given equational theories E_1 and E_2 with the respective signatures Σ_1 and Σ_2, the combined unification constraint system consists of all $(E_1 \cup E_2)$-unification problems (with the usual semantics). If the semantics of the component constraint systems is defined with the help of solution structures, things are more difficult. One must combine the solution structures of the components into a solution structure for the combined constraint language, and it not always obvious how this combined solution structure should look like. We will briefly come back to this problem of combining solution structures in Subsection 3.6.2.

Finally, given the combined constraint language with an appropriate semantics, one needs a constraint solver for this combined constraint system. For two specific constraint systems and their combination, one can of course try to construct an ad hoc constraint solver for the combined system, which may or may not employ the single solvers as subprocedures. A more satisfactory approach, however, is to design a combination scheme that applies to a whole class of constraint systems. The combination procedures that we will consider in the next two sections are of this form. For example, the Nelson-Oppen combination procedure can be used to combine decision procedures for the universal theories of T_1 and T_2 into one for the universal theory of $T_1 \cup T_2$. This holds for arbitrary theories T_1 and T_2 with decidable universal theory (and not just for two specific theories), provided that the signatures of T_1 and T_2 are disjoint. The combination result for E-unification algorithms is of the same type.

In both cases, the combination approach treats the solvers of the single constraint systems as black boxes, i.e., it does not make any assumptions on how these solvers work. This distinguishes these approaches from others that assume the existence of constraint solvers of a certain type. For example, a semi-complete (i.e., confluent and weakly normalizing) term rewriting system can be used to decide the word problem of the corresponding equational theory. Since confluence and weak normalization are modular properties, the union of two semi-complete term rewriting systems over disjoint signatures is again semi-complete [Ohl95], and thus the word problem in the combined equational theory is decidable as well. However, this result is of no help at all if the decision procedures for the word problem in the component equational theories are not based on rewriting.

3.3 The Nelson-Oppen Combination Procedure

This procedure, which was first introduced in [NO79], is concerned with combining decision procedures for the validity of universal sentences in first-order theories, or equivalently with combining constraint solvers that test satisfiability of quantifier-free formulae in some model of the theory. To be more precise, assume that Σ_1 and Σ_2 are two *disjoint* signatures, and that T is obtained as the union of a Σ_1-theory T_1 and a Σ_2-theory T_2. How can decision procedures for validity (equivalently: satisfiability) in T_i $(i = 1, 2)$ be used to obtain a decision procedure for validity (equivalently: satisfiability) in T?

When considering the satisfiability problem, as done in Nelson and Oppen's method, we may without loss of generality restrict our attention to *conjunctive* quantifier-free formulae, i.e., conjunctions of Σ-atoms and negated Σ-atoms. In fact, a given quantifier-free formula can be transformed into an equivalent formula in disjunctive normal form (i.e., a disjunction of conjunctive quantifier-free formulae), and this disjunction is satisfiable in T iff one of its disjuncts is satisfiable in T.

Given a conjunctive quantifier-free formula φ over the combined signature $\Sigma_1 \cup \Sigma_2$, it is easy to generate a conjunction $\varphi_1 \wedge \varphi_2$ that is equivalent to φ, where φ_i is a pure Σ_i-formula, i.e., contains only symbols from Σ_i $(i = 1, 2)$. Here equivalent means that φ and $\varphi_1 \wedge \varphi_2$ are satisfiable in exactly the same models of T. This is achieved by *variable abstraction*, i.e., by replacing alien subterms by variables and adding appropriate equations.

Variable abstraction. Assume that t is a term whose topmost function symbol is in Σ_i, and let j be such that $\{i, j\} = \{1, 2\}$. A subterm s of t is called *alien subterm* of t iff its topmost function symbol belongs to Σ_j and every proper superterm of s in t has its top symbol in Σ_i.

Given a conjunctive quantifier-free formula φ, the variable abstraction process iteratively replaces terms by variables and adds appropriate equations to the conjunction:

- If φ contains an equational conjunct $s = t$ such that the topmost function symbols of s and t belong to different signatures, then replace $s = t$ by the conjunction $u = s \wedge u = t$, where u is a new variable, i.e., a variable not occurring in φ.
- If φ contains a negated equational conjunct $s \neq t$ such that the topmost function symbols of s and t belong to different signatures, then replace $s = t$ by the conjunction $u \neq v \wedge u = s \wedge v = t$, where u, v are new variables.
- If φ contains a relational conjunct $P(\dots, s_i, \dots)$ such that the topmost function symbol of s_i does not belong to the signature of P, then replace $P(\dots, s_i, \dots)$ by the conjunction $P(\dots, u, \dots) \wedge u = s_i$, where u is a new variable. Conjuncts of the form $\neg P(\dots, s_i, \dots)$ are treated analogously.

- If φ contains a (relational or equational) conjunct $P(\dots, s_i, \dots)$ such that s_i contains an alien subterm t, then replace $P(\dots, s_i, \dots)$ by the $P(\dots, s'_i, \dots) \wedge u = t$, where u is a new variable and s'_i is obtained from s_i by replacing the alien subterm t by u. Conjuncts of the form $\neg P(\dots, s_i, \dots)$ are treated analogously.

Obviously, this abstraction process always terminates and the resulting formula can be written in the form $\varphi_1 \wedge \varphi_2$, where φ_i is a pure Σ_i-formula ($i = 1, 2$). In addition, it is easy to see that the original formula φ and the new formula $\varphi_1 \wedge \varphi_2$ are satisfiable in exactly the same models of $T = T_1 \cup T_2$. Consequently, if φ is satisfiable in a model of T, then both φ_1 and φ_2 are satisfiable in a model of T, which is also a model of T_1 and of T_2. This shows that satisfiability of φ in a model of T implies satisfiability of φ_i in a model of T_i for $i = 1, 2$. Unfortunately, the converse need not hold, i.e., satisfiability of φ_i in a model of T_i ($i = 1, 2$) does not necessarily imply satisfiability of $\varphi_1 \wedge \varphi_2$ in a model of T, and thus also not satisfiability of φ in a model of T.

The reason for this problem is that φ_1 and φ_2 may share variables, and one formula may force some of these variables to be interpreted by the same element of the model, whereas the other is only satisfiable if they are interpreted by distinct elements of the model. To overcome this problem, Nelson and Oppen's procedure propagates equalities between variables from the formula φ_1 to φ_2, and vice versa.

The combination procedure. Given a conjunctive quantifier-free ($\Sigma_1 \cup \Sigma_2$)-formula φ to be tested for satisfiability (in some model of $T_1 \cup T_2$), Nelson and Oppen's method for combining decision procedures proceeds in three steps:

1. *Use variable abstraction to generate a conjunction $\varphi_1 \wedge \varphi_2$ that is equivalent to φ, where φ_i is a pure Σ_i-formula ($i = 1, 2$).*
2. *Test the pure formulae for satisfiability in the respective theories.*
   If φ_i is unsatisfiable in T_i for $i = 1$ or $i = 2$, then return "unsatisfiable." Otherwise proceed with the next step.
3. *Propagate equalities between different shared variables (i.e., distinct variables u_i, v_i occurring in both φ_1 and φ_2), if a disjunction of such equalities can be deduced from the pure parts.*
   A disjunction $u_1 = v_1 \vee \dots \vee u_k = v_k$ of equations between different shared variables can be deduced from φ_i in T_i iff $\varphi_i \wedge u_1 \neq v_1 \wedge \dots \wedge u_k \neq v_k$ is unsatisfiable in T_i. Since the satisfiability problem in T_i was assumed to be decidable, and since there are only finitely many shared variables, it is decidable whether there exists such a disjunction.
   If no such disjunctions can be deduced, return "satisfiable." Otherwise, take any of them,[1] and propagate its equations as follows. For every disjunct $u_j = v_j$, proceed with the second step for the formula $\sigma_j(\varphi_1) \wedge$

[1] For efficiency reasons, one should take a disjunction with minimal k.

$\sigma_j(\varphi_2)$, where $\sigma_j := \{u_j \mapsto v_j\}$ (for $j = 1, \ldots, k$). The answer is "satisfiable" iff one of these cases yields "satisfiable."

Obviously, the procedure *terminates* since there are only finitely many shared variables to be identified. In addition, it is easy to see that satisfiability is preserved at each step. This implies *completeness* of the procedure, that is, if it answers "unsatisfiable" (since already one of the pure subformulae is unsatisfiable in its theory), the original formula was indeed unsatisfiable. Before showing soundness of the procedure (which is more involved), we illustrate the working of the procedure by an example.

Example 1. Consider the equational[2] theories $T_1 := \{f(x,x) = x\}$ and $T_2 := \{g(g(x)) = g(x)\}$ over the signatures $\Sigma_1 := \{f\}$ and $\Sigma_2 := \{g\}$. Assume that we want to know whether the quantifier-free (mixed) formula

$$g(f(g(z), g(g(z)))) = g(z)$$

is valid in $T_1 \cup T_2$. To this purpose we apply the Nelson-Oppen procedure to its negation $g(f(g(z), g(g(z)))) \neq g(z)$.

In *Step 1*, $f(g(z), g(g(z)))$ is an alien subterm in $g(f(g(z), g(g(z))))$ (since $g \in \Sigma_2$ and $f \in \Sigma_1$). In addition, $g(z)$ and $g(g(z))$ are alien subterms in $f(g(z), g(g(z)))$. Thus, variable abstraction yields the conjunction $\varphi_1 \wedge \varphi_2$, where

$$\varphi_1 := u = f(v,w) \quad \text{and} \quad \varphi_2 := g(u) \neq g(z) \wedge v = g(z) \wedge w = g(g(z)).$$

In *Step 2*, it is easy to see that both pure formulae are satisfiable in their respective theories. The equation $u = f(v,w)$ is obviously satisfiable in the trivial model of T_1 (of cardinality 1). The formula φ_2 is, for example, satisfiable in the T_2-free algebra with two generators, where u is interpreted by one generator, z by the other, and v, w as required by the equations.

In *Step 3*, we can deduce $w = v$ from φ_2 in T_2 since φ_2 contains $v = g(z) \wedge w = g(g(z))$ and T_2 contains the (universally quantified) identity $g(g(x)) = g(x)$. Propagating the equality $w = v$ yields the pure formulae

$$\varphi_1' := u = f(v,v) \quad \text{and} \quad \varphi_2' := g(u) \neq g(z) \wedge v = g(z) \wedge v = g(g(z)),$$

which again turn out to be separately satisfiable in *Step 2* (with the same models as used above).

In *Step 3*, we can now deduce the equality $u = v$ from φ_1' in T_1, and its propagation yields

$$\varphi_1'' := v = f(v,v) \quad \text{and} \quad \varphi_2'' := g(v) \neq g(z) \wedge v = g(z) \wedge v = g(g(z)).$$

In *Step 2*, it turns out that φ_2'' is not satisfiable in T_2, and thus the answer is "unsatisfiable," which shows that $g(f(g(z), g(g(z)))) = g(z)$ is valid in $T_1 \cup T_2$. In fact, $v = g(z)$ and the identity $g(g(x)) = g(x)$ of T_2 imply that $g(v) = g(z)$, which contradicts $g(v) \neq g(z)$.

[2] Recall that the identities in equational theories are (implicitly) universally quantified.

Soundness of the procedure. As mentioned above, termination and completeness of the procedure are quite trivial. Soundness of the procedure, i.e., if the procedure answers "satisfiable," then the input formula is indeed satisfiable, is less trivial. In fact, for arbitrary theories T_1 and T_2, the combination procedure need not be sound. One must assume that each T_i is *stably infinite*, that is, such that a quantifier-free formula φ_i is satisfiable in T_i iff it is satisfiable in an infinite model of T_i. This restriction was not mentioned in Nelson and Oppen's original article [NO79]; it was introduced by Oppen in [Opp80].

The following example demonstrates that the Nelson-Oppen combination procedure need not be sound for theories that are not stably infinite, even if the theories in question are non-trivial[3] equational theories.

Example 2. Let $E_1 := \{f(g(x), g(y)) = x,\ f(g(x), h(y)) = y\}$ and $E_2 := \{k(x) = k(x)\}$. The theory E_2 is obviously non-trivial, and it is easy to see that E_1 is also non-trivial: by orienting the equations from left to right, one obtains a canonical term rewriting system, in which any two distinct variables have different normal forms.

First, we show that E_1 is not stably infinite. To this purpose, we consider the quantifier-free formula $g(x) = h(x)$. Obviously, this formula is satisfiable in the trivial (one-element) model of E_1. In every model $\mathcal{A}$ of E_1 that satisfies $g(x) = h(x)$, there exists an element e such that $g^{\mathcal{A}}(e) = h^{\mathcal{A}}(e)$. Here, $g^{\mathcal{A}}, h^{\mathcal{A}}$ denote the interpretations of the unary function symbols f, g by functions $A \to A$. But then we have for any element a of $\mathcal{A}$ that

$$a = f^{\mathcal{A}}(g^{\mathcal{A}}(a), g^{\mathcal{A}}(e)) = f^{\mathcal{A}}(g^{\mathcal{A}}(a), h^{\mathcal{A}}(e)) = e,$$

i.e., all elements of $\mathcal{A}$ are equal to e, which shows that $\mathcal{A}$ is the trivial model. Thus, $g(x) = h(x)$ is satisfiable only in the trivial model of E_1, which show that the (non-trivial) equational theory E_1 is not stably infinite.

To show that this really leads to an unsound behavior of the Nelson-Oppen method, we consider the mixed conjunction $g(x) = h(x) \wedge k(x) \neq x$. Clearly, $k(x) \neq x$ is satisfiable in E_2 (for instance, in the E_2-free algebra with 1 generator) and, as we saw earlier, $g(x) = h(x)$ is satisfiable in E_1. In addition, no equations between distinct shared variables can be deduced (since there is only one shared variable). It follows that Nelson and Oppen's procedure would answer "satisfiable" on input $g(x) = h(x) \wedge k(x) \neq x$. However, since $g(x) = h(x)$ is satisfiable only in the trivial model of E_1, and no disequation can be satisfied in a trivial model, $g(x) = h(x) \wedge k(x) \neq x$ is unsatisfiable in $E_1 \cup E_2$.

Nelson and Oppen's original proof of soundness of the procedure as well as a more recent one by Tinelli and Harandi [TH96] use Craig's Interpolation Theorem [CK90]. In the following, we sketch a proof that uses a very

[3] Recall that non-trivial means that the theory has a model of cardinality greater than 1.

elementary model theoretic construction: the fusion of structures. It goes back to Ringeissen [Rin96] and was further refined by Ringeissen and Tinelli [TR98, Tin99].[4]

In the following, let Σ_1 and Σ_2 be disjoint signatures, and $\Sigma := \Sigma_1 \cup \Sigma_2$ their union. A given Σ-structure $\mathcal{A}$ can also be viewed as a Σ_i-structure, by just forgetting about the interpretation of the symbols not contained in Σ_i. We call this Σ_i-structure the *Σ_i-reduct* of $\mathcal{A}$, and denote it by $\mathcal{A}^{\Sigma_i}$.

Definition 1. The Σ-structure $\mathcal{A}$ is a fusion of the Σ_1-structure $\mathcal{A}_1$ and the Σ_2-structure $\mathcal{A}_2$ iff the Σ_i-reduct $\mathcal{A}^{\Sigma_i}$ of $\mathcal{A}$ is Σ_i-isomorphic to $\mathcal{A}_i$ $(i = 1, 2)$.

Since the signatures Σ_1 and Σ_2 are disjoint, the existence of a fusion depends only on the cardinality of the structures $\mathcal{A}_1$ and $\mathcal{A}_2$.

Lemma 1. *Let $\mathcal{A}_1$ be a Σ_1-structure and $\mathcal{A}_2$ a Σ_2-structure. Then $\mathcal{A}_1$ and $\mathcal{A}_2$ have a fusion iff their domains A_1 and A_2 have the same cardinality.*

Proof. The only-if direction of the lemma is an immediate consequence of the definition of fusion. The if direction can be seen as follows: if A_1 and A_2 have the same cardinality, then there exists a bijection $\pi : A_1 \to A_2$. This bijection can be used to transfer the interpretation of the elements of Σ_2 from $\mathcal{A}_2$ to $\mathcal{A}_1$. To be more precise, let $\mathcal{A}$ be the Σ-structure that has domain A_1, interprets the elements of Σ_1 like $\mathcal{A}_1$, and interprets the elements of Σ_2 as follows:

- If f is an n-ary function symbol in Σ_2, and $a_1, \ldots, a_n \in A_1$, then we define $f^{\mathcal{A}}(a_1, \ldots, a_n) := \pi^{-1}(f^{\mathcal{A}_2}(\pi(a_1), \ldots, \pi(a_n)))$.
- If P is an n-ary predicate symbol in Σ_2, and $a_1, \ldots, a_n \in A_1$, then $(a_1, \ldots, a_n) \in P^{\mathcal{A}}$ iff $(\pi(a_1), \ldots, \pi(a_n)) \in P^{\mathcal{A}_2}$.

Then $\mathcal{A}$ is a fusion of $\mathcal{A}_1$ and $\mathcal{A}_2$ since $\mathcal{A}^{\Sigma_1}$ is identical to $\mathcal{A}_1$, and π is a Σ_2-isomorphism from $\mathcal{A}^{\Sigma_2}$ to $\mathcal{A}_2$ by construction of $\mathcal{A}$. $\square$

There is an interesting connection between the union of theories and fusions of models of the theories.

Proposition 1. *For $i = 1, 2$, let T_i be a Σ_i-theory. The Σ-structure $\mathcal{A}$ is a model of $T_1 \cup T_2$ iff it is a fusion of a model $\mathcal{A}_1$ of T_1 and a model $\mathcal{A}_2$ of T_2.*

Proof. The only-if direction is an immediate consequence of the facts that $\mathcal{A}$ is a fusion of its Σ_1-reduct $\mathcal{A}^{\Sigma_1}$ and its Σ_2-reduct $\mathcal{A}^{\Sigma_2}$, and that $\mathcal{A}^{\Sigma_i}$ is a model of T_i $(i = 1, 2)$.

The if direction is also trivial since, if $\mathcal{A}$ is a fusion of a model $\mathcal{A}_1$ of T_1 and a model $\mathcal{A}_2$ of T_2, then its Σ_i-reduct is isomorphic to $\mathcal{A}_i$, and thus a model of T_i $(i = 1, 2)$. Consequently, $\mathcal{A}$ is a model of $T_1 \cup T_2$. $\square$

[4] Actually, these papers consider the more general situation where the signatures need not be disjoint. Here, we restrict our attention to the disjoint case.

We are now ready to show a result from which soundness of the Nelson-Oppen procedure follows immediately. For a finite set of variables X, let $\Delta(X)$ denote the conjunction of all negated equations $x \neq y$ for distinct variables $x, y \in X$.

Proposition 2. *Let T_1 and T_2 be two stably infinite theories over the disjoint signatures Σ_1 and Σ_2, respectively; let φ_i be a quantifier-free Σ_i-formula ($i = 1, 2$), and let X be the set of variables occurring in both φ_1 and φ_2. If $\varphi_i \wedge \Delta(X)$ is satisfiable in a model $\mathcal{A}_i$ of T_i for $i = 1, 2$, then $\varphi_1 \wedge \varphi_2$ is satisfiable in a fusion of $\mathcal{A}_1$ and $\mathcal{A}_2$, and thus in a model of $T_1 \cup T_2$.*

Proof. Since the theories T_1 and T_2 are stably infinite and signatures are at most countable, we may without loss of generality assume that the structures $\mathcal{A}_1$ and $\mathcal{A}_2$ are both countably infinite. Since $\varphi_i \wedge \Delta(X)$ is satisfiable in $\mathcal{A}_i$, there is an evaluation $\alpha_i : X \to A_i$ that satisfies φ_i and replaces the variables in X by distinct elements of A_i ($i = 1, 2$). This implies that there exists a bijection $\pi : A_1 \to A_2$ such that $\pi(\alpha_1(x)) = \alpha_2(x)$. As shown in the proof of Lemma 1, this bijection induces a fusion $\mathcal{A}$ of $\mathcal{A}_1$ and $\mathcal{A}_2$. It is easy to see that $\varphi_1 \wedge \varphi_2$ is satisfiable in $\mathcal{A}$. The evaluation α showing satisfiability is defined as follows: on the variables in φ_1 it coincides with α_1, and for the non-shared variables x in φ_2 we define $\alpha(x) := \pi^{-1}(\alpha_2(x))$. □

It is easy to see that this proposition yields soundness of the Nelson-Oppen combination procedure, i.e, if the procedure answers "satisfiable," then the original formula was indeed satisfiable. In fact, if in Step 3 no disjunction of equalities between shared variables can be derived from the pure formulae, the prerequisite for the proposition is satisfied: since the disjunction of all equations $x = y$ for distinct variables $x, y \in X$ cannot be deduced from φ_i in T_i, we know that $\varphi_i \wedge \Delta(X)$ is satisfiable in T_i. Thus, we can deduce that $\varphi_1 \wedge \varphi_2$ is satisfiable in $T_1 \cup T_2$, and since each step of the procedure preserves satisfiability, the input formula was also satisfiable.

To sum up, we have shown correctness of Nelson and Oppen's combination procedure, which yields the following theorem:

Theorem 3. *Let T_1 and T_2 be two stably infinite theories over disjoint signatures such that the universal theory of T_i is decidable for $i = 1, 2$. Then the universal theory of $T_1 \cup T_2$ is also decidable.*

Complexity of the procedure. The main sources of complexity are (i) the transformation of the quantifier-free formula into a disjunction of *conjunctive* quantifier-free formulae, and (ii) Step 3 of the combination procedure for conjunctive quantifier-free formulae. It is well-known that the transformation of an arbitrary Boolean formula into disjunctive normal form may cause an exponential blow-up. Step 3 of the procedure has again two sources of complexity. First, there is the problem of deciding whether there is a disjunction of equalities between distinct variables that can be derived from φ_i in T_i. If one must really test all possible disjunctions using the satisfiability procedure for T_i, then already a single such step needs exponential time. However, even if we assume that there is a polynomial procedure that determines an appropriate disjunction (if there is one), then the overall algorithm is still not polynomial unless all these disjunctions consist of a single disjunct. Otherwise, the algorithm must investigate different branches, and since this may happen each time Step 3 is performed, an exponential number of branches may need to be investigated.

Nelson and Oppen [NO79] introduce the notion of a convex theory, and Oppen [Opp80] shows that, for convex theories, the two sources of complexity in Step 3 of the procedure can be avoided. A theory T is *convex* iff the following the following holds: if a disjunction of equalities between distinct variables can be deduced from a quantifier-free formula in T, then a single equality between distinct variables can already be deduced. For convex theories, Step 3 of the procedure can thus be modified as follows: it is only tested whether a single equation between distinct shared variables can be deduced. Since there are only a polynomial number of such equations, this can be tested by a polynomial number of calls to the satisfiability procedure for T_i. In addition, there is no more branching in Step 3. This shows that the modified combination procedure runs in polynomial time, if applied to conjunctive quantifier-free input formulae.

Theorem 4. *Let T_1 and T_2 be two* convex *and stably infinite theories over disjoint signatures such that the* conjunctive *universal theory of T_i is decidable in polynomial time for $i = 1, 2$. Then the* conjunctive *universal theory of $T_1 \cup T_2$ is also decidable in polynomial time.*

In the general case, the Nelson-Oppen combination approach yields a *nondeterministic* polynomial procedure. First, given an arbitrary quantifier-free formula, the nondeterministic procedure chooses from each disjunction one of the disjuncts. This yields a conjunctive quantifier-free formula. This formula is then treated by the following nondeterministic variant of the combination procedure:

1. *Use variable abstraction to generate a conjunction $\varphi_1 \wedge \varphi_2$ that is equivalent to φ, where φ_i is a pure Σ_i-formula ($i = 1, 2$).*
2. *Nondeterministically choose a variable identification, i.e., choose a partition $\Pi = \{\pi_1, \ldots, \pi_k\}$ of the variables shared by φ_1 and φ_2.*
   For each of the classes π_i, let $x_i \in \pi_i$ be a representative of this class, and let $X_\Pi := \{x_1, \ldots, x_k\}$ be the set of these representatives. The substitution that replaces, for all $i = 1, \ldots, k$, each element of π_i by its representative x_i is denoted by σ_Π.

3. *Test the pure formulae for satisfiability in the respective theories.*
   If $\sigma_{\Pi}(\varphi_i) \wedge \Delta(X_{\Pi})$ is unsatisfiable in T_i for $i = 1$ or $i = 2$, then return "unsatisfiable;" otherwise return "satisfiable."[5]

Obviously, it is possible to choose one partition using only a polynomial number of binary choices, which shows that the above procedure is indeed nondeterministic polynomial. Completeness is again easy to see, and soundness is an immediate consequence of Proposition 2.

Theorem 5. *Let T_1 and T_2 be two stably infinite theories over disjoint signatures such that the universal theory of T_i is decidable in NP for $i = 1, 2$. Then the universal theory of $T_1 \cup T_2$ is also decidable in NP.*

Extensions and related work. Shostak [Sho84] describes a more efficient combination procedure, which is based on congruence closure [NO80]. However, unlike the Nelson-Oppen procedure, this approach assumes that decision procedures of a specific form (so-called "canonizers" and "solvers") exist for the component theories, i.e., is does not treat the component decision procedures as black boxes. A formally more rigorous presentation of the method can be found in [CLS96].

Above, we have always assumed that the theories to be combined are over disjoint signatures. Without any such restriction, a general combination procedure cannot exist. In fact, it is very easy to find examples of decidable theories whose (non-disjoint) combination is undecidable. Nevertheless, it is worthwhile to try to weaken the disjointness assumption. The first extension of Nelson and Oppen's approach in this direction is due to Ringeissen [Rin96]. It was further extended by Ringeissen and Tinelli [TR98, Tin99] to the case of theories sharing "constructors."

Baader and Tinelli [BT97] consider the application of the Nelson-Oppen procedure to equational theories. Although equational theories need not be stably infinite (see Example 2), Nelson and Oppen's procedure can be applied, after some minor modifications, to combine decision procedures for the validity of quantifier-free formulae in equational theories. It is also shown that, contrary to a common belief, the method cannot be used to combine decision procedures for the word problem. The paper then presents a method that solves this kind of combination problem. In [BT99, BT00] it is shown that this approach can also be extended to the case of theories sharing constructors.

3.4 Combination of *E*-Unification Algorithms

The constraints that we treat in this section are E-unification problems, i.e., systems of term equations that must be solved modulo a given equational

[5] Recall that $\Delta(X_{\Pi})$ denotes the conjunction of all negated equations $x \neq y$ for distinct variables $x, y \in X_{\Pi}$.

theory E. From a more logical point of view, this means that we are interested in the positive existential or the positive AE theory of E (see Theorems 1 and 2), depending on whether we consider elementary unification or unification with constants. During the last three decades, research in unification theory has produced E-unification algorithms (i.e., constraint solvers for E-unification constraints) for a great variety of equational theories E (see [JK91, BS94, BS00]). Such an algorithm either actually computes solutions of the E-unification constraints (usually complete sets of E-unifiers), or it just decides satisfiability of E-unification constraints. Using decision procedures instead of algorithms computing complete sets of unifiers may be advantageous for theories where the complete sets are large or even infinite.

E-unification algorithms that compute complete sets of unifiers are, e.g., applied in theorem proving with "built in" theories (see, e.g., [Plo72, Sti85]), in generalizations of the Knuth-Bendix completion procedure to rewriting modulo theories (see, e.g., [JK86, Bac91]), and in logic programming with equality (see, e.g., [JLM84]). With the development of constraint approaches to theorem proving (see, e.g., [Bür91, NR94]), term rewriting (see, e.g., [KK89]), and logic programming (see, e.g., [JL87, Col90]), decision procedures for E-unification have been gaining in importance.

The combination problem for E-unification algorithms is directly motivated by these applications. In this section, we first motivate the problem and briefly review the research on this topic, which led to a complete solution for the case of theories over disjoint signatures. Subsequently, we describe the combination algorithm developed in [BS92, BS96], and sketch how to prove its correctness. In Section 3.5, we derive an algebraic and logical reformulation of the combination problem and the combination algorithm. This leads to a more abstract proof, which can also be generalized to other classes of constraints (Section 3.6). Finally, in Section 3.7 we comment on the complexity of the combination problem, and describe possible optimizations of the combination procedure.

3.4.1 The Problem and Its History

Basically, the problem of combining E-unification algorithms can be described as follows:

> *Assume we are given unification algorithms for solving unification problems modulo the equational theories E_1 and E_2. How can we obtain a unification algorithm for the union of the theories, $E_1 \cup E_2$?*

Here, unification algorithms may either be algorithms computing complete sets of unifiers or decision procedures for unifiability.

The relevance of this problem relies on the observation that, quite often, a given E-unification algorithm can only treat unification problems where the terms occurring in the problem are composed over the signature of E (elementary E-unification), possibly enriched by some free constants (E-unification with constants). This is, for example, the case for the "natural" unification algorithms for the theory AC of an associative-commutative function symbol [Sti81, HS87], which depend on solving linear Diophantine equations in the natural numbers. However, in the applications mentioned above, unification problems often contain "mixed" terms, i.e, terms that are constructed from function symbols belonging to different theories.

For example, in automated theorem proving, free function symbols of arbitrary arity are frequently introduced by Skolemization. Thus, the E-unification problems that must be solved there are usually *general* E-unification problems. If the given E-unification algorithm can treat only E-unification problems with constants, the question arises whether it is always possible to construct an algorithm for general E-unification from a given algorithm for E-unification with constants. This can be seen as an instance of the combination problem where E_1 is the theory E, and E_2 is the free theory for the free function symbols (e.g., consisting of the "dummy" identities $f(x_1, \dots, x_n) = f(x_1, \dots, x_n)$ for the free function symbols f). The combination problem in its general form arises if the semantic properties of several function symbols are to be integrated into the unification; for example, one may want to build in an associative symbol representing concatenation of lists, and an associative-commutative symbol representing addition of numbers.

Similarly as in the case of the Nelson-Oppen procedure, there cannot be a general solution to the combination problem as stated above: there exist equational theories E_1 and E_2 where unification with constants is decidable both for E_1 and for E_2, but solvability of unification problems with constants modulo $E_1 \cup E_2$ is undecidable. For example, both unification with constants modulo left-distributivity of the binary symbol f over the binary symbol g [TA87] and unification with constants modulo associativity of the binary symbol g [Mak77] are decidable, but unification with constants modulo the union of these theories is undecidable [SS89b]. Again, the restriction to theories over disjoint signatures avoids this problem. Until now, most of the research was concentrated on this restricted case.

A first important instance of the combination problem was considered by Stickel [Sti75, Sti81]. Stickel's AC-unification algorithm allowed for the presence of several AC-symbols and free symbols. However, termination of this algorithm could only be shown for restricted cases and it took almost a decade until Fages [Fag84] could close this gap.

Subsequently, more general combination problems were, e.g., treated in [Kir85, Tid86, Her86, Yel87, BJSS89], but the theories considered in these papers always had to satisfy certain restrictions (such as collapse-freeness or regularity) on the syntactic form of their defining identities. Recall that a theory E is called collapse-free if it does not contain an identity of the form $x = t$ where x is a variable and t is a non-variable term, and it is called regular if the left- and right-hand sides of the identities contain the same variables. Such restrictions simplify the combination problem, both from the conceptual and from the computational complexity point of view.

An important break-through in the research on the combination problem was the combination algorithm by Schmidt-Schauß [SS89a]. The algorithm applies to arbitrary equational theories over disjoint signatures, provided that an additional algorithmic requirement is fulfilled: in addition to algorithms for unification with constants, one needs algorithms that solve so-called constant elimination problems. A more efficient version of this highly nondeterministic algorithm has been described by Boudet [Bou93]. Basically, whereas the algorithm by Schmidt-Schauß performs two nondeterministic steps right at the beginning, Boudet's algorithm tries to defer nondeterministic steps as long as possible; nondeterminism is only used "on demand" to resolve certain conflicts. For restricted classes of theories (e.g., collapse-free theories) some of these conflicts cannot occur, and thus the corresponding nondeterministic steps can be avoided.

The combination procedures mentioned until now all considered the problem of combining algorithms that compute (finite) complete sets of E-unifiers. The problem of how to combine decision procedures is not solved by these approaches, in particular not for theories like associativity, where unification problems need not have a *finite* complete set of unifiers. Actually, the paper by Schmidt-Schauß [SS89a] also considered the problem of combining decision procedures. It showed that decision procedures can be combined, provided that solvability of *general* unification problems is decidable in the component theories. The drawback of this result was that for many theories (e.g., associativity) one already needs to employ combination methods to show that general unification (i.e., unification in the combination of the given theory with syntactic equality of the free function symbols) is decidable.

The problem of how to combine decision procedures was finally solved in a very general form in [BS92, BS96], where a combination algorithm was given that can be used both for combining decision procedures and for combining algorithms computing complete sets of unifiers. This algorithm applies to arbitrary equational theories over disjoint signatures, but it requires as a prerequisite that algorithms solving so-called unification problems *with linear constant restrictions*[6] are available for these theories (which was, e.g., the case for associativity). In the sequel, we describe this combination algorithm.

All the combination results that will be presented in the following are restricted to the case of disjoint signatures. There are some approaches that try to weaken the disjointness assumption, but the theories to be combined must satisfy rather strong conditions [Rin92, DKR94].

[6] In retrospect, if one looks at the combination method for decision procedures by Schmidt-Schauß, then one sees that he used the free function symbols in the general unification problems just to encode linear constant restrictions.

3.4.2 A Combination Algorithm for *E*-Unification Algorithms

In the following, we consider equational theories E_1 and E_2 over the disjoint signatures Σ_1 and Σ_2. We denote the union of the theories by $E := E_1 \cup E_2$ and the union of the signatures by $\Sigma := \Sigma_1 \cup \Sigma_2$. The theories E_1, E_2 are called the component theories of the combined theory E.

Before describing the algorithm in detail, we motivate its central steps and the ideas underlying these steps by examples. In these examples, E_1 will be the theory of a binary associative function symbol f, i.e., $E_1 := \{f(f(x,y),z) = f(x,f(y,z))\}$, and E_2 will be the free theory for the unary function symbol g and the constant symbol a, .e., $E_2 := \{g(x) = g(x),\ a = a\}$.

Assume that we are given an elementary E-unification problem Γ. First, we proceed in the same way as in the Nelson-Oppen procedure: using variable abstraction, we decompose Γ into a union of two pure unification problems, i.e., we compute an equivalent system of equations of the form[7] $\Gamma_1 \uplus \Gamma_2$, where Γ_i contains only terms over the signature Σ_i $(i = 1, 2)$.

Again, it is easy to see that an E-unifier σ of the original problem Γ is also an E-unifiers of the problems Γ_1 and Γ_2. By using an appropriate projection technique, which replaces alien subterms by variables, σ can be turned into an E_i-unifier of Γ_i (see Subsection 3.4.4 for more details). As in the case of the Nelson-Oppen procedure, the other direction need not be true: given solutions σ_1 and σ_2 of Γ_1 and Γ_2, it is not clear how to combine them into a solution of $\Gamma_1 \uplus \Gamma_2$.

An obvious problem that one may encounter is that the substitutions σ_1 and σ_2 may assign different terms to the same variable.

Example 3. Consider the decomposed problem $\Gamma_1 \uplus \Gamma_2$, where

$$\Gamma_1 := \{x =^? f(z,z)\} \quad \text{and} \quad \Gamma_2 := \{x =^? a\}.$$

The substitution $\sigma_1 := \{x \mapsto f(z,z)\}$ solves Γ_1 and $\sigma_2 := \{x \mapsto a\}$ solves Γ_2. Thus, there are conflicting assignment for the variable x, and it is not clear how to combine the two substitutions into a single one. In fact, in the example, there is no solution for the combined problem $\Gamma_1 \uplus \Gamma_2$.

This problem motivates the following step of the combination algorithm.

> *Theory labeling.* Given $\Gamma_1 \uplus \Gamma_2$, we (nondeterministically) introduce for each variable occurring in this problem a label 1 or 2. The label 1 for variable x indicates that x may be instantiated by solutions of the E_1-unification problem Γ_1, whereas it must be treated as a constant by solutions of the E_2-unification problem Γ_2. Label 2 is interpreted in the dual way.

[7] The symbol "$\uplus$" indicates that this union is disjoint.

In the example, the variable x must be assigned either label 1 or 2. In the first case, Γ_2 does not have a solution, whereas in the second case Γ_1 does not have a solution.

Unfortunately, this step introduces a new problem. Some of the new free "constants" generated by the labeling step may need to be identified by a solution of the subproblem, but this is no longer possible since they are constants, and thus cannot be replaced.

Example 4. Consider the decomposed problem $\Gamma_1 \uplus \Gamma_2$, where

$$\Gamma_1 := \{x =^? f(z,z), y =^? f(z,z)\} \text{ and } \Gamma_2 := \{g(x) =^? g(y)\}.$$

Obviously, Γ_1 can only be solved if both x and y obtain label 1. However, then Γ_2 does not have a solution since x and y are viewed as different constants in Γ_2. Nevertheless, $\Gamma_1 \uplus \Gamma_2$ has a solution, namely $\sigma := \{x \mapsto f(z,z),\ y \mapsto f(z,z)\}$.

There is, however, a simple solution to this problem.

> *Variable identification.* Before introducing the theory labeling, variables are (nondeterministically) identified.[8]

In the example, we just identify x and y (e.g., by replacing all occurrences of y by x). The resulting problem $\Gamma_2' = \{g(x) =^? g(x)\}$ is now obviously solvable.

After we have performed the identification and the labeling step, we can be sure that given solutions σ_1 and σ_2 of Γ_1 and Γ_2 have a disjoint domain, and thus it makes sense to consider the substitution $\sigma_1 \cup \sigma_2$. Nevertheless, this substitution need not be a solution of $\Gamma_1 \uplus \Gamma_2$, as illustrated by the following example.

Example 5. Consider the decomposed problem $\Gamma_1 \uplus \Gamma_2$, where

$$\Gamma_1 := \{x = f(z,z)\} \text{ and } \Gamma_2 := \{z =^? a\}.$$

We assume that no variables are identified and that x obtains label 1 and z label 2. Then $\sigma_1 := \{x \mapsto f(z,z)\}$ solves Γ_1 and $\sigma_2 := \{z \mapsto a\}$ solves Γ_2. However, $\sigma := \sigma_1 \cup \sigma_2 = \{x \mapsto f(z,z),\ z \mapsto a\}$ does not solve $\Gamma_1 \uplus \Gamma_2$ since $\sigma(x) = f(z,z)$ and $\sigma(f(z,z)) = f(a,a)$.

To avoid this kind of problem, we must iteratively apply the substitutions σ_1 and σ_2 to each other, i.e., consider the sequence σ_1, $\sigma_2 \circ \sigma_1$, $\sigma_1 \circ \sigma_2 \circ \sigma_1$,[9] etc. until it stabilizes. In the above example, the correct combined substitution would be $\{x \mapsto f(a,a),\ z \mapsto a\} = \sigma_2 \circ \sigma_1 = \sigma_1 \circ \sigma_2 \circ \sigma_1 = \dots$. In general, this process need not terminate since there may be cyclic dependencies between the variable instantiations.

[8] This is a step that also occurs in the nondeterministic variant of the Nelson-Oppen procedure.

[9] The symbol "$\circ$" denotes composition of substitution, where the substitution on the right is applied first.

Example 6. Consider the decomposed problem $\Gamma_1 \uplus \Gamma_2$, where

$$\Gamma_1 := \{x = f(z, z)\} \text{ and } \Gamma_2 := \{z =^? g(x)\}.$$

We assume that no variables are identified and that x obtains label 1 and z label 2. Then $\sigma_1 := \{x \mapsto f(z, z)\}$ solves Γ_1 and $\sigma_2 := \{z \mapsto g(x)\}$ solves Γ_2. Because x is replaced by a term containing y and y by a term containing x, iterated application of the substitutions to each other does not terminate. In fact, it is easy to see that the combined problem $\Gamma_1 \uplus \Gamma_2$ does not have a solution.

In the combination procedure, such cyclic dependencies between solutions of the component problems are prohibited by the following step:

> *Linear ordering.* We (nondeterministically) choose a linear ordering $<$ on the variables occurring in $\Gamma_1 \uplus \Gamma_2$. Given a labeling of variables as explained above, we ask for solutions σ_i of Γ_i $(i = 1, 2)$ that respect the labeling in the sense introduced above, and satisfy the following additional condition: if a variable y with label j occurs in $\sigma_i(x)$, where x has label $i \neq j$, then $y < x$.

As a consequence of these steps, the E_i-unification problems obtained as output of the combination procedure are no longer elementary E_i-unification problems. Because of the labeling, they contain free constants (the variables with label $j \neq i$), and the linear ordering imposes additional restrictions on the possible solutions. We call such a problem an E_i-unification problem with linear constant restrictions.

Definition 2. Let F be an equational theory. An *F-unification problem with linear constant restriction* is a quadruple $(\Gamma, X, C, <)$. The first component, Γ, is an elementary F-unification problem. X and C are disjoint finite sets such that $X \cup C$ is a superset of the set of variables occurring in Γ. The last component, $<$, is a linear ordering on $X \cup C$. A *solution* of $(\Gamma, X, C, <)$ is a substitution σ that satisfies the following conditions:

1. σ solves the elementary F-unification problem Γ;
2. σ treats all elements of C as constants, i.e., $\sigma(x) = x$ for all $x \in C$;
3. for all $x \in X$ and $c \in C$, if $x < c$, then c must not occur in $\sigma(x)$.

We are now ready to give a formal description of the combination algorithm in Fig. 3.1. As before, we restrict the description to the combination of two component algorithms. It should be noted, however, that the generalization to $n > 2$ theories would be straightforward. Since Steps 2–4 are nondeterministic, the procedure actually generates a finite set of possible output pairs. The following proposition shows that the combination algorithm is sound and complete if used as a scheme for combining E_i-unification algorithms that decide solvability. A sketch of a proof will be given later on.

Input: A finite set Γ of equations between $(\Sigma_1 \cup \Sigma_2)$-terms. The following steps are applied in consecutive order.

1. Decomposition.
Using variable abstraction, we compute an equivalent system $\Gamma_1 \uplus \Gamma_2$ where Γ_1 only contains pure Σ_1-terms and Γ_2 only contains pure Σ_2-terms.

2. Choose Variable Identification.
A partition Π of the set of variables occurring in $\Gamma_1 \uplus \Gamma_2$ is chosen, and for each equivalence class of Π a representative is selected. If y is the representative of $\pi \in \Pi$ and $x \in \pi$ we say that y is the representative of x. Let Y denote the set of all representatives. Now each variable is replaced by its representative. We obtain the new system $\Gamma_1' \uplus \Gamma_2'$.

3. Choose Theory Labeling.
A labeling function $Lab : Y \to \{1, 2\}$ is chosen. Let Y_1 and Y_2 respectively denote the set of variables with label 1 and 2.

4. Choose Linear Ordering.
A linear ordering "$<$" on Y is selected.

Output: The pair $((\Gamma_1', Y_1, Y_2, <), (\Gamma_2', Y_2, Y_1, <))$.
Each component $(\Gamma_i', Y_i, Y_j, <)$ is treated as an E_i-unification problem with linear constant restriction $(i = 1, 2)$.

Fig. 3.1. The Combination Algorithm

Proposition 3. *The input problem, Γ, has a solution iff there exists an output pair of the Combination Algorithm, $((\Gamma_1', Y_1, Y_2, <), (\Gamma_2', Y_2, Y_1, <))$, such that both components are solvable.*

3.4.3 Consequences

The straightforward generalization of Proposition 3 to $n \geq 2$ theories yields the following combination result for decision procedures.

Theorem 6. *Let $E_1, \ldots, E_n$ be equational theories over pairwise disjoint signatures such that solvability of E_i-unification problems with linear constant restrictions is decidable for $i = 1, \ldots, n$. Then unifiability is decidable for the combined theory $E := E_1 \cup \ldots \cup E_n$.*

By "unifiability" we mean here solvability of elementary E-unification problems. Since, for each set Ω of free function symbols, solvability of unification problems with linear constant restriction in the free theory $F_\Omega = \{f(\ldots) = f(\ldots) \mid f \in \Omega\}$ is decidable (see below), the result of Theorem 6 can also be lifted to general E-unification problems. In fact, given a general E-unification problem, Γ, we just have to apply the theorem to the theories $E_1, \ldots, E_n, F_\Omega$ where Ω denotes the set of free function symbols occurring in Γ. In Section 3.5.1 we shall see that E-unification problems with linear constant restrictions can always be encoded as E-unification problems with free function symbols. As a consequence, Theorem 6 also holds for E-unification problems with linear constant restrictions, which yields a modularity result for unification with linear constant restrictions.

A simple analysis of the (nondeterministic) steps of the Combination Algorithm also provides us with the following complexity result, which is analogous to the one of Theorem 5:

Theorem 7. *If solvability of E_i-unification problems with linear constant restrictions is decidable in NP, then unifiability in the combined theory $E_1 \cup E_2$ is also decidable in NP.*

Although it was designed for the purpose of combining decision procedures, the Combination Algorithm can also be used to compute complete sets of unifiers modulo the union of equational theories.

Theorem 8. *Let $E_1, \ldots, E_n$ be equational theories over pairwise disjoint signatures, and let $E := E_1 \cup \ldots \cup E_n$ be their union. Assume that we have unification algorithms that compute, for each E_i-unification problem with linear constant restrictions, a finite complete set of E_i-unifiers ($i = 1, \ldots, n$). Then we can compute a finite complete set of E-unifiers for each elementary E-unification problem.*

The main idea for proving this theorem (sketched here for the case of $n = 2$ theories) is as follows. In the proof of soundness of the combination algorithm (see Section 3.4.4 below), we will show how an arbitrary pair (σ_1, σ_2) of solutions of an output pair of the combination algorithm can be combined into a solution $\sigma_1 \oplus \sigma_2$ of the input problem (see also Example 5). Given a single output pair $((\Gamma_1', Y_1, Y_2, <), (\Gamma_2', Y_2, Y_1, <))$, one can compute complete sets of unifiers for the two component problems, and then combine the elements of these complete sets in all possible ways. If this is done for all output pairs, then the set of all combined solutions obtained this way is a complete set of unifiers for the input problem (see [BS96] for details).

The last two results can again be lifted from elementary unification problems to general unification problems and to unification problems with linear constant restrictions in the combined theory, which provides us with a modularity result.

In order to apply these general combination results to specific theories, one needs algorithms that can solve unification problems *with linear constant restrictions* for these theories. For regular theories, an algorithm for computing complete sets of unifiers for unification with constants can be used to obtain an algorithm for computing complete sets of unifiers for unification with linear constant restrictions: just remove the unifiers violating the constant restrictions from the complete set. In particular, since the free theory is obviously regular, one can test solvability of a unification problem with linear constant restrictions in the free theory by computing the most general unifier, and then checking whether this unifier satisfies the constant restrictions. For non-regular theories, this simple way of proceeding is not possible.

However, the constant elimination procedures required by the approach of Schmidt-Schauß can be used to turn complete sets of unifiers for unification with constants into complete sets of unifiers for unification with linear constant restrictions (see [BS96], Section 5.2, for details).

With respect to decision procedures, it has turned out that, for several interesting theories (e.g., the theory *AC* of an associative-commutative symbol or the theory *ACI* of an associative-commutative-idempotent symbol), the known decision procedures for unification with constants can easily be modified into algorithms for unification with linear constant restrictions [BS93]. In particular, it is easy to show that Theorem 7 applies to *AC* and *ACI*, which yields a simple proof that the decision problem for general *AC*- and *ACI*-unification is in NP. For the theory A of an associative function symbol, decidability of unification problems with linear constant restrictions is an easy consequence (see [BS93]) of a result by Schulz [Sch92] on a generalization of Makanin's decision procedure. As a consequence, general A-unification is also decidable (this problem had been open before the development of the combination algorithm presented above).

There are, however, also theories for which unification with linear constant restrictions is considerably harder than unification with constants. For example, it can be shown [Baa98] that Boolean unification with linear constant restrictions is PSPACE-complete whereas Boolean unification with constants is "only" Π_2^p-complete. Until now, it is not known whether there exists an equational theory for which unification with constants is decidable, but unification with linear constant restrictions is undecidable.

3.4.4 Correctness

In the remainder of this section, we sketch how to prove Proposition 3. The full proof can be found in [BS96].

To show *soundness* of the Combination Algorithm, it suffices to show that, for each output pair $((\Gamma_1', Y_1, Y_2, <), (\Gamma_2', Y_2, Y_1, <))$, a given pair of solutions (σ_1, σ_2) of the two components can be combined into a solution σ of $\Gamma_1' \uplus \Gamma_2'$, which is treated as an elementary E-unification problem here. In fact, this obviously implies that σ can be extended to a solution of the input problem Γ.

The combined solution σ is defined by induction on the linear ordering $<$. Assume that σ is defined for all variables $y \in Y$ that are smaller than $z \in Y$ with respect to $<$. Without loss of generality we may assume that $z \in Y_1$ has label 1 and that $\sigma_1(z)$ does not contain any variables from Y_1. Since σ_1 satisfies the linear constant restrictions, it follows that all labeled variables y occurring in $\sigma_1(z)$ are smaller than z with respect to "$<$", which implies that $\sigma(y)$ is defined by induction hypothesis. We define $\sigma(z) := \sigma(\sigma_1(z))$. It is easy to see that the substitution σ obtained in this way is an instance of both σ_1 and σ_2. It follows that σ is an E_i-unifier, and hence an E-unifier, of Γ_i' $(i = 1, 2)$. Consequently, σ is a solution of $\Gamma_1' \uplus \Gamma_2'$.

It is more difficult to prove the *completeness* part of Proposition 3. Basically, the proof proceeds as follows. A given solution σ of Γ is used to define suitable choices in the nondeterministic steps of the Combination Algorithm, i.e., choices that lead to an output pair where both components are solvable:

- at the variable identification step, two variables x and y are identified iff $\sigma(x) =_E \sigma(y)$. Obviously σ is a solution of the system $\Gamma_1' \uplus \Gamma_2'$ reached after this identification.
- at the labeling step, a representative y receives label 1 iff $\sigma(x)$ has a symbol from Σ_1 as topmost function symbol.
- the linear ordering "$<$" that is chosen is an arbitrary extension of the partial ordering that is induced by the subterm relationship of the σ-values of representatives.

However, this way of proceedings is correct only if the solution σ of the input problem is assumed to be normalized in a particular way. In [BS92], using so-called "unfailing completion," a (possibly infinite) canonical rewrite system R for the combined theory E is defined. For each variable x in the system $\Gamma_1' \uplus \Gamma_2'$ it is then assumed that $\sigma(x)$ is in R-normal form. Another possibility is to assume that the terms $\sigma(x)$ are in the so-called layer-reduced form [SS89a, KR94]. In principle, this normal form is obtained by applying collapse-equations as much as possible.

It remains to find, given a normalized solution σ, suitable solutions σ_1 and σ_2 of the output pair determined by the choices induced by σ. To define these solutions, a "projection technique" is introduced that transforms possibly mixed solution terms of the form $\sigma(y)$ to a pure Σ_i-terms $\sigma_i(y)$. Basically, to define $\sigma_i(y)$, "alien" subterms of $\sigma(y)$ (i.e., maximal subterms starting with a symbol not belonging to Σ_i) are replaced by new variables, while ensuring that E-equivalent subterms are replaced by the same variable. If $\sigma(y)$ itself is alien, then $\sigma_i(y) := y$, which ensures that variables with label $j \neq i$ are treated as free constants.

3.5 The Logical and Algebraic Perspective

In this section, we describe the problem of combining unification algorithms from a more logical and algebraic point of view. This leads to a modified description of the combination algorithm and to a new proof of its correctness. In the next section, we will show that the techniques developed in the present section allow us to lift the combination methodology to more general classes of constraints.

3.5.1 A Logical Reformulation of the Combination Algorithm

Theorems 1 and 2 show that elementary E-unification problems and E-unification problems with constants correspond to natural classes of logical decision problems. The question arises whether this classification can be extended to general E-unification problems and to E-unification problems with linear constant restrictions. The following theorem, which was first proved in [BS96], gives a positive answer to this question. In particular, it states that both problems correspond to the same class of logical formulae.

Theorem 9. *Let E be an equational theory with signature Σ, and V a countably infinite set of variables. Then the following statements are equivalent:*

1. *Solvability of E-unification problems with linear constant restrictions is decidable.*
2. *The positive theory of E is decidable.*
3. *The positive theory of $\mathcal{T}(\Sigma, V)/{=_E}$ is decidable.*
4. *Solvability of general E-unification problems is decidable.*

From a practical point of view, the theorem is interesting because it shows that any theory that can reasonably be integrated in a universal deductive machinery via unification can also be combined with other such theories. In fact, as mentioned at the beginning of Section 3.4.1, such an integration usually requires an algorithm for *general* unification. The theorem shows that such an algorithm also makes sure that the precondition for our combination method to apply—namely, the existence of an algorithm for unification with linear constant restrictions—are satisfied.[10]

Theorem 9, together with our combination result for decision procedures, yields the following modularity result for the decidability of positive theories:

Theorem 10. *Let $E_1, \ldots, E_n$ be equational theories over disjoint signatures. Then the positive theory of $E_1 \cup \ldots \cup E_n$ is decidable iff the positive theories of the component theories E_i are decidable, for $i = 1, \ldots, n$.*

In the following, we motivate the equivalences stated in Theorem 9 by sketching how the respective problems can be translated into each other (see [BS96] for a detailed proof of the theorem):

- Any E-unification problem with linear constant restrictions $(\Gamma, X, C, <)$ can be translated into a positive Σ-sentence ϕ_Γ as follows: both variables (i.e., elements of X) and free constants (i.e., elements of C) are treated as variables in this formula; the matrix of ϕ_Γ is the conjunction of all equations in Γ; and in the quantifier prefix, the elements of X are existentially quantified, the elements of C are universally quantified, and the order of the quantifications is given by the linear ordering $<$.

[10] Strictly speaking, the theorem makes this statement only for decision procedures. In [BS96] it is shown, however, that the equivalence between general unification and unification with linear constant restrictions also holds with respect to algorithms that compute complete sets of unifiers.

- The equivalence between 2) and 3) is due to the well-known fact that the E-free algebra with countably many generators is canonical for the positive theory of E [Mal73], i.e., a positive sentence is valid in $\mathcal{T}(\Sigma, V)/{=_E}$ iff it is valid in all models of E.
- Given a positive Σ-sentence ϕ, one first removes universal quantifiers by Skolemization. The positive existential sentence obtained this way may contain additional free function symbols, the Skolem functions. It can be transformed into a disjunction of conjunctive positive existential sentences, and each of the disjuncts can obviously be translated into a general E-unification problem.
- The combination method described in Section 3.4 can be used to reduce solvability of a given general E-unification problem to solvability of E-unification problems with linear constant restrictions.

As an example, consider the free theory $F_{\{g\}} := \{g(x) = g(x)\}$, and the $F_{\{g\}}$-unification problem with constants $\{x =^? g(c)\}$. If we add the constant restriction $x < c$, then this problem is not solvable (since any solution must substitute x by the term $g(c)$, which contains the constant c). However, under the restriction $c < x$ the problem is solvable. The following are the positive sentences and general unification problems obtained by translating these two unification problems with linear constant restrictions:

unification with lcr	*positive sentence*	*general unification*
$\{x =^? g(c)\},\ x < c$	$\exists x.\forall y.\ x = g(y)$	$\{x =^? g(h(x))\}$
$\{x =^? g(c)\},\ c < x$	$\forall y.\exists x.\ x = g(y)$	$\{x =^? g(d)\}$

For example, $\exists x.\forall y.\ x = g(y)$ is not valid in all models of $F_{\{g\}}$ since this formula says that g must be a constant function, which obviously does not follow from $F_{\{g\}}$. Correspondingly, $\{x = g(h(x))\}$ does not have a solution because it causes an occur-check failure during syntactic unification.

Returning now to the combination problem, let E_1 and E_2 be two nontrivial equational theories over disjoint signatures Σ_1 and Σ_2, let $E := E_1 \cup E_2$ denote the union of the theories and $\Sigma := \Sigma_1 \cup \Sigma_2$ the union of the signatures. Using the correspondence between elementary E-unification problems and existentially quantified conjunctions of equations for the input of the algorithm, and the correspondence between E_i-unification problems with linear constant restriction and positive sentences for the output components we obtain the reformulation of the Combination Algorithm shown in Fig. 3.2. The advantage of the new formulation is that it does no longer rely on notions and concepts that are specific to unification problems modulo equational theories, such as linear constant restrictions, which are quite technical restrictions on the form of the allowed solutions. Correctness follows from the following proposition.

Input: A $(\Sigma_1 \cup \Sigma_2)$-sentence of the form $\exists \boldsymbol{u}.\, \gamma$, where γ is a conjunction of equations between $(\Sigma_1 \cup \Sigma_2)$-terms and $\boldsymbol{u}$ is a finite sequence consisting of the variables occurring in γ. The following steps are applied in consecutive order.

1. Decomposition.
Using variable abstraction, compute an equivalent sentence $\exists \boldsymbol{v}.\, (\gamma_1 \wedge \gamma_2)$, where γ_i is a conjunction of equations between pure Σ_i-terms for $i = 1, 2$.

2. Choose Variable Identification.
A partition Π of the set of variables occurring in $\boldsymbol{v}$ is chosen, and for each equivalence class of Π a representative is selected. If v is the representative of $\pi \in \Pi$ and $u \in \pi$, then we say that v is the representative of u. Let W denote the set of all representatives. Now each variable is replaced by its representative both in the quantifier prefix and in the matrix. Multiple quantifications over the same variable in the prefix are discarded. We obtain the new sentence $\exists \boldsymbol{w}.\, (\gamma_1' \wedge \gamma_2')$.

3. Choose Labeling.
A labeling function $Lab : W \to \{1, 2\}$ is chosen.

4. Choose Linear Ordering.
A linear ordering "$<$" on W is selected.

Output: The pair

$$\alpha = \forall \boldsymbol{u}_1.\exists \boldsymbol{v}_1. \cdots \forall \boldsymbol{u}_k.\exists \boldsymbol{v}_k.\, \gamma_1' \text{ and } \beta = \exists \boldsymbol{u}_1.\forall \boldsymbol{v}_1. \cdots \exists \boldsymbol{u}_k.\forall \boldsymbol{v}_k.\, \gamma_2'.$$

Here $\boldsymbol{u}_1\boldsymbol{v}_1 \ldots \boldsymbol{u}_k\boldsymbol{v}_k$ is the unique re-ordering of W along $<$. The sequences $\boldsymbol{u}_i$ ($\boldsymbol{v}_i$) represent the blocks of variables with label 1 (label 2).

Fig. 3.2. The Combination Algorithm (Logical Reformulation)

Proposition 4. *The input sentence $\exists \boldsymbol{u}.\, \gamma$ holds in the combined quotient term algebra $\mathcal{T}(\Sigma_1 \cup \Sigma_2, V)/{=_{E_1 \cup E_2}}$ iff there exists an output pair (α, β) such that α holds in $\mathcal{T}(\Sigma_1, V)/{=_{E_1}}$ and β holds in $\mathcal{T}(\Sigma_2, V)/{=_{E_2}}$.*

Since the new combination algorithm is just a reformulation of the earlier version, Proposition 4 is a trivial consequence of Proposition 3.

The remainder of this section is devoted to giving an independent correctness proof for the logical version of the Combination Algorithm. The new proof will have some significant advantages: it is more abstract and less technical, and thus easier to generalize to larger classes of constraints.

3.5.2 Fusions of Free Algebras

The proof of soundness of the Nelson-Oppen combination procedure that we have presented in Section 3.3 depends on a very simple algebraic construction: the fusion of structures. Our goal is to adapt this algebraic approach to the task of proving correctness of the (logical reformulation of the) combination procedure for unification algorithms. At first sight, the input problems considered in the case of unification look like a special case of the problems accepted by the Nelson-Oppen procedure: they are (existentially quantified) conjunctions of equations.[11] The main difference between the two combination problems lies in the semantics of the constraints. In the case treated by Nelson and Oppen, the input constraint must be satisfied in *some* model of the combined theory $T_1 \cup T_2$, whereas in the case of unification algorithms the input constraint must be satisfied in the *free* model of the combined theory $E_1 \cup E_2$. In the proof of correctness this means that, for the Nelson-Oppen procedure, it is sufficient to show that the input constraint can be satisfied in an arbitrary fusion of a model of T_1 with a model of T_2. In the unification case, we must make sure that this fusion is in fact the $(E_1 \cup E_2)$-free algebra with countably infinitely many generators. Thus, given the E_1- and E_2-free algebras $\mathcal{B}_1 := \mathcal{T}(\Sigma_1, V)/{=_{E_1}}$ and $\mathcal{B}_2 := \mathcal{T}(\Sigma_2, V)/{=_{E_2}}$, respectively, we want to construct a fusion of both algebras that is (isomorphic to) the $(E_1 \cup E_2)$-free algebra $\mathcal{B} := T(\Sigma_1 \cup \Sigma_2, V)/{=_{E_1 \cup E_2}}$. This construction will be called the *amalgamation construction.*

In the sequel, as always in this section, we assume that the signatures Σ_1 and Σ_2 are disjoint, and that the theories E_1 and E_2 are nontrivial. For simplicity we shall identify each variable $x \in V$ with its equivalence class w.r.t. E_i in $\mathcal{B}_i$, i.e., write again x for the E_i-class $[x]_{E_i} = \{t \in T(\Sigma_i, V) \mid t =_{E_i} x\}$.

The construction starts with a preparatory step where we extend $\mathcal{B}_i$ to an E_i-free algebra $\mathcal{B}_i^\infty$ of the form $T(\Sigma_i, V \cup Y_i)/{=_{E_i}}$ where Y_i denotes a countably infinite set of additional variables $(i = 1, 2)$. Since the sets $V \cup Y_i$ and V have the same cardinality, B_1^∞ and B_2^∞ are isomorphic to $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively. We assume (without loss of generality) that $B_1^\infty \cap B_2^\infty = V$. These two algebras (as well as details of the amalgamation construction) are depicted in Fig. 3.3.

We shall now construct a bijection between the domains B_1^∞ and B_2^∞ of the extended algebras $\mathcal{B}_1^\infty$ and $\mathcal{B}_2^\infty$. This bijection will then be used to define a fusion of $\mathcal{B}_1^\infty$ and $\mathcal{B}_2^\infty$ (see Lemma 1), which is also a fusion of $\mathcal{B}_1$ and $\mathcal{B}_2$. Note, however, that we cannot use an arbitrary bijection between B_1^∞ and B_2^∞ since we want this fusion to be the $(E_1 \cup E_2)$-free algebra with countably infinitely many generators.

In the following, let us call an element of $B_i^\infty \setminus (V \cup Y_i)$ a *non-atomic* element of $\mathcal{B}_i^\infty$. The elements of $V \cup Y_i$ are called *atomic.* The crucial property that we want to obtain is that *non-atomic* elements of one side are always mapped to *atomic* elements of the other side. The definition of the bijection proceeds in an infinite series of zig-zag steps: at each step an existing partial bijection is extended by adding a new partial bijection with domain and image sets disjoint to the sets already used.

[11] The Nelson-Oppen procedure additionally allows for negation and for non-equational atoms.

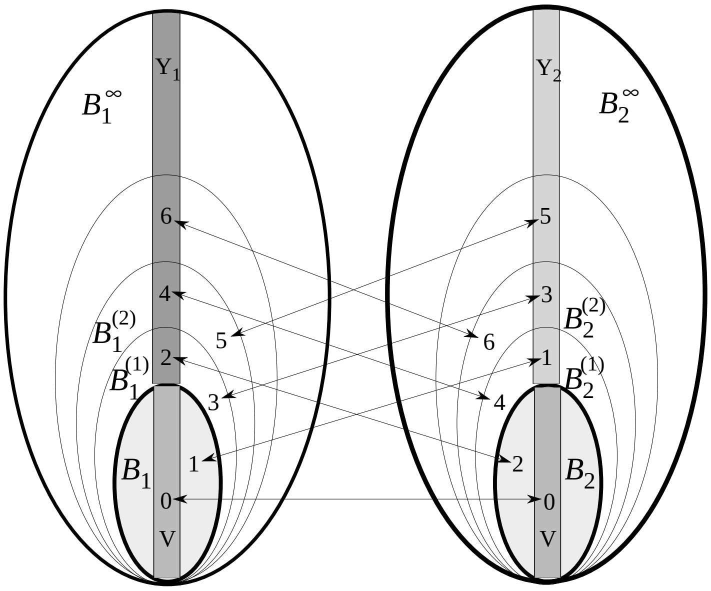


Fig. 3.3. The amalgamation construction.

In step 0 we use the identity mapping on V to obtain a bijection between the common set of generators of both sides (see areas 0 in Fig. 3.3). We say that the elements in V of both sides are now fibered.

In step 1 we assign suitable images to the elements of $B_1 \setminus V$ (see area 1 on the left-hand side). To this end, we select a set of atoms $Y_2^{(1)} \subset Y_2$ with the same cardinality as $B_1 \setminus V$ (area 1 on the right-hand side represents $Y_2^{(1)}$). The existing partial bijection is extended by adding a bijection between $B_1 \setminus V$ and $Y_2^{(1)}$ (indicated by a double arrow between the areas 1). We say that the elements in $B_1 \setminus V$ and $Y_2^{(1)}$ are now fibered as well. In step 1 and in the sequel, whenever we select a set of new atoms, we leave an infinite set of atoms untouched, which are thus available in subsequent steps of the construction.

In the symmetric step 2 we add a bijection between $B_2 \setminus V$ (area 2, right-hand side) and a suitable set of new atoms $Y_1^{(1)} \subset Y_1$ (area 2, left-hand side). With this step we say that now the elements in $B_2 \setminus V$ and $Y_1^{(1)}$ are fibered as well.

For $i = 1, 2$, let $\mathcal{B}_i^{(1)}$ denote the subalgebra of $\mathcal{B}_i^\infty$ that is generated by $V \cup Y_i^{(1)}$. The elements of $\mathcal{B}_1^{(1)}$ that do not yet have an image are fibered in step 3 using a fresh set of atoms $Y_2^{(2)}$ of the right-hand side (areas 3); in step 4 the elements of $B_2^{(1)}$ that do not yet have images are fibered using a fresh set of atoms $Y_1^{(2)}$ of left-hand side (areas 4).

For $i = 1, 2$, let $\mathcal{B}_i^{(2)}$ denote the subalgebra of $\mathcal{B}_i^\infty$ that is generated by $V \cup Y_i^{(1)} \cup Y_i^{(2)}$. We continue in the same way as above (areas $5, 6$), etc. The construction determines for $i = 1, 2$ an ascending tower of Σ_i-subalgebras

$$\mathcal{B}_i = \mathcal{B}_i^{(0)} \subseteq \mathcal{B}_i^{(1)} \subseteq \mathcal{B}_i^{(2)} \subseteq \dots$$

of $\mathcal{B}_i^\infty$. For simplicity we assume that the construction eventually covers each atom of both sides, hence we have $B_i^\infty = \bigcup_{k=0}^\infty B_i^{(k)}$. Since the limit bijection can be read in two directions we now have two inverse bijections

$$h_{1-2} : \mathcal{B}_1^\infty \to \mathcal{B}_2^\infty \quad \text{and} \quad h_{2-1} : \mathcal{B}_2^\infty \to \mathcal{B}_1^\infty.$$

As in the proof of Lemma 1, these bijections can be used to carry the Σ_i-structure of $\mathcal{B}_i^\infty$ to $\mathcal{B}_j^\infty$ (where $\{i, j\} = \{1, 2\}$). Let f be an n-ary function symbol of Σ_i and $b_1, \dots, b_n \in B_j^\infty$. We define

$$f^{\mathcal{B}_j^\infty}(b_1, \dots, b_n) := h_{i-j}(f^{\mathcal{B}_i^\infty}(h_{j-i}(b_1), \dots, h_{j-i}(b_n))).$$

With this definition, the mappings h_{1-2} and h_{2-1} are inverse isomorphisms between the $(\Sigma_1 \cup \Sigma_2)$-algebras obtained from $\mathcal{B}_1^\infty$ and $\mathcal{B}_2^\infty$ by means of the above signature expansion. For this reason, it is irrelevant which of the two algebras we take as the combined algebra. We take, say, the $(\Sigma_1 \cup \Sigma_2)$-algebra obtained from $\mathcal{B}_1^\infty$, and denote it by $\mathcal{B}_1 \oplus \mathcal{B}_2$.

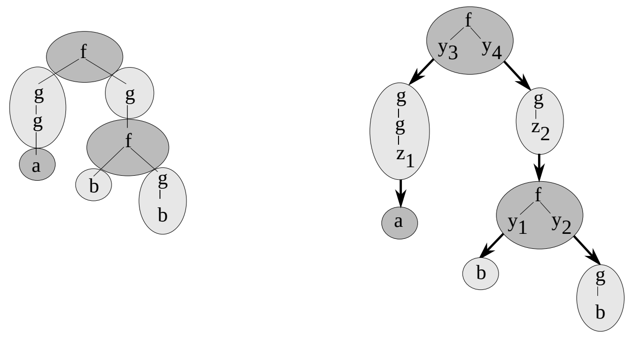
Recall that $\mathcal{B}_i$ and $\mathcal{B}_i^\infty$ are Σ_i-isomorphic algebras since both are free over a countably infinite set of generators for the class of all models of E_i. In addition, the construction makes sure that $\mathcal{B}_1 \oplus \mathcal{B}_2$ is Σ_i-isomorphic to $\mathcal{B}_i^\infty$ $(i = 1, 2)$, which yields the following lemma:

Lemma 2. *$\mathcal{B}_1 \oplus \mathcal{B}_2$ is a fusion of $\mathcal{B}_1$ and $\mathcal{B}_2$.*

More interestingly, the following theorem [BS98] shows that we have indeed found the desired description of $T(\Sigma_1 \cup \Sigma_2, V)/{=_{E_1 \cup E_2}}$ as a fusion of the component algebras $\mathcal{B}_1 = T(\Sigma_1, V)/{=_{E_1}}$ and $\mathcal{B}_2 = T(\Sigma_2, V)/{=_{E_2}}$:

Theorem 11. *$\mathcal{B}_1 \oplus \mathcal{B}_2$ is (isomorphic to) the $(E_1 \cup E_2)$-free algebra over a countably infinite set of generators.*

At first sight, it is perhaps not easy to see the relationship between the amalgamation construction and the usual description of $T(\Sigma_1 \cup \Sigma_2, V)/_{E_1 \cup E_2}$ in terms of $=_{(E_1 \cup E_2)}$-equivalence classes of terms (i.e., finite trees). In order to illustrate this connection, let us look at the simplest possible case where we combine two absolutely free algebras, i.e., where E_1 and E_2 are free theories. Let us assume that $\Sigma_1 = \{f, a\}$ and $\Sigma_2 = \{g, b\}$, where f is binary, g is unary and a, b are constants. The following figure depicts, on the left-hand side, an element of the combined domain using the conventional description as a finite tree. Subparts belonging to distinct signatures are highlighted accordingly.



The "leaf" elements a and $b, g(b)$ correspond to elements of $\mathcal{B}_1$ and $\mathcal{B}_2$ that are fibered in steps 1 and 2 of the construction, say, with atoms z_1 and y_1, y_2, respectively. Thus, the subtree $f(b, g(b))$ corresponds to the element $f^{\mathcal{B}_1^\infty}(y_1, y_2)$ of $B_1^{(1)}$, and $g(g(a))$ to the element $g^{\mathcal{B}_2^\infty}(g^{\mathcal{B}_2^\infty}(z_1))$ of $B_2^{(1)}$. These elements are fibered with new atoms (say z_2 and y_3) in the steps 2 and 3. The subtree $g(f(b, g(b)))$ corresponds to an element of $B_2^{(2)}$, which is fibered by a new atom (say y_4) in step 6. Finally, the complete tree $f(g(g(a)), g(f(b, g(b))))$ corresponds to an element of $B_1^{(3)}$. On the right-hand side of the figure, we have represented all elements of the fusion that are involved in the representation of the complete tree and made the fibering bijections explicit using arrows. Due to the inductive form of the construction, the elements of the fusion can be considered as generalized "finite trees" where nodes represent elements of the two components, and links represent ordered pairs of the fibering function.

If E_1 and E_2 are more interesting equational theories, the relationship between a given mixed term and the corresponding element of $\mathcal{B}_1 \oplus \mathcal{B}_2$ may be less obvious. For example, if E_2 contains the collapse axiom $g(x) = x$, then $g(g(a))$ is equivalent to a, and thus the corresponding element belongs to B_1, and not to $B_2^{(1)}$. A similar phenomenon occurs in the presence of non-regular axioms. For example, if E_1 is the free theory, then $f(b, g(b))$ corresponds to an element of $B_1^{(1)}$. However, if E_1 contains the (non-regular) axiom $f(x, y) = a$, then $f(b, g(b))$ is equivalent to a, and the corresponding element belongs to B_1. This issue is closely related to the fact that, in the proof of completeness of Proposition 3, we needed a normalized substitution. Given a mixed term that is normalized by the (possibly infinite) canonical rewrite system R, the simple syntactic correspondence between subtrees of this term and elements of $\mathcal{B}_1 \oplus \mathcal{B}_2$ holds also for theories that are not regular and collapse-free.

In the next subsection, we will use the new description of the combined algebra $\mathcal{T}(\Sigma_1 \cup \Sigma_2, V)/{=_{E_1 \cup E_2}}$ as a fusion $\mathcal{B}_1 \oplus \mathcal{B}_2$ to show correctness of the combination algorithm in its logical reformulation.

3.5.3 Correctness of the Combination Algorithm

First, we show soundness of the Combination Algorithm (logical formulation). In the following, boldface letters like $\boldsymbol{u}, \boldsymbol{v}$ and $\boldsymbol{b}, \boldsymbol{d}$ (possibly with subscripts) will respectively denote finite sequences of variables and algebra elements. An expression like $\boldsymbol{b} \in \boldsymbol{B}$ expresses that $\boldsymbol{b}$ is a sequence of elements of B, which denotes the carrier set of the algebra $\mathcal{B}$. We denote by $h(\boldsymbol{b})$ the result of applying the homomorphism h to the sequence $\boldsymbol{b}$, i.e., the sequence consisting of the components $h(b)$ for all components b of $\boldsymbol{b}$.

Lemma 3. *Let $\exists \boldsymbol{u}.\,\gamma$ be an input sentence of the Combination Algorithm. Then $\mathcal{B}_1 \oplus \mathcal{B}_2 \models \exists \boldsymbol{u}.\,\gamma$ if $\mathcal{B}_1 \models \alpha$ and $\mathcal{B}_2 \models \beta$ for some output pair (α, β).*

Proof. Since $\mathcal{B}_1$ and $\mathcal{B}_1^\infty$ are isomorphic Σ_i-algebras, we know that $\mathcal{B}_1^\infty \models \alpha$. Accordingly, we also have $\mathcal{B}_2^\infty \models \beta$. More precisely, this means

$$(*)\quad \mathcal{B}_1^\infty \models \forall \boldsymbol{u}_1.\exists \boldsymbol{v}_1.\cdots\forall \boldsymbol{u}_k.\exists \boldsymbol{v}_k.\,\gamma_1'(\boldsymbol{u}_1, \boldsymbol{v}_1, \ldots, \boldsymbol{u}_k, \boldsymbol{v}_k),$$
$$(**)\quad \mathcal{B}_2^\infty \models \exists \boldsymbol{u}_1.\forall \boldsymbol{v}_1.\cdots\exists \boldsymbol{u}_k.\forall \boldsymbol{v}_k.\,\gamma_2'(\boldsymbol{u}_1, \boldsymbol{v}_1, \ldots, \boldsymbol{u}_k, \boldsymbol{v}_k).$$

Because of the existential quantification over $\boldsymbol{u}_1$ in $(**)$, there exists a sequence $\boldsymbol{b}_1 \in \boldsymbol{B}_2^\infty$ such that

$$(***)\quad \mathcal{B}_2^\infty \models \forall \boldsymbol{v}_1.\cdots\exists \boldsymbol{u}_k.\forall \boldsymbol{v}_k.\,\gamma_2'(\boldsymbol{b}_1, \boldsymbol{v}_1, \ldots, \boldsymbol{u}_k, \boldsymbol{v}_k).$$

We consider $\boldsymbol{a}_1 := h_{2-1}(\boldsymbol{b}_1)$. Because of the universal quantification over $\boldsymbol{u}_1$ in $(*)$ we have

$$\mathcal{B}_1^\infty \models \exists \boldsymbol{v}_1.\cdots\forall \boldsymbol{u}_k.\exists \boldsymbol{v}_k.\,\gamma_1'(\boldsymbol{a}_1, \boldsymbol{v}_1, \ldots, \boldsymbol{u}_k, \boldsymbol{v}_k).$$

Because of the existential quantification over $\boldsymbol{v}_1$ in this formula there exists a sequence $\boldsymbol{c}_1 \in \boldsymbol{B}_1^\infty$ such that

$$\mathcal{B}_1^\infty \models \forall \boldsymbol{u}_2.\exists \boldsymbol{v}_2.\cdots\forall \boldsymbol{u}_k.\exists \boldsymbol{v}_k.\,\gamma_1'(\boldsymbol{a}_1, \boldsymbol{c}_1, \boldsymbol{u}_2, \boldsymbol{v}_2, \ldots, \boldsymbol{u}_k, \boldsymbol{v}_k).$$

We consider $\boldsymbol{d}_1 := h_{1-2}(\boldsymbol{c}_1)$. Because of the universal quantification over $\boldsymbol{v}_1$ in $(***)$ we have

$$\mathcal{B}_2^\infty \models \exists \boldsymbol{u}_2.\forall \boldsymbol{v}_2.\cdots\exists \boldsymbol{u}_k.\forall \boldsymbol{v}_k.\,\gamma_2'(\boldsymbol{b}_1, \boldsymbol{d}_1, \boldsymbol{u}_2, \boldsymbol{v}_2, \ldots, \boldsymbol{u}_k, \boldsymbol{v}_k).$$

Iterating this argument, we thus obtain

$$\mathcal{B}_1^\infty \models \gamma_1'(\boldsymbol{a}_1, \boldsymbol{c}_1, \ldots, \boldsymbol{a}_k, \boldsymbol{c}_k),$$
$$\mathcal{B}_2^\infty \models \gamma_2'(\boldsymbol{b}_1, \boldsymbol{d}_1, \ldots, \boldsymbol{b}_k, \boldsymbol{d}_k),$$

where $\boldsymbol{a}_i = h_{2-1}(\boldsymbol{b}_i)$ and $\boldsymbol{d}_i = h_{1-2}(\boldsymbol{c}_i)$ (for $1 \leq i \leq k$). Since h_{1-2} and h_{2-1} are inverse $(\Sigma_1 \cup \Sigma_2)$-isomorphisms we also know that

$$\mathcal{B}_1^\infty \models \gamma_2'(\boldsymbol{a}_1, \boldsymbol{c}_1, \ldots, \boldsymbol{a}_k, \boldsymbol{c}_k).$$

It follows that

$$\mathcal{B}_1 \oplus \mathcal{B}_2 \models \gamma_1'(\boldsymbol{a}_1, \boldsymbol{c}_1, \ldots, \boldsymbol{a}_k, \boldsymbol{c}_k) \wedge \gamma_2'(\boldsymbol{a}_1, \boldsymbol{c}_1, \ldots, \boldsymbol{a}_k, \boldsymbol{c}_k).$$

Obviously, this implies that $\mathcal{B}_1 \oplus \mathcal{B}_2 \models \exists \boldsymbol{v}.\ (\gamma_1' \wedge \gamma_2')$, i.e., the sentences obtained after Step 1 of the algorithm holds in $\mathcal{B}_1 \oplus \mathcal{B}_2$. It is easy to see that this implies that $\mathcal{B}_1 \oplus \mathcal{B}_2 \models \exists \boldsymbol{u}.\, \gamma$. □

Before we can show completeness of the decomposition algorithm, we need one more prerequisite. The following lemma characterizes validity of positive sentences in free algebras in terms of satisfying assignments. The proof of this lemma, which uses the well-known fact that validity of positive formulae is preserved under surjective homomorphisms, is not difficult and can be found in [BS98] for the more general case of quasi-free structures.

Lemma 4. *Let $\mathcal{A} = \mathcal{T}(\Delta, V)/{=_E}$ be the E-free Δ-algebra over the countably infinite set of generators V, and let*

$$\gamma = \forall \boldsymbol{u}_1.\exists \boldsymbol{v}_1. \cdots \forall \boldsymbol{u}_k.\exists \boldsymbol{v}_k.\, \varphi(\boldsymbol{u}_1, \boldsymbol{v}_1, \ldots, \boldsymbol{u}_k, \boldsymbol{v}_k)$$

be a positive Δ-sentence. Then the following conditions are equivalent:

1. *$\mathcal{A} \models \forall \boldsymbol{u}_1.\exists \boldsymbol{v}_1. \cdots \forall \boldsymbol{u}_k.\exists \boldsymbol{v}_k.\, \varphi(\boldsymbol{u}_1, \boldsymbol{v}_1, \ldots, \boldsymbol{u}_k, \boldsymbol{v}_k)$.*
2. *There exist tuples $\boldsymbol{x}_1 \in \boldsymbol{V}, \boldsymbol{e}_1 \in \boldsymbol{A}, \ldots, \boldsymbol{x}_k \in \boldsymbol{V}, \boldsymbol{e}_k \in \boldsymbol{A}$ and finite subsets $Z_1, \ldots, Z_k$ of V such that*
   (a) $\mathcal{A} \models \varphi(\boldsymbol{x}_1, \boldsymbol{e}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{e}_k)$,
   (b) all generators occurring in the tuples $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_k$ are distinct,
   (c) for all $j, 1 \leq j \leq k$, the components of $\boldsymbol{e}_j$ are generated by Z_j, i.e., they belong to $\mathcal{T}(\Delta, Z_j)/{=_E}$
   (d) for all $j, 1 < j \leq k$, no component of $\boldsymbol{x}_j$ occurs in $Z_1 \cup \ldots \cup Z_{j-1}$.

Using this lemma, we can now prove completeness of the Combination Algorithm (logical reformulation).

Lemma 5. *Let $\exists \boldsymbol{u}.\, \gamma$ be an input sentence of the Combination Algorithm. If $\mathcal{B}_1 \oplus \mathcal{B}_2 \models \exists \boldsymbol{u}.\, \gamma$ then there exists an output pair with components α and β such that $\mathcal{B}_1 \models \alpha$ and $\mathcal{B}_2 \models \beta$.*

Proof. Assume that $\mathcal{B}_1^\infty \simeq \mathcal{B}_1 \oplus \mathcal{B}_2 \models \exists \boldsymbol{u}_0.\, \gamma_0$.[12] Obviously, this implies that $\mathcal{B}_1^\infty \models \exists \boldsymbol{v}.\ (\gamma_1(\boldsymbol{v}) \wedge \gamma_2(\boldsymbol{v}))$, i.e., $\mathcal{B}_1^\infty$ satisfies the sentence that is obtained after Step 2 of the Combination Algorithm. Thus there exists an assignment $\nu : V \to B_1^\infty$ such that $\mathcal{B}_1^\infty \models \gamma_1(\nu(\boldsymbol{v})) \wedge \gamma_2(\nu(\boldsymbol{v}))$.

In Step 3 of the decomposition algorithm we identify two variables u and u' of $\boldsymbol{v}$ if, and only if, $\nu(u) = \nu(u')$. With this choice, the assignment ν satisfies the formula obtained after the identification step, i.e.,

$$\mathcal{B}_1^\infty \models \gamma_1'(\nu(\boldsymbol{w})) \wedge \gamma_2'(\nu(\boldsymbol{w})),$$

[12] Here and in the sequel, $\mathcal{B}_1^\infty$ is sometimes treated as a $(\Sigma_1 \cup \Sigma_2)$-algebra, using the signature expansion described in the construction of $\mathcal{B}_1 \oplus \mathcal{B}_2$.

and all components of $\nu(\boldsymbol{w})$ are distinct.

In Step 4, a variable w in $\boldsymbol{w}$ is labeled with 2 if $\nu(w) \in Y_1$, and with 1 otherwise. In order to choose the linear ordering on the variables, we partition the range B_1^∞ of ν as follows:

$$B_1^{(0)},\ \ Y_1^{(1)},\ \ B_1^{(1)} \setminus (B_1^{(0)} \cup Y_1^{(1)}),\ \ Y_1^{(2)},\ \ B_1^{(2)} \setminus (B_1^{(0)} \cup Y_1^{(2)}),$$
$$Y_1^{(3)},\ \ B_1^{(3)} \setminus (B_1^{(2)} \cup Y_3),\ \ldots$$

In Fig. 3.3 these subsets correspond to the areas (0 and 1), 2, 3, 4, 5, 6,... of the left-hand side. Now, let $\boldsymbol{u}_1, \boldsymbol{v}_1, \ldots, \boldsymbol{u}_k, \boldsymbol{v}_k$ be a re-ordering of the tuple $\boldsymbol{w}$ such that the following holds:

1. The tuple $\boldsymbol{u}_1$ contains exactly the variables whose ν-images are in $B_1^{(0)}$.
2. For all $i, 1 \leq i \leq k$, the tuple $\boldsymbol{v}_i$ contains exactly the variables whose ν-images are in $Y_1^{(i)}$.
3. For all $i, 1 < i \leq k$, the tuple $\boldsymbol{u}_i$ contains exactly the variables whose ν-images are in $B_1^{(i-1)} \setminus (B_1^{(i-2)} \cup Y_1^{(i-1)})$.

Obviously, this implies that the variables in the tuples $\boldsymbol{v}_i$ have label 2, whereas the variables in the tuples $\boldsymbol{u}_i$ have label 1. Note that some of these tuples may be of dimension 0. This re-ordering of $\boldsymbol{w}$ determines the linear ordering we choose in Step 4. Let

$$\alpha = \forall \boldsymbol{u}_1.\exists \boldsymbol{v}_1.\cdots \forall \boldsymbol{u}_k.\exists \boldsymbol{v}_k.\,\gamma_1' \quad \text{and} \quad \beta = \exists \boldsymbol{u}_1.\forall \boldsymbol{v}_1.\cdots \exists \boldsymbol{u}_k.\forall \boldsymbol{v}_k.\,\gamma_2'$$

be the output pair that is obtained by these choices. Let $\boldsymbol{x}_i := \nu(\boldsymbol{w}_i)$ and $\boldsymbol{e}_i := \nu(\boldsymbol{v}_i)$. For $i = 1, \ldots, k$, let Z_i denote a finite set of variables in $B_1^{(i-1)} \cap (V \cup Y_1)$ that generates all elements in $\boldsymbol{e}_i$. We claim that the sequence $\boldsymbol{x}_1, \boldsymbol{e}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{e}_k$ and the sets $Z_1, \ldots, Z_k$ satisfy Condition 2 of Lemma 4 for $\varphi = \gamma_1'$ and the structure $\mathcal{B}_1^\infty = T(\Sigma_1, V \cup Y_1)/{=_{E_1}}$.

Part (a) of this condition is satisfied since $\mathcal{B}_1^\infty \models \gamma_1'(\nu(\boldsymbol{w}))$, and thus

$$\mathcal{B}_1^\infty \models \gamma_1'(\boldsymbol{x}_1, \boldsymbol{e}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{e}_k).$$

Part (b) of the condition is satisfied since the ν-images of all variables in $\boldsymbol{w}$ are distinct according to our choice in the variable identification step. Part (c) is satisfied due to our choice of the sets Z_j. Part (d) is satisfied since the components of $\boldsymbol{x}_j$ belong to $\boldsymbol{Y}_1^{(j)}$ and $Y_1^{(j)} \cap (Z_1 \cup \ldots \cup Z_{j-1}) = \emptyset$, the last equality following from the fact that $Y_1^{(j)}$ and $\bigcup_{i=0}^{j-1} Y_1^{(i)}$ are disjoint by definition.

Thus, we can apply Lemma 4, which yields $\mathcal{B}_1^\infty \models \alpha$. Since $\mathcal{B}_1^\infty$ and $\mathcal{B}_1$ are Σ_1-isomorphic we have $\mathcal{B}_1 \models \alpha$.

Using the fact the $h_{1-2} : \mathcal{B}_1^\infty \to \mathcal{B}_2^\infty$ is a $(\Sigma_1 \cup \Sigma_2)$-isomorphism, $\mathcal{B}_2 \models \beta$. can be shown similarly. $\square$

3.6 Generalizations

The combination method for equational unification algorithms that we have described in the previous two sections can be generalized along several orthogonal dimensions. Three such extensions will be described in this section. The first generalization concerns the syntactic form of input problems: we study the effect of adding negation to the mixed input sentences. Afterwards we introduce a class of structures that properly extends the class of free algebras, and show how to lift our combination results to this more general class of structures. In the third subsection we sketch a variant of the amalgamation construction introduced in Subsection 3.5.2, which leads to a different combined solution structure and a combination algorithm with less nondeterminism.

3.6.1 Adding Negation to Unification Problems

Compared to the Nelson-Oppen approach, the major limitation of the combination results presented in the previous two sections is that they are restricted to *positive* sentences, i.e., negated equations, so-called *disequations*, are not allowed. We shall now consider the combination of unification constraints with negation. Since the constraint solvers of constraint programming languages often have to check entailment of constraints, and since entailment uses an implicit form of negation, this extension is of great practical relevance.

As before, let E_1 and E_2 denote equational theories over disjoint signatures Σ_1 and Σ_2. When treating sentences with negation, we must first decide which form of semantics we want to use: the equivalence between validity in all models of $E_1 \cup E_2$ on the one hand, and validity in the $(E_1 \cup E_2)$-free algebra over a countably infinite set of generators V on the other hand does no longer hold if we do not restrict ourselves to positive sentences. We shall look at two alternative semantics usually considered in the literature. First, we consider validity of existential $(\Sigma_1 \cup \Sigma_2)$-sentences in the free algebra $T(\Sigma_1 \cup \Sigma_2, V)/{=_{E_1 \cup E_2}}$. Later, we consider validity in the initial algebra $T(\Sigma_1 \cup \Sigma_2, \emptyset)/{=_{E_1 \cup E_2}}$. In the first case, we talk about *solvability* and in the second about *ground solvability* of the constraints.

As long as we want to decide validity of positive existential sentences, both semantics lead to the same result as long as we assume that the joint signature contains at least one constant. This follows directly from the fact that validity of positive existential sentences is preserved under homomorphisms. For constraints with negation, the two semantics definitely lead to distinct notions of validity. The latter semantics is often preferred in the literature on constraints with negation (see, e.g., [Com91]), but the first semantics can also be found [BB94].

Since a sentence holds in a given algebra $\mathcal{A}$ if, and only if, its negation does not hold in $\mathcal{A}$, a decision procedure for validity of existential sentences in $\mathcal{A}$ immediately gives a decision procedure for validity of universal sentences in $\mathcal{A}$ and vice versa. Hence the results of this subsection concern the universal fragments of the given algebras as well.

Disunification over the free algebra. In order to describe the following results, some terminology is needed. Given an equational theory E with signature Σ, an *elementary E-disunification problem* is a finite system Γ of equations $s =^? t$ and disequations $s \neq^? t$ between Σ-terms. A substitution σ solves Γ iff $\sigma(s) =_E \sigma(t)$ for each equation $s =^? t$ in Γ and $\sigma(s) \neq_E \sigma(t)$ for each disequations $s \neq^? t$ in Γ. E-disunification problems with linear constant restrictions, and solutions for E-disunification problems with linear constant restrictions are defined as in the case of E-unification problems.

Theorem 12. *Let $E_1, \ldots, E_n$ be equational theories over pairwise disjoint signatures, and let $E := E_1 \cup \ldots \cup E_n$ denote their union. Then solvability of E-disunification problems is decidable provided that solvability of E_i-disunification problems with linear constant restrictions is decidable for $i = 1, \ldots n$.*

Since existential quantification distributes over disjunction, the theorem shows that validity of existential sentences in $T(\Sigma_1 \cup \ldots \cup \Sigma_n, V)/{=_E}$ is decidable if solvability of E_i-disunification problems with linear constant restrictions is decidable for $i = 1, \ldots n$. Unfortunately, we do not have a logical characterization of E_i-*dis*unification problems with linear constant restrictions.

A proof of this theorem can be found in [BS95b]. It is based on a combination algorithm that is a variant of the Combination Algorithm for combined E-unification problems. In principle, the only difference is that, for each pair (x, y) of variables in the input problem that is not identified at the variable identification step, we add a disequation $x \neq y$ to both output systems. For details we refer to [BS95b].

Disunification over the initial algebra. The solution σ of the E-disunification problem Γ is a *ground solution* iff $\sigma(x)$ is a ground term (i.e., does not contain variables) for all variables x occurring in Γ.

In view of Theorem 12, an obvious conjecture could be that ground solvability of a disunification problem Γ in the combined theory E can be decided by decomposing Γ into a finite set of pairs of E_i-disunification problems with linear constant restrictions, and then asking for ground solvability of the subproblems. However, in [BS95b] an example is given that shows that this method is only sound, but not complete (see Example 4.2, p. 243). The proper adaption of Theorem 13 to the case of ground solvability needs another notation: a solution σ of an E-disunification problem with linear constant restrictions, $(\Gamma, X, C, <)$, is called *restrictive* if, under σ, all variables $x \in X$ are mapped to terms $\sigma(x)$ that are not E-equivalent to a variable.

Theorem 13. *Let $E_1, \ldots, E_n$ be equational theories over pairwise disjoint signatures $\Sigma_1, \ldots, \Sigma_n$, and let $E := E_1 \cup \ldots \cup E_n$ denote the combined theory. Assume that the initial algebras $T(\Sigma_i, \emptyset)/{=_{E_i}}$ are infinite for $i = 1, \ldots, n$. Then ground solvability of E-disunification problems is decidable provided that restrictive solvability of E_i-disunification problems with linear constant restrictions is decidable for $i = 1, \ldots n$.*

A proof of this theorem, as well as of some variants that relax the condition that all the initial algebras must be infinite, can be found in [BS95b]. These techniques yield the following result.

Corollary 1. *Solvability of disunification problems is decidable for every equational theory that is a disjoint combination of finitely many theories E_f expressing associativity, associativity-commutativity, or associativity-commutativity-idempotence of some binary function symbol, together with a free theory F. If the free theory F contains at least one constants and one function symbol of arity $n \geq 1$, then ground solvability of disunification problems over the combined theory is decidable as well.*

3.6.2 More General Solution Structures

Except for the initial description of the Nelson-Oppen procedure, our discussion has been restricted to constraints that are composed of equations and disequations, and the only solution domains that we considered so far were free algebras. Obviously, a much broader variety of constraints and solution domains are relevant for the general field of constraint programming. In this subsection we first introduce a class of structures that properly extends the class of free algebras. The class contains many non-free algebras and relational structures that are of interest for constraint solving. Then we discuss the problem of combining solution domains within the given class. Finally, we show how the combination results that we obtained for free algebras can be lifted to this more general situation.

Quasi-free structures. The motivation for introducing the class of quasi-free structures is the observation that most of the non-numerical and non-finite solution domains that are used in different areas of constraint programming can be treated within a common algebraic background when we generalize the concept of a free algebra appropriately.

In a first step, one can go from free algebras to free structures where, in addition to function symbols, the signature may also contain predicate symbols. Free structures are defined Mal'cev [Mal71] analogously to free algebras: a Σ-structure $\mathcal{A}$ is called free over X in the class of Σ-structures $\mathcal{K}$ if $\mathcal{A} \in \mathcal{K}$ is generated by X, and if every mapping from X into the domain of a structure $\mathcal{B} \in \mathcal{K}$ can be extended to a Σ-homomorphism of $\mathcal{A}$ into $\mathcal{B}$. Mal'cev shows that free structures have properties that are very similar to the properties of free algebras. This fact was used in [BS95a] to extend the combination results for unification constraints to more general constraints over free solution structures.

The following lemma (see [BS98], Theorem 3.4) yields an internal characterization of structures that are free in some class of over a countably infinite set of generators. It will be the basis for our generalization from free structures to quasi-free structures.

Lemma 6. *A Σ-structure $\mathcal{A}$ is free (in some class of Σ-structures) over $X \subseteq A$ iff*

1. *$\mathcal{A}$ is generated by X,*
2. *for every finite subset X_0 of X, every mapping from X_0 to A can be extended to a surjective endomorphism of $\mathcal{A}$.*

We will now generalize the first condition in order to arrive at the concept of a quasi-free structure. Since some of the following notions are quite abstract, the algebra of rational trees will be used to exemplify definitions. In the sequel, we consider a fixed Σ-structure $\mathcal{A}$ with domain A. With $End_{\mathcal{A}}^{\Sigma}$ we denote the monoid of Σ-endomorphisms of $\mathcal{A}$. It should be stressed that in the sequel the signature Σ is arbitrary in the sense that it may contain predicate symbols as well as function symbols.

Definition 3. Let A_0, A_1 be subsets of the Σ-structure $\mathcal{A}$. Then A_0 *stabilizes* A_1 iff all elements m_1 and m_2 of $End_{\mathcal{A}}^{\Sigma}$ that coincide on A_0 also coincide on A_1. For $A_0 \subseteq A$ the *stable hull* of A_0 is the set

$$SH^{\mathcal{A}}(A_0) := \{a \in A \mid A_0 \text{ stabilizes } \{a\}\}.$$

The stable hull of a set A_0 has properties that are similar to those of the subalgebra generated by A_0: $SH^{\mathcal{A}}(A_0)$ is always a Σ-substructure of $\mathcal{A}$, and $A_0 \subseteq SH^{\mathcal{A}}(A_0)$. In general, however, the stable hull can be larger than the generated substructure. For example, if $\mathcal{A} := \mathcal{R}(\Sigma, X)$ denotes the algebra of rational trees over signature Σ and with variables in X, and if $Y \subseteq X$ is a subset of the set of variables X, then $SH^{\mathcal{A}}(Y)$ consists of all *rational* trees with variables in Y, while Y generates all *finite* trees with variables in Y only.

Definition 4. The set $X \subseteq A$ is an atom set for $\mathcal{A}$ if every mapping $X \to A$ can be extended to an endomorphism of $\mathcal{A}$.

For example, if $\mathcal{A} := \mathcal{R}(\Sigma, X)$ is the algebra of rational trees with variables in X, then X is an atom set for $\mathcal{A}$.

Definition 5. A countably infinite Σ-structure $\mathcal{A}$ is *quasi-free* iff $\mathcal{A}$ has an infinite atom set X where every $a \in A$ is stabilized by a finite subset of X.

The definition generalizes the characterization of free algebras in Lemma 6. The countably infinite set of generators is replaced by the atom set, but we retain some properties of generators. In the free case, every element of the algebra is generated by a finite set of generators, whereas in the quasi-free case it is stabilized by a finite set of atoms. It can be shown easily that the second condition of Lemma 6 holds in the quasi-free case as well.

Examples 14 *Each free algebra and each free structure is quasi-free. Examples of non-free quasi-free structures are rational tree algebras; nested, hereditarily finite non-wellfounded sets, multisets, and lists; as well as various types of feature structures. In each case we have to assume the presence of a countably infinite set of atoms (variables, urelements, etc.). For the exact definitions of these example structures we refer to [BS98].*

Free amalgamation of quasi-free structures. When combining constraint systems for quasi-free structures, the question arises how to define the combined solution structure. It turns out that the amalgamation construction that we have described in Subsection 3.5.2 can be generalized from free algebras to quasi-free structures. The result of this construction is a quasi-free structure over the combined signature. In the modified construction, the atom sets of the two quasi-free component structures play the rôle of the variable sets. The intermediate substructures that occur during the fibering process are now defined as the stable hulls of the atom sets considered at the steps of the construction.

In the case of free algebras, the use of the amalgamation construction was justified by the fact that it yielded exactly the combined algebra we were looking for, i.e., the free algebra for the combined theory. In the case of quasi-free structures, we do not have an equational theory defining the component structures. Thus, the question arises whether the amalgamation construction really yields a "sensible" combined solution structure. This question has been answered affirmatively in [BS98].

In fact, the resulting combined structure has a unique and privileged status. In [BS98] we have introduced the notion of an admissible combination of two structures. The *free amalgamated product* $\mathcal{A}_1 \oplus \mathcal{A}_2$ of two structures is the most general admissible combination of $\mathcal{A}_1$ and $\mathcal{A}_2$ in the sense that every other admissible combination $\mathcal{C}$ is a homomorphic image of $\mathcal{A}_1 \oplus \mathcal{A}_2$ (see [BS98] for an exact definition). It can be shown that the free amalgamated product of two quasi-free structures over disjoint signatures always exists since it coincides with the structure produced by our amalgamation construction (i.e., the extension to quasi-free structures of the construction presented above for the case of free algebras).

Solving mixed constraints in the free amalgamated product. When using the free amalgamated product of two given quasi-free structures $\mathcal{A}_1$ and

$\mathcal{A}_2$ as the solution domain for mixed constraints, a simple adaption of the logical reformulation of the Combination Algorithm can be used to reduce solvability of positive existential formulae in $\mathcal{A}_1 \oplus \mathcal{A}_2$ to solvability of positive sentences in the component structures $\mathcal{A}_1$ and $\mathcal{A}_2$. The only difference comes from the fact that we now have a new type of atomic formulae in the input problems, namely, atomic formulae that are built with predicate symbols in the signature. It is, however, straightforward to show that, given an existential sentence $\exists \boldsymbol{u}.\, \gamma$ over the mixed signature $\Sigma_1 \cup \Sigma_2$, it is possible to compute an equivalent existential sentence of the form $\exists \boldsymbol{v}.\, (\gamma_1 \wedge \gamma_2)$ where the conjunction of atomic formulae γ_i is built using symbols from Σ_i only; in fact, the variable abstraction step introduced in Section 3.3 also treats non-equational atoms. The remaining steps of the logical version of the Combination Algorithm can be used without any changes.

The correctness of the modified Combination Algorithm, which is proved in [BS98], yields the following result.

Theorem 15. *Let $\mathcal{A}_1, \dots, \mathcal{A}_n$ be quasi-free structures over disjoint signatures $\Sigma_1, \dots, \Sigma_n$, and let Σ denote the union of these signatures. Then validity of positive existential Σ-sentences in the free amalgamated product $\mathcal{A}_1 \oplus \cdots \oplus \mathcal{A}_n$ is decidable provided that validity of positive Σ_i-sentences in the component structures is decidable.*

As in the case of free algebras, it is possible to lift this result to general positive input sentences.

Theorem 16. *Let $\mathcal{A}_1, \dots, \mathcal{A}_n$ be quasi-free structures over disjoint signatures $\Sigma_1, \dots, \Sigma_n$, and let Σ denote the union of these signatures. Then validity of positive Σ-sentences in the free amalgamated product $\mathcal{A}_1 \oplus \cdots \oplus \mathcal{A}_n$ is decidable provided that validity of positive Σ_i-sentences in the component structures is decidable.*

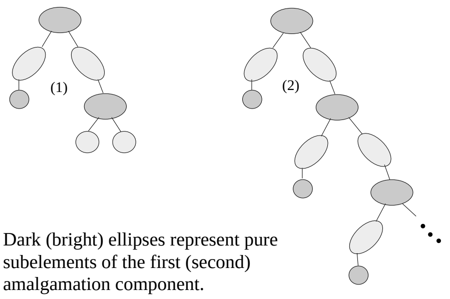
For the following quasi-free structures, the positive theories turn out to be decidable (cf. Ex. 14): non-ground rational feature structures with arity; finite or rational tree algebras; nested, hereditarily finite wellfounded or non-wellfounded sets; and nested, hereditarily finite wellfounded or non-wellfounded lists. Hence, provided that the signatures are disjoint, the free amalgamated product of any finite number of these structures has a decidable positive theory.

It is also possible to extend the results combination results for disunification to the case of quasi-free structures. This yields decidability results for the existential (or universal) theory of the free amalgamated product of a great variety of structures, such as feature structures, nested lists, sets and multisets, rational tree algebras and others. We refer to Kepser [Kep99, Kep98] for details.

3.6.3 Other Amalgamation Techniques

In Schulz and Kepser [KS96] a second systematic way of combining constraint systems over quasi-free structures, called *rational amalgamation*, has been introduced. Like the free amalgamated product, rational amalgamation yields a combined structure with "mixed" elements that inter-weave a finite number of "pure" elements of the two components in a particular way. The difference between both constructions becomes transparent when we ignore the interior structure of these pure subelements and consider them as construction units with a fixed arity, similar to "complex function symbols." Under this perspective, and ignoring details that concern the ordering of the children of a node, mixed elements of the free amalgamated product can be considered as finite trees, whereas mixed elements of the rational amalgam are like rational trees.[13]

Mixed element of free amalgam (1) and of rational amalgam (2).


(1)
(2)
Dark (bright) ellipses represent pure subelements of the first (second) amalgamation component.

With this background, it should not be surprising that in praxis rational amalgamation appears to be the preferred combination principle in the literature in situations where the two solution structures to be combined are themselves "rational" or "cyclic" domains: for example, it represents the way how rational trees and rational lists are interwoven in the solution domain of Prolog III [Col90], and a variant of rational amalgamation has been used to combine feature structures with non-wellfounded sets in a system introduced by Rounds [Rou88].

Rational amalgamation can be used to combine so-called non-collapsing quasi-free structures over disjoint signatures.

Definition 6. An quasi-free structure $\mathcal{A}$ with atom set X is *non-collapsing* if every endomorphism of $\mathcal{A}$ maps non-atoms to non-atoms (i.e., $m(a) \in A\setminus X$ for all $a \in A \setminus X$ and all endomorphisms m of $\mathcal{A}$).

For example, quotient term algebras for collapse-free equational theories, rational tree algebras, feature structures, feature structures with arity, the do-

[13] A (possibly infinite) tree is *rational* if it is finitely branching and has only a finite number of distinct subtrees; see [Col84, Mah88, Cou83].

mains with nested, finite or rational lists, and the domains with nested, finite or rational multi-sets are always non-collapsing.

The amalgamation construction for rational amalgamation is rather technical and thus beyond the scope of this paper; we refer to [KS96] for details. Just as in the case of free amalgamation, constraint solvers for two component structures can be combined to a constraint solver for their rational amalgam. To be more precise, validity of positive existential sentences in the rational amalgam can be reduced to solvability of conjunctions of atomic constraints with so-called atom/non-atom declarations in the component structures (see [KS96] for a formal definition of this notion). From the algorithmic point of view, rational amalgamation appears to be interesting since the combination technique for rational amalgamation avoids one source of nondeterminism that is needed in the corresponding scheme for free amalgamation: the choice of a linear ordering, which is indispensable for free amalgamation, must be omitted in the case of rational amalgamation.

One interesting connection between free and rational amalgamation is the observation that the free amalgamated product is always a substructure of the rational amalgamated product (see [Kep98]).

3.7 Optimization and Complexity Issues

Until now, we have described the combination method for unification algorithms from a theoretical point of view, that is, our main emphasis was on clearness of presentation and on ease of proving correctness. It should be clear, however, that a naïve implementation of the highly nondeterministic Combination Algorithm cannot be used in practice. It is easy to see that the nondeterminism of the procedure indeed represents a serious problem: the number of possible partitions of a set of n variables is known as the n-th Bell number, which grows faster than 2^n. The choice of a labeling function and a linear ordering leads to another exponential number of subcases that must be investigated. Hence, significant optimizations are necessary before one can hope for a combined unification algorithm that can be used in a realistic application.

In the following, we show how the algorithm that combines decision procedures can be optimized. (An optimized version of the combination method for algorithms that compute complete sets of unifiers can be found in [Bou93].) In general, however, there is an inherent nondeterminism in the problem of combining unification algorithms, which cannot be avoided. We will come back to this point at the end of this section.

Some simple optimizations of the Combination Algorithm are straightforward. It is possible to restrict all nondeterministic choices to "shared" variables, that is, variables that occur in at least two subproblems of the decomposed problem. Another simple optimization relies on the observation that different linear orders need not lead to different constant restrictions.

For example, assume that x, y are variables and c, d are (variables treated as) constants. Then the ordering $x < c < d < y$ leads to the same restrictions on solutions of a unification problem as the ordering $x < d < c < y$ (both just say that x must not be replaced by a term containing c or d). This observation can easily be used to prune the number of different linear orderings that must be considered.

On a more sophisticated level, Kepser and Richts [KR99a] have described two powerful orthogonal optimization methods. We describe the first method, called "deductive method," in more detail, and then briefly sketch the second one, called "iterative method," and the integration of both approaches.

The deductive method tries to reduce the amount of nondeterminism by avoiding certain branches in the search tree for which one can "easily detect" that they cannot lead to solutions. Before going into more detail, we consider an example that illustrates the basic idea.

Example 7. Assume that the component theory E_i is collapse-free and the decomposed input problem contains an equation $x =^? f(\dots)$ where $f \in \Sigma_i$. Then x must receive label i since $x \neq_{E_i} \sigma(f(\dots))$ for all substitutions σ, i.e., if x is treated as a constant in the ith subproblem, then this problem does not have a solution. Consequently, labeling functions Lab with $Lab(x) \neq i$ need not be considered.

If E_i is regular, the decomposed input problem contains an equation $x =^? t$, and $y \in Var(t)$ for a variable y with $Lab(x) \neq Lab(y)$, then there cannot be a solution σ (of the subproblem in which x is instantiated) in which y does not occur in $\sigma(x)$. Hence, we can deterministically choose the order $y < x$ between x and y, i.e., the other alternative need not be considered.

In order to formalize this idea, we introduce a constraint language that allows us to represent such mandatory choices on the way to a fully specified output pair of the Combination Algorithm. A complete set of guesses of the algorithm—with the trivial optimizations mentioned above included now—can be described in the form $(\Pi, Lab, <)$, where

- Π is a partition of the set X of shared variables of the decomposed problem $\Gamma_1 \uplus \dots \uplus \Gamma_n$ reached after the first step,
- $Lab : X \to \{1, \dots, n\}$ is a labeling function that respects equivalence classes of Π, i.e., if x and y belong to the same class, then $Lab(x) = Lab(y)$, and
- $<$ is a strict linear ordering on the equivalence classes. We write $x < y$ if the equivalence classes $[x]$ and $[y]$ of x and y are in the relation $[x] < [y]$.

In the sequel, output problems will be described as quadruples of the form $(\Gamma_i, \Pi, Lab, <)$. The corresponding E_i-unification with linear constant restrictions, $(\Gamma'_i, X_i, C_i, <)$, can be obtained from this quadruple as described in the Combination Algorithm, i.e., Γ'_i is obtained from Γ_i by replacing all shared variables by the representatives of their equivalence classes w.r.t. Π, X_i is the union of the set of shared variables with label i and the set of non-shared variables in Γ_i, and C_i is the set of shared variables with a label different from i. The quadruple $(\Gamma_i, \Pi, Lab, <)$ is said to be *solvable* iff the corresponding E_i-unification with linear constant restrictions is solvable.

Constraints are of the form $x = y$, $\neg(x = y)$, $x < y$, $\neg(x < y)$, $x : i$, or $\neg(x : i)$, with the obvious meaning that $x = y$ ($\neg(x = y)$) excludes partitions in which x and y belong to different classes (the same class), $x < y$ ($\neg(x < y)$) excludes orderings and partitions in which $y \leq x$ ($x < y$), and $x : i$ ($\neg(x : i)$) excludes labelling functions Lab such that $Lab(x) \neq i$ ($Lab(x) = i$). On the one hand, a set of constraints excludes certain triples $(\Pi, Lab, <)$. On the other hand, it can also be seen as a partial description of a triple that satisfies these constraints (i.e., is not excluded by them). A set of constraints $\mathcal{C}$ is called *complete* iff there is exactly one triple $(\Pi, Lab, <)$ that satisfies $\mathcal{C}$, and it is called *inconsistent* iff no triple satisfies $\mathcal{C}$ (i.e., it contains two contradictory constraints).

The deductive method assumes that each theory E_i is equipped with a *component algorithm* that, given a pure E_i-unification problem Γ_i together with a set of constraints $\mathcal{C}$, deduces a (possibly empty) set of additional constraints $\mathcal{D}$. This algorithm is required to be correct in the following sense: if $(\Pi, Lab, <)$ is a triple that satisfies $\mathcal{C}$ and for which $(\Gamma_i, \Pi, Lab, <)$ is solvable, then $(\Pi, Lab, <)$ also satisfies $\mathcal{D}$.

Given a system $\Gamma_1 \uplus \cdots \uplus \Gamma_n$ in decomposed form, the search for an appropriate triple $(\Pi, Lab, <)$ is now performed by the nondeterministic algorithm of Fig. 3.4.

```
Initialize $\mathcal{C} := \emptyset$;
Repeat
   Repeat
      For each system $i$
         (* **Deduce new constraints** *)
         call the component algorithm of theory $E_i$ to calculate
         new consequences $\mathcal{D}$ of $\Gamma_i$ and $\mathcal{C}$;
         set the current set of constraints to $\mathcal{C} := \mathcal{C} \cup \mathcal{D}$
   Until $\mathcal{C}$ is inconsistent or no more new constraints are computed;
   If $\mathcal{C}$ is consistent and not complete
      (* **Select next choice** *)
      Select a constraint $c$ such that $\{c, \neg c\} \cap \mathcal{C} = \emptyset$;
      Non-deterministically choose either
      $\mathcal{C} := \mathcal{C} \cup \{c\}$ or
      $\mathcal{C} := \mathcal{C} \cup \{\neg c\}$
Until $\mathcal{C}$ is inconsistent or complete;
Return $\mathcal{C}$
```

Fig. 3.4. The deductive method.

Proposition 5. *Let $\varGamma := \varGamma_1 \uplus \cdots \uplus \varGamma_n$ be an (elementary) $(E_1 \cup \cdots \cup E_n)$-unification problem in decomposed form where the equational theories E_i have pairwise disjoint signatures. Then the following statements are equivalent:*

1. *$\varGamma$ is solvable, i.e., there exists an $(E_1 \cup \cdots \cup E_n)$-unifier of $\varGamma$.*
2. *One of the complete constraint sets generated by the nondeterministic algorithm of Fig. 3.4 describes a triple $(\varPi, \mathrm{Lab}, <)$ such that, for all $i = 1, \ldots, n$, $(\varGamma_i, \varPi, \mathrm{Lab}, <)$ is solvable.*

One should note that the trivial component algorithm that always returns the empty set of constraints is correct. If all component algorithms are trivial, then the algorithm of Fig. 3.4 simply generates all possible triples $(\varPi, Lab, <)$.

We have already illustrated by an example that the fact that a theory is regular and/or collapse-free can be used to derive new constraints. For a free theory E_i, the most general unifier of $\varGamma_i$ (which can be computed in linear time) can be used to read off new constraints. The following example shows how information provided by one component algorithm can help another component algorithm in deriving additional constraints. This explains why the step of deducing new constraints must be iterated.

Example 8. Consider the following mixed input problem $\{f(g(x_4), x_2) =^? f(g(y), x_4),\ x_4 =^? f(a, a)\}$, where f, a belong to the regular, collapse-free theory E_1 (e.g., AC_f) and g belongs to the free theory E_2. By decomposition, the E_1-subsystem $\{f(x_1, x_2) =^? f(x_3, x_4),\ x_4 =^? f(a, a)\}$ and the E_2-subsystem $\{x_1 =^? g(x_4),\ x_3 =^? g(y)\}$ are created. Since E_1 is collapse-free, the equation $x_4 =^? f(a, a)$ can be used by the first component algorithm to deduce the constraint $x_4 : 1$. From the most general unifier $\{x_1 \mapsto g(x_4),\ x_3 \mapsto g(y)\}$ of the E_2-subsystem, the second component algorithm can derive the constraints $x_1 : 2, x_3 : 2$ and $x_4 < x_1$. Given the regularity of E_1, the first component algorithm can now derive $x_1 = x_3$. In fact, x_1 (which must be treated as a constant in the E_1-subsystem) occurs on the left-hand side of $f(x_1, x_2) =^? f(x_3, x_4)$, and thus must occur on the (instantiated) right-hand side $f(\sigma(x_3), \sigma(x_4))$ for any solution σ of the E_1-subsystem. Since we already have the constraints $x_4 : 1$, $x_4 < x_1$, and $x_3 : 2$, we know that $\sigma(x_3) = x_3$ and x_1 cannot occur in $\sigma(x_4)$. Consequently, x_1 can only occur in $f(\sigma(x_3), \sigma(x_4)) = f(x_3, \sigma(x_4))$ if x_1 and x_3 are identified.

Obviously, the quality of the component algorithms used in the deductive method decides the amount of optimization achieved. The goal is to deduce as much information as is possible with a reasonable effort. Detailed descriptions of component algorithms for the free theory, the theory AC of an associative-commutative function symbol, and the theory ACI of an associative-commutative-idempotent function symbol can be found in [Ric99].

While the deductive method helps to reach certain decisions deterministically, the "iterative method" introduced in [KR99a,Kep98]—which is relevant if $n \geq 3$ theories are combined—determines in which order the nondeterministic decisions should best be made. Basically, the output systems are solved iteratively, one system at a time. All decisions in nondeterministic steps are made locally, for the current system i only. This means, for example, that we only consider variables occurring in the system Γ_i, and for such a variable we just decide if it receives label i or not. In the latter case, the exact label $j \neq i$ is not specified. Once all decisions relevant to system Γ_i have been made, it is immediately tested for solvability. If Γ_i turns out to be unsolvable, we thus have avoided finding this out as many times as there are possible choices for the constraints not relevant to the subsystem Γ_i.

An integration of the deductive and the iterative method is achieved by plugging the iterative selection strategy into the deductive algorithm. To be more precise, whenever the deductive algorithm (see Fig. 3.4) needs to make a nondeterministic choice (since no more constraints can be deduced), the selection strategy of the iterative method decides for which constraint this choice is made. This synthesis of both optimization techniques has been implemented, and run time tests show that the optimized combination method obtained this way leads to combined decision procedures that have a quite reasonable practical time complexity [KR99a, KR99b].

Fundamental limitations for optimization. Complexity theoretical considerations in [Sch00a] show that, in many cases, there are clear limitations for optimizing the Combination Algorithm. We close this section with some results concerning the situation where an equational theory E is combined with a free theory in order to obtain an algorithm for general E-unification.

Definition 7. A *polynomial-time optimization* of the Combination Algorithm for general E-unification is an algorithm that accepts as input an arbitrary general E-unification problem Γ and computes in polynomial time a finite set M of output pairs $((\Gamma_1, \Pi, X_1, X_2, <), (\Gamma_2, \Pi, X_2, X_1, <))$ of an E-unification problems with linear constant restrictions and a free unification problems with linear constant restrictions such that

- each output pair in M is also a possible output pair of the original Combination Algorithm, and
- Γ is solvable iff, for some output pair in M, both components are solvable.

On the one hand, Schulz [Sch00a] characterizes a large class of equational theories E where a polynomial optimization of the Combination Algorithm for general E-unification is impossible unless $P = NP$. In order to formulate one result that follows from this characterization, we need the following notation: a binary function symbol "f" is called a commutative (resp. associative) function symbol of the equational theory E if f belongs to the signature of E and $f(x, y) =_E f(y, x)$ (resp. $f(x, f(y, z)) =_E f(f(x, y), z)$).

Theorem 17. *Let E be an equational theory that contains an associative or commutative function symbol. If E is regular, then there exists no polynomial-time optimization of the Combination Algorithm for general E-unification, unless $P = NP$.*

In [Sch00b] it is shown that such impossibility results for polynomial optimization of combination algorithms are by no means specific to the problem of combining E-unification algorithms. The paper presents a general framework that characterizes situations in which combination algorithms for decision procedures cannot be polynomially optimized. In particular, various combinations of first-order theories are characterized where the non-deterministic variant of the Nelson-Oppen procedure does not have a polynomial optimization.

On the other hand, Schulz [Sch00a] also introduces a class of equational theories for which a polynomial-time optimization of the Combination Algorithm is always possible. Basically, these are regular and collapse-free theories of unification type unitary (i.e., all solvable unification problems have a most general unifier) such that "enough" information about the most general unifier can be computed in polynomial time.

3.8 Open Problems

The results described in this paper show that the problem of combining constraint solvers over disjoint signatures is well-investigated, at least if one considers as constraint solvers procedures that decide satisfiability of constraints.

As mentioned in Section 3.2.1, it is often desirable to have constraint solvers that are able to compute solved forms in an incremental way. To the best of our knowledge, there are no *general* results on how to combine such incremental constraint solvers. A general solution to this problem depends on a general and abstract definition of the concept of a solved form that covers most of the relevant instances.

Another challenging field for future research is the problem of combining constraint solvers over non-disjoint signatures. Since non-disjoint combinations may lead to undecidability, the main point is to find appropriate restrictions on the constraint languages to be combined. For the kind of combination problems considered by Nelson-Oppen, first combination results for the non-disjoint case have been obtained by Ch. Ringeissen and C. Tinelli [Rin96, Tin99, TR98]. Similarly, the known combination methods for solving the word problem in the union of equational theories have been lifted to the case of non-disjoint signatures in [DKR94,BT97,BT99,BT00]. Concerning the combination of unification algorithms for equational theories over non-disjoint signatures, first results have been presented in [DKR94]. Using the more abstract algebraic concepts that have been developed during the last years it should be possible to simplify and then generalize this work, which only addresses the combination of algorithms for computing complete sets of unifiers.

Acknowledgements. The authors should like to thank the anonymous referee for his comments, which helped to improve the presentation of this paper. This work was partially supported by the ESPRIT Working Group CCL and by the DFG SPP "Deduktion".

References

[Baa98] F. Baader. On the complexity of Boolean unification. *Information Processing Letters*, 67(4):215–220, 1998.

[Bac91] L. Bachmair. *Canonical Equational Proofs.* Birkhäuser, Boston, Basel, Berlin, 1991.

[BB94] W.L. Buntine and H.-J. Bürckert. On solving equations and disequations. *Journal of the ACM*, 41(4):591–629, 1994.

[BJSS89] A. Boudet, J.-P. Jouannaud, and M. Schmidt-Schauß. Unification in Boolean rings and Abelian groups. *J. Symbolic Computation*, 8:449–477, 1989.

[Boc92] A. Bockmayr. Algebraic and logical aspects of unification. In K.U. Schulz, editor, *Proceedings of the 1st International Workshop on Word Equations and Related Topics (IWWERT '90)*, volume 572 of *Lecture Notes in Computer Science*, pages 171–180, Tübingen, Germany, October 1992. Springer-Verlag.

[Bou93] A. Boudet. Combining unification algorithms. *Journal of Symbolic Computation*, 8:449–477, 1993.

[BS92] F. Baader and K.U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In D. Kapur, editor, *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 50–65, Saratoga Springs, NY, USA, 1992. Springer-Verlag.

[BS93] F. Baader and K.U. Schulz. General A- and AX-unification via optimized combination procedures. In *Proceedings of the Second International Workshop on Word Equations and Related Topics*, volume 677 of *Lecture Notes in Computer Science*, pages 23–42, Rouen, France, 1993. Springer-Verlag.

[BS94] F. Baader and J.H. Siekmann. Unification theory. In D.M. Gabbay, C.J. Hogger, and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, pages 41–125. Oxford University Press, Oxford, UK, 1994.

[BS95a] F. Baader and K.U. Schulz. Combination of constraint solving techniques: An algebraic point of view. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Artificial Intelligence*, pages 352–366, Kaiserslautern, Germany, 1995. Springer-Verlag.

[BS95b] F. Baader and K.U. Schulz. Combination techniques and decision problems for disunification. *Theoretical Computer Science B*, 142:229–255, 1995.

[BS96] F. Baader and K.U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. *J. Symbolic Computation*, 21:211–243, 1996.

[BS98] F. Baader and K.U. Schulz. Combination of constraint solvers for free and quasi-free structures. *Theoretical Computer Science*, 192:107–161, 1998.

[BS00] F. Baader and W. Snyder. Unification theory. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning.* Elsevier Science Publishers, 2000. To appear.

[BT97] F. Baader and C. Tinelli. A new approach for combining decision procedures for the word problem, and its connection to the Nelson-Oppen combination method. In W. McCune, editor, *Proceedings of the 14th International Conference on Automated Deduction (Townsville, Australia)*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 19–33. Springer-Verlag, 1997.

[BT99] F. Baader and C. Tinelli. Deciding the word problem in the union of equational theories sharing constructors. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA-99)*, volume 1631 of *Lecture Notes in Computer Science*, pages 175–189. Springer-Verlag, 1999.

[BT00] F. Baader and C. Tinelli. Combining equational theories sharing non-collapse-free constructors. In H. Kirchner and Ch. Ringeissen, editors, *Proceedings of the 3rd International Workshop on Frontiers of Combining Systems (FroCoS 2000)*, volume 1794 of *Lecture Notes in Artificial Intelligence*, pages 257–271, Nancy, France, 2000. Springer-Verlag.

[Bür91] H.-J. Bürckert. *A Resolution Principle for a Logic with Restricted Quantifiers*, volume 568 of *Lecture Notes in Artificial Intelligence.* Springer-Verlag, 1991.

[CH96] J. Calmet and K. Homann. Classification of communication and cooperation mechanisms for logical and symbolic computation systems. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop, Munich (Germany)*, Applied Logic, pages 221–234. Kluwer Academic Publishers, March 1996.

[CK90] C.C. Chang and H.J. Keisler. *Model Theory*, volume 73 of *Studies in Logic and the Foundations of Mathematics.* Elsevier, Amsterdam, 3rd edition, 1990. (1st ed., 1973; 2nd ed., 1977).

[CLS96] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In M.A. McRobbie and J.K. Slaney, editors, *Proceedings of the 13th International Conference on Automated Deduction, (New Brunswick, NJ)*, volume 1104 of *Lecture Notes in Artificial Intelligence*, pages 463–477. Springer-Verlag, 1996.

[Col84] A. Colmerauer. Equations and inequations on finite and infinite trees. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 85–99, Tokyo, Japan, 1984. North Holland.

[Col90] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[Com91] H. Comon. Disunification: A survey. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 322–359. MIT Press, Cambridge, MA, 1991.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, 1983.

[DKR94] E. Domenjoud, F. Klay, and C. Ringeissen. Combination techniques for non-disjoint equational theories. In A. Bundy, editor, *Proceedings of the*

12th International Conference on Automated Deduction, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 267–281, Nancy, France, 1994. Springer-Verlag.

[Fag84] F. Fages. Associative-commutative unification. In R. E. Shostak, editor, *Proceedings of the 7th International Conference on Automated Deduction*, volume 170 of *Lecture Notes in Computer Science*, pages 194–208, Napa, USA, 1984. Springer-Verlag.

[Hem94] E. Hemaspaandra. Complexity transfer for modal logic. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science (LICS '94)*, pages 164–175, Paris, France, 1994. IEEE Computer Society Press.

[Her86] A. Herold. Combination of unification algorithms. In J.H. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 450–469, Oxford, UK, 1986. Springer-Verlag.

[HS87] A. Herold and J.H. Siekmann. Unification in Abelian semigroups. *J. Automated Reasoning*, 3:247–283, 1987.

[JK86] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM J. Computing*, 15(4):1155–1194, 1986.

[JK91] J.-P. Jouannaud and C. Kirchner. Solving equations in abstract algebras: A rule-based survey of unification. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of A. Robinson.* MIT Press, Cambridge, MA, 1991.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages*, pages 111–119, Munich, Germany, 1987.

[JLM84] J. Jaffar, J.-L. Lassez, and M. Maher. A theory of complete logic programs with equality. *J. Logic Programming*, 1, 1984.

[Kep98] S. Kepser. *Combination of Constraint Systems.* Phd thesis, CIS–Universität München, München, Germany, 1998. Also available as CIS-Report 98-118.

[Kep99] S. Kepser. Negation in combining constraint systems. In Dov Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2, Papers presented at FroCoS'98*, pages 117–192, Amsterdam, 1999. Research Studies Press/Wiley.

[Kir85] C. Kirchner. *Méthodes et Outils de Conception Systématique d'Algorithmes d'Unification dans les Théories Equationelles.* Thèse d'État, Université de Nancy I, France, 1985.

[KK89] C. Kirchner and H. Kirchner. Constrained equational reasoning. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation*, Portland, Oregon, 1989. ACM Press.

[KR94] H. Kirchner and Ch. Ringeissen. Combining symbolic constraint solvers on algebraic domains. *Journal of Symbolic Computation*, 18(2):113–155, 1994.

[KR99a] S. Kepser and J. Richts. Optimisation techniques for combining constraint solvers. In Dov Gabbay and Maarten de Rijke, editors, *Frontiers of Combining Systems 2, Papers presented at FroCoS'98*, pages 193–210, Amsterdam, 1999. Research Studies Press/Wiley.

[KR99b] S. Kepser and J. Richts. Unimok – a system for combining equational unification algorithms. In P. Narendran and M. Rusinowitch, editors, *Proceedings of the 10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, pages 248–251, Trento, Italy, 1999. Springer-Verlag.

[KS96] S. Kepser and K. U. Schulz. Combination of constraint systems II: Rational amalgamation. In E.C. Freuder, editor, *Principles and Practice of Constraint Programming - CP96*, volume 1118 of *LNCS*, pages 282–296. Springer, 1996. Long version to appear in Theoretical Computer Science.

[KW91] M. Kracht and F. Wolter. Properties of independently axiomatizable bimodal logics. *The Journal of Symbolic Logic*, 56(4):1469–1485, December 1991.

[LB96] C. Landauer and K.L. Bellman. Integration systems and interaction spaces. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop, Munich (Germany)*, Applied Logic, pages 249–266. Kluwer Academic Publishers, March 1996.

[Mah88] M.J. Maher. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings of the Third Annual IEEE Symposium on Logic in Computer Science*, pages 348–357, Edinburgh, Scotland, 1988. IEEE Computer Society.

[Mak77] G.S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik*, 103:147–236, 1977. English translation in Math. USSR Sbornik 32, 1977.

[Mal71] A.I. Mal'cev. *The Metamathematics of Algebraic Systems*, volume 66 of *Studies in Logic and the Foundation of Mathematics*. North Holland, Amsterdam, 1971.

[Mal73] A.I. Mal'cev. *Algebraic Systems*, volume 192 of *Die Grundlehren der mathematischen Wissenschaften in Einzeldarstellungen*. Springer-Verlag, Berlin, 1973.

[NO79] G. Nelson and D.C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NO80] G. Nelson and D.C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[NR94] R. Nieuwenhuis and A. Rubio. AC-superposition with constraints: No AC-unifiers needed. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 545–559, Nancy, France, 1994. Springer-Verlag.

[Ohl94] E. Ohlebusch. On the modularity of termination of term rewriting systems. *Theoretical Computer Science*, 136:333–360, 1994.

[Ohl95] E. Ohlebusch. Modular properties of composable term rewriting systems. *Journal of Symbolic Computation*, 20(1):1–41, 1995.

[Opp80] D.C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12:291–302, 1980.

[Plo72] G. Plotkin. Building in equational theories. *Machine Intelligence*, 7:73–90, 1972.

[Ric99] J. Richts. *Effiziente Entscheidungsverfahren zur E-Unifikation*. Dissertation, RWTH Aachen, Germany, 1999. Published by Shaker Verlag Aachen, *Berichte aus der Informatik*, 2000.

[Rin92] C. Ringeissen. Unification in a combination of equational theories with shared constants and its application to primal algebras. In A. Voronkov, editor, *Proceedings of the Conference on Logic Programming and Automated Reasoning*, Lecture Notes in Artificial Intelligence, pages 261–272, St. Petersburg, Russia, 1992. Springer-Verlag.

[Rin96] Ch. Ringeissen. Cooperation of decision procedures for the satisfiability problem. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop, Munich (Germany)*, Applied Logic, pages 121–140. Kluwer, March 1996.

[Rou88] W.C. Rounds. Set values for unification-based grammar formalisms and logic programming. Research Report CSLI-88-129, CSLI, Stanford, 1988.

[Sch92] K.U. Schulz. Makanin's algorithm for word equations: Two improvements and a generalization. In K.U. Schulz, editor, *Proceedings of the 1st International Workshop on Word Equations and Related Topics (IWWERT '90)*, volume 572 of *Lecture Notes in Computer Science*, pages 85–150, Berlin, Germany, October 1992. Springer-Verlag.

[Sch00a] K.U. Schulz. Tractable and intractable instances of combination problems for unification and disunification. *J. Logic and Computation*, 10(1):105–135, 2000.

[Sch00b] K.U. Schulz. Why combined decision procedures are often intractable. In H. Kirchner and Ch. Ringeissen, editors, *Proceedings of the 3rd International Workshop on Frontiers of Combining Systems (FroCoS 2000)*, volume 1794 of *Lecture Notes in Artificial Intelligence*, pages 217–244, Nancy, France, 2000. Springer-Verlag.

[Sho84] R.E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31:1–12, 1984.

[SS89a] M. Schmidt-Schauß. Unification in a combination of arbitrary disjoint equational theories. *J. Symbolic Computation*, 8(1,2):51–99, 1989.

[SS89b] J.H. Siekmann and P. Szabó. The undecidability of the D_A-unification problem. *J. Symbolic Computation*, 54(2):402–414, 1989.

[Sti75] M. E. Stickel. A complete unification algorithm for associative-commutative functions. In *Proceedings of the 4th International Joint Conference on Artificial Intelligence*, pages 71–82, Tblisi, USSR, 1975.

[Sti81] M.E. Stickel. A unification algorithm for associative commutative functions. *J. of the ACM*, 28(3):423–434, 1981.

[Sti85] M.E. Stickel. Automated deduction by theory resolution. *J. Automated Reasoning*, 1(4):333–355, 1985.

[TA87] E. Tidén and S. Arnborg. Unification problems with one-sided distributivity. *J. Symbolic Computation*, 3(1–2):183–202, 1987.

[TH96] C. Tinelli and M. Harandi. A new correctness proof of the Nelson-Oppen combination procedure. In F. Baader and K.U. Schulz, editors, *Frontiers of Combining Systems: Proceedings of the 1st International Workshop, Munich (Germany)*, Applied Logic, pages 103–120. Kluwer, March 1996.

[Tid86] E. Tidén. Unification in combinations of collapse-free theories with disjoint sets of function symbols. In J.H. Siekmann, editor, *Proceedings of the 8th International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 431–449, Oxford, UK, 1986. Springer-Verlag.

[Tin99] C. Tinelli. *Combining Satisfiability Procedures for Automated Deduction and Constraint-based Reasoning.* Phd thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, Illinois, 1999.

[Toy87] Y. Toyama. On the Church-Rosser property for the direct sum of term rewriting systems. *J. ACM*, 34:128–143, 1987.

[TR98] C. Tinelli and Ch. Ringeissen. Non-disjoint unions of theories and combinations of satisfiability procedures: First results. Technical Report UIUCDCS-R-98-2044, Department of Computer Science, University of Illinois at Urbana-Champaign, April 1998. (also available as INRIA research report no. RR-3402).

[Yel87] K. Yelick. Unification in combinations of collapse-free regular theories. *J. Symbolic Computation*, 3(1,2):153–182, 1987.

4 Constraints and Theorem Proving

Harald Ganzinger[1] and Robert Nieuwenhuis[2]*

[1] Max-Planck-Institut für Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany.
[2] Technical University of Catalonia, Jordi Girona 1, 08034 Barcelona, Spain.

4.1 Introduction

Symbolic constraints provide a convenient way of describing the connection between deduction at the level of ground formulae and the general inferences as they are computed by a prover on formulae with variables.

For instance, each resolution inference represents infinitely many ground inferences, obtained by substituting ground terms for the variables in the clauses. For refutational completeness only certain restricted cases of ground inferences are required. In the case of ordered resolution, essential ground inferences only resolve maximal literals. On the non-ground level, these ordering restrictions can be represented as a kind of restricted quantification: only ground terms that satisfy the ordering restrictions should be substituted for the variables. This leads us to resolution calculi with order-constrained clauses, where in each inference the conclusion is restricted by the ordering constraints of the current deduction step, combined with the constraints that are already present in the premises. Clauses with unsatisfiable constraints represent no ground clauses and can hence be removed during the deduction process, thus pruning the search space.

In these notes we mainly focus on saturation-based theorem proving methods including resolution, superposition and chaining. In this context, apart from ordering restrictions, constraints are also used to describe unification problems. This paper consist of three main parts.

First we introduce the main concepts for the case of pure equational logic, where rewriting and Knuth/Bendix completion are fundamental deduction methods. The topics which we cover include Birkhoff's theorem, first-order vs. inductive consequences, term rewrite systems, confluence, termination, orderings and critical pair computation (i.e., superposition).

In the second part, we extend these ideas to saturation-based methods for general first-order clauses, including superposition and chaining-based inference systems, completeness proofs by means of the model generation method and an abstract notion of redundancy for inferences.

The third part covers several extensions, with special attention to those aspects for which constraints play an important role, like basic strategies,

* Both authors supported by the ESPRIT working group CCL-II, ref. WG # 22457.

deduction modulo equational theories, constraint-based redundancy methods, and deduction-based complexity analysis and decision procedures.

For more details and further references the reader is referred to [BG98] and [NR01].

4.2 Equality Clauses

We briefly recall some basic concepts on first-order logic with equality. In particular, we give the syntax and semantics for the fragment of clausal (and hence only universally quantified) formulae. We refer the reader to standard textbooks such as [Fit90] for clausal normal form transformations of arbitrarily quantified first-order formulae.

4.2.1 Syntax: Terms, Atoms, Equations, and Clauses

Let $\mathcal{F}$ be a set of *function symbols*, $\mathcal{X}$ an infinite set of *variables* (disjoint with $\mathcal{F}$), and a function *arity*: $\mathcal{F} \to \mathbb{N}$ associating an *arity* (a natural number) to each function symbol. Function symbols f with $\mathit{arity}(f) = n$ are called *n-ary* symbols. In particular, f is called *unary*, if $n = 1$, *binary*, if $n = 2$, and a *constant symbol*, if $n = 0$.

$T(\mathcal{F}, \mathcal{X})$ denotes the set of *terms with variables* or simply *terms*: a term is a variable or an expression $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms for $1 \leq i \leq n$.

Note that terms can be seen as ordered trees where the leaves are labeled by constant symbols or variables, and where every other node is labeled by a function symbol of arity at least 1. The nodes in terms can be addressed by *positions* which are sequences of natural numbers. The root in a term is addressed by position λ (denoting the empty sequence) and if p is the address of a node v in t having the $n \geq 0$ successors $v_1, \ldots, v_n$, then these have addresses $p.1, \ldots, p.n$, respectively. We say that a function symbol or a variable *occurs* (at position p) in a term t if the node addressed by p carries that symbol as a label. By $\mathit{vars}(t)$ we denote the set of all variables occurring in t.

Let $\mathcal{P}$ be a set of *predicate symbols* with arity (disjoint from $\mathcal{F}$ and $\mathcal{X}$). Then $p(t_1, \ldots, t_n)$ is an *atom* if p is an n-ary predicate symbol and the t_i are terms for $1 \leq i \leq n$. In the sequel it will often simplify matters technically to also view atoms as terms of a specific sort which, in particular, cannot occur as proper subterms of other terms.

By $t|_p$ we denote the *subterm of t at position p*: we have $t|_\lambda = t$, and $f(t_1, \ldots, t_n)|_{i.p} = t_i|_p$ if $1 \leq i \leq n$ (and is undefined if $i > n$). We also write $t[s]_p$ to denote the term obtained by replacing in t the subterm at position p by the term s. For example, if t is $f(a, g(b, h(c)), d)$, then $t|_{2.2.1} = c$, and $t[d]_{2.2} = f(a, g(b, d), d)$.

A *multiset* over a set S is a function $M\colon S \to I\!N$. The union of multisets is defined by $M_1 \cup M_2(x) = M_1(x) + M_2(x)$. We sometimes also use a set-like notation: $M = \{a, a, b\}$ denotes the multiset M where $M(a) = 2$, $M(b) = 1$, and $M(x) = 0$ for $x \neq a$ and $x \neq b$.

A first-order *clause* is a pair of finite multisets of atoms (Γ, Δ), written $\Gamma \to \Delta$, where Γ is called the *antecedent*, and Δ the *succedent* of the clause. The *empty clause* $\Box$ is a clause where both Γ and Δ are empty, and a *Horn clause* is a clause whose succedent Δ contains at most one atom. In clauses we often use comma to denote the union of multisets or the inclusion of an atom in a multiset; for example, we write $A, \Gamma, \Gamma' \to \Delta$ instead of $\{A\} \cup \Gamma \cup \Gamma' \to \Delta$. A literal is either an atom (a *positive literal*) or a negation thereof (a *negative literal*). Clauses $A_1, \ldots, A_n \to B_1, \ldots, B_k$ can also be written as a disjunction of literals $\neg A_1 \vee \ldots \vee \neg A_n \vee B_1 \vee \ldots \vee B_k$. Hence, the A_i are called the *negative* atoms, and the B_j the *positive* atoms, respectively, of C.

4.2.2 Semantics

An *interpretation* I for $\mathcal{F}$ and $\mathcal{P}$ consists of three components:

1. a non-empty set D, called the *domain*,
2. for each n-ary f in $\mathcal{F}$, a function $f_I\colon D^n \to D$, and
3. for each n-ary p in $\mathcal{P}$, a function $p_I\colon D^n \to \{\mathit{true}, \mathit{false}\}$

Any interpretation I is a mathematical model of some intended real-world meaning of the symbols in the vocabulary. Variables denote elements in the domain D. This is formalized by the notion of an *assignment* which is a function $as\colon \mathcal{X} \to D$. A clause C is *true* in I, written $I \models C$, if for every assignment as, at least one of its negative atoms evaluates to *false*, or at least one of its positive atoms evaluates to *true*, according to the following evaluation function *eval*:

$$\begin{array}{lll} \mathit{eval}(p(t_1, \ldots, t_n)) & = p_I(\mathit{eval}(t_1), \ldots, \mathit{eval}(t_n)) & \text{if } p \in \mathcal{P} \\ \mathit{eval}(f(t_1, \ldots, t_n)) & = f_I(\mathit{eval}(t_1), \ldots, \mathit{eval}(t_n)) & \text{if } f \in \mathcal{F} \\ \mathit{eval}(x) & = as(x) & \text{if } x \in \mathcal{X} \end{array}$$

Instead of saying that C is true in I, we also say that I *satisfies* C or that I *is a model of* C. A set of clauses S is true in an interpretation I if every clause of S is true in I. From these stipulations we see that the variables in clauses are implicitly universally quantified.

Propositional logic arises as the particular case when all predicate symbols have arity zero, and hence the formulae contain no function symbols or variables. Then an interpretation just says which predicate symbols are *true* and which ones are *false*.

4.2.3 Why Is Satisfiability so Interesting?

Let us now consider a number of relevant questions that appear in practical applications of (any) logic. If we describe a certain real-world problem such as a circuit, a communication protocol, a data base by a logical formula F, we may ask ourselves:

1. whether F has no inherent contradictions; this amounts to whether there exists an interpretation I such that $I \models F$, i.e, whether F is *satisfiable.*
2. whether F is true in all situations; this amounts to the question whether $I \models F$ for every interpretation I, i.e., whether F is a *tautology.*
3. whether another formula G holds in all the real-world situations described by F; this amounts to the question whether $I \models G$ whenever $I \models F$, for every interpretation I, i.e., whether G is a *logical consequence* of F. In this case we also simply say that G *follows from* F, and we write $F \models G$ (overloading the symbol $\models$).
4. whether another (perhaps simpler in some sense) formula G describes exactly the same real-world situations as F; this amounts to the question whether $I \models G$ if, and only if, $I \models F$, for every interpretation I, i.e., whether F and G are *logically equivalent.*

Here we concentrate on the satisfiability problem for sets of clauses, since all the aforementioned questions (for full first-order logic) can be reduced to it: F is a tautology iff $\neg F$ is unsatisfiable, $F \models G$ iff $F \wedge \neg G$ is unsatisfiable. F and G are logically equivalent iff $(F \wedge \neg G) \vee (G \wedge \neg F)$ is unsatisfiable. Also, numerous satisfiability-preserving transformations of arbitrary formulae F into sets of clauses S have been described in the literature (but such transformations are out of the scope of this course).

For propositional logic, the satisfiability problem of a finite clause set can be decided as follows. If the cardinality of $\mathcal{P}$ is n, then there are 2^n possible interpretations (the rows of the so-called truth-tables). For each interpretation the formulae can be evaluated. Less naive algorithms are used in practice (but the problem is well-known to be NP-complete, and hence no polynomial algorithm is believed to exist).

4.2.4 Herbrand Interpretations

It is well-known that the satisfiability problem for sets of clauses S is undecidable, that is, there is no algorithm that takes as input any clause set S and that always terminates producing a correct yes/no answer regarding the satisfiability of S. So what can we do if we want to (try to) find out whether or not a given clause set S is satisfiable? It is certainly not a good idea to just generate interpretations and, for each one of them, try to check whether it satisfies S. One of the reasons is that the set of interpretations to be tried is large, usually infinite. Fortunately, as we will explain now, it suffices to consider a much smaller (although still not finite) set of interpretations, the so-called *Herbrand* interpretations: we will see that a set of clauses S is satisfiable if, and only if, it has a Herbrand model. Both the domain and the interpretation of the function symbols are fixed by definition in Herbrand interpretations:

Let $T(\mathcal{F})$ denote the set of *ground* terms (also sometimes called *closed* terms) consists of all terms without variables, that is, the terms $f(t_1, \dots, t_n)$, where f is an n-ary function symbol and the t_i are ground terms for $1 \leq i \leq n$.

A *Herbrand* interpretation is an interpretation with domain $T(\mathcal{F})$, and where $f_I(t_1, \dots, t_n) = f(t_1, \dots, t_n)$ for every n-ary function symbol f and ground terms $t_1, \dots, t_n$.

(Some people find this definition a bit tricky: the rather similar expressions $f_I(t_1, \dots, t_n)$ and $f(t_1, \dots, t_n)$ have completely different meanings; we are saying that the function f_I applied to the terms $t_1, \dots, t_n$ gives as result the term $f(t_1, \dots, t_n)$. In Herbrand interpretations, functions are interpreted as term constructors.) Note that two Herbrand interpretations for given $\mathcal{F}$ and $\mathcal{P}$ can only differ in the interpretation of the predicate symbols. Therefore it is appropriate to identify any Herbrand interpretation I with the set of ground atoms that are true in I. Under this view, a ground clause $\Gamma \to \Delta$ is true in I if either $\Gamma \not\subseteq I$, or else $\Delta \cap I \neq \emptyset$.

It is not difficult to see that a set of clauses S is satisfiable if, and only if, it has a Herbrand model. For the non-trivial implication, if $I \models S$, we can construct the Herbrand model I' as the set of all ground atoms $p(t_1, \dots, t_n)$ for which $p_I(\mathit{eval}(t_1), \dots, \mathit{eval}(t_n)) = \mathit{true}$, where eval is the evaluation function associated with I. The details are left as an exercise to the reader.

This result is the basis for many deduction methods in first-order logic: in order to show unsatisfiability, it suffices to show that there is no Herbrand model.

4.2.5 Herbrand's Theorem

A *substitution* σ is a mapping from variables to terms. We sometimes write substitutions as a set of pairs $x \mapsto t$, where x is a variable and t is a term. For example, if $\sigma(x) = a$ and $\sigma(y) = b$ and $\sigma(z) = z$ for all $z \in \mathcal{X} \setminus \{x, y\}$, then we write $\sigma = \{x \mapsto a, y \mapsto b\}$. The *domain* of a substitution σ is the set of variables x for which $\sigma(x) \neq x$. We require the domain of substitutions to be finite. A substitution can be extended to a function from terms (atoms, clauses) to terms (atoms, clauses) as follows: using a postfix notation, $t\sigma$ denotes the result of simultaneously replacing in t every $x \in \mathit{Dom}(\sigma)$ by $x\sigma$. For example, $f(x, y)\{x \mapsto g(y), y \mapsto b\} = f(g(y), b)$. A substitution σ is *ground* if its range is $T(\mathcal{F})$.

A *ground instance* of a clause C is a ground clause $C\sigma$ for some ground substitution σ. The set of ground instances $\mathit{gnd}(S)$ of a set of clauses S is the union of the sets of ground instances of the individual clauses in S. $\mathit{gnd}(S)$ is also called the *Herbrand expansion* of S. Ground atoms are not essentially different from propositional variables. Different ground atoms have no semantic connections between them so that their truth values can be chosen independently in any interpretation. Therefore propositional (clause) logic also arises as the restriction of first-order clause logic to the variable-free case.

Now there are several important results to be mentioned. The first one (Gödel-Herbrand-Skolem), states that S is satisfiable if, and only if, $gnd(S)$ is satisfiable. The reason is that, as we have seen, it suffices to consider Herbrand interpretations, where the variables range over the domain of ground terms, i.e, every assignment *as* to be tried is a ground substitution. This result essentially reduces the first-order satisfiability problem to the satisfiability problem for (in general, infinite) sets of propositional clauses. By the *compactness theorem* for propositional logic, an infinite set S of propositional clauses is satisfiable if, and only if, every finite subset of S is satisfiable. From this, we finally obtain Herbrand's theorem: a set of clauses S is unsatisfiable if, and only if, there is an unsatisfiable finite subset of $gnd(S)$.

This gives us a semi-decision procedure for the unsatisfiability problem, that is, a procedure that takes as input a set of clauses S and that (i) terminates with output "yes" if S is unsatisfiable, and (ii) terminates with output "no" or does not terminate otherwise. This procedure just enumerates $gnd(S)$, and from time to time checks whether the current part is satisfiable in propositional logic.

Compactness and Herbrand's theorem are the key to many classical completeness results for refutation calculi including resolution. Here, we will take a different route in that our completeness proofs for resolution and paramodulation will *not* require clause sets to be finite, hence Herbrand's theorem and compactness will follow as a corollary. For readers interested in direct proofs of compactness we refer to [Fit90].

4.2.6 Equality

Because of its importance in practice, the equality predicate deserves special attention in first-order logic and theorem proving. Equality can be defined within first-order logic as a binary predicate $\simeq$ (written in infix-notation) satisfying this set of clauses $\mathcal{E}$:

$$\begin{array}{rl}
\rightarrow x \simeq x & \textit{(reflexivity)}\\
x \simeq y \rightarrow y \simeq x & \textit{(symmetry)}\\
x \simeq y \wedge y \simeq z \rightarrow x \simeq z & \textit{(transitivity)}\\
x_1 \simeq y_1 \wedge \ldots \wedge x_n \simeq y_n \rightarrow f(x_1, \ldots, x_n) \simeq f(y_1, \ldots, y_n) & \textit{(monotonicity)}\\
x_1 \simeq y_1 \wedge \ldots \wedge x_n \simeq y_n \qquad\qquad\qquad & \\
\wedge\, p(x_1, \ldots, x_n) \rightarrow p(y_1, \ldots, y_n) & \textit{(monotonicity)}
\end{array}$$

where the monotonicity axioms are, in fact, axiom *schemes*, and one instance is needed for each n-ary function and predicate symbol f and p, respectively. A binary relation satisfying $\mathcal{E}$ is called a *congruence*. An *equation* is an atom with $\simeq$ as its predicate. We will build-in the symmetry of equality into our notation, that is, we will use $s \simeq t$ to also denote the reverse equation $t \simeq s$. An *equality clause* is a clause in which all atoms are equations. By an *equality interpretation* we mean a interpretation in which $\simeq$ is a congruence.

Another way to deal with the equality predicate is to build it into the logic. Then we obtain *first-order logic with equality* (FOLE) where the only interpretations I under consideration are the ones where the semantics of the equality predicate is "being the same element of D", that is, where $\simeq_I$ is fixed by definition as: $\simeq_I(x, y) = true$ if, and only if, x and y are the same element of D. For our purposes, the view of equality as a congruence will be more appropriate. It is justified by the fact that a set of clauses S is satisfiable in FOLE if, and only if, $S \cup \mathcal{E}$ is satisfiable in FOL (without equality). The proof of this fact is left as a relatively easy exercise to the reader.

Note that FOLE is more "expressive" in the following sense. In FOLE $x \simeq a \vee x \simeq b$ expresses that models have cardinality at most two, which cannot be expressed in FOL without equality, where, if $I \models F$ and some a is in the domain D of I, one can always build another model I' with domain $D \cup \{a'\}$ where a' is a "clone" of a that behaves identically to a under all functions and predicates. (More generally, in FOL without equality, if I is a model of F, then F also has models over domains with any cardinality larger than that of I.)

From now on, unless explicitly indicated otherwise, we will restrict attention to equality interpretations. In particular, $F \models G$ will mean implication with respect to equality interpretations, that is, $I \models G$ whenever $I \models F$, for every equality interpretation I.

4.2.7 Equality Herbrand Interpretations

If $\Rightarrow$ is a binary relation on a set, then $\Leftarrow$ denotes its inverse, $\Leftrightarrow$ its symmetric closure, $\Rightarrow^+$ its transitive closure and $\Rightarrow^*$ its reflexive-transitive closure. A relation $\Rightarrow$ on terms is called *monotonic* if $s \Rightarrow t$ implies $u[s]_p \Rightarrow u[t]_p$ for all terms s, t and u and positions p in u. A *congruence* on terms is a reflexive, symmetric, transitive and monotonic relation on terms. It is easy to see that congruences on terms as just defined are precisely the relations on $T(\mathcal{F})$ that satisfy $\mathcal{E}$. Therefore, by an *equality Herbrand interpretation* we mean a Herbrand interpretation in which $\simeq$ is a congruence. Considering equality as a congruence (rather than dealing with FOLE), the concept of a Herbrand interpretation remains meaningful and the related results continue to hold.

In what follows, we will consider equality to be the only predicate, since for every other predicate symbol p, (positive or negative) atoms $p(t_1 \dots t_n)$ can be expressed as (positive or negative) equations $p(t_1 \dots t_n) \simeq \top$, where $\top$ is a new special symbol, and where p is considered a function symbol rather than a predicate symbol.[1] It is easy to see that this transformation preserves satisfiability. The proof of this fact is left as an exercise to the reader. (Hint: one can "translate" the interpretations accordingly such that a ground atom is true in an equality Herbrand interpretation I iff in the Herbrand interpretation I' over the modified signature the term $p(t_1 \dots t_n)$ is congruent to $\top$. Note, however that I and I' are not isomorphic since two ground atoms that are false in I need not be in the same congruence class of I'.) Also by a Herbrand interpretation we shall from now on always mean an equality Herbrand interpretation.

4.3 The Purely Equational Case: Rewriting and Completion

Here we consider purely equational logic, that is, FOL where all formulae are equations (that is, positive unit clauses). Given a set E of equation and another equation $s \simeq t$, called a *query*, the problem whether or not $E \models s \simeq t$ is called the *word problem* for E. This problem can be attacked through reasoning using Leibniz' law for the "replacement of equals by equals": an equation $s \simeq t$ can be "applied" to a term u by replacing a subterm of the form $s\sigma$ of u by $t\sigma$.

More formally, if E is a set of equations, the relation $\Leftrightarrow_E^*$ is the smallest congruence such that $s\sigma \Leftrightarrow_E^* t\sigma$ for all $s \simeq t$ in E and all substitutions σ. Birkhoff's theorem states that the application of equations is correct and complete for solving the word problem: for every equation $s \simeq t$ and set of equations E, we have $E \models s \simeq t$ if, and only if, $s \Leftrightarrow_E^* t$. The reason is of course that $\Leftrightarrow_E^*$ is precisely the unique minimal Herbrand model of E; it exists for all E, and hence all sets of equations E are satisfiable.

However, Birkhoff's theorem does not provide a practical (or automatizable) deduction method. If, as an exercise, the reader tries to prove a simple statement in group theory like:

$$\{ \ (x+y)+z \simeq x+(y+z), \ \ e+x \simeq x, \ \ i(x)+x \simeq e \ \} \models \ \ x+e \simeq x$$

it quickly becomes clear why: there are too many ways of applying the equations, and hence too many (sometimes larger and larger) terms are obtained.

One way to reduce the search space is to restrict the application of the equations, by only replacing "big" terms by "smaller" ones. Let us now formalize this idea.

First we need some more definitions on relations. If $\Rightarrow$ is a relation, we write $s \Rightarrow^! t$ if $s \Rightarrow^* t$ and there is no t' such that $t \Rightarrow t'$. Then t is called *irreducible* and a *normal form* of s with respect to $\Rightarrow$. The relation $\Rightarrow$ is

[1] To avoid meaningless expressions in which predicate symbols occur at proper subterms one should, however, adopt a two-sorted type discipline on terms in the encoding.

well-founded or *terminating* if there exists no infinite sequence $s_1 \Rightarrow s_2 \Rightarrow \ldots$ and it is *confluent* or *Church-Rosser* if the relation $\Leftarrow^* \circ \Rightarrow^*$ is contained in $\Rightarrow^* \circ \Leftarrow^*$. It is *locally confluent* if $\Leftarrow \circ \Rightarrow \ \subseteq \ \Rightarrow^* \circ \Leftarrow^*$. By Newman's lemma, terminating locally-confluent relations are confluent.

A *rewrite rule* is an ordered pair of terms (s, t), written $s \Rightarrow t$, and a set of rewrite rules R is a *term rewrite system* (TRS). The *rewrite relation* by R on $T(\mathcal{F}, \mathcal{X})$, denoted $\Rightarrow_R$, is the smallest monotonic relation such that $l\sigma \Rightarrow_R r\sigma$ for all $l \Rightarrow r \in R$ and all σ, and if $s \Rightarrow_R t$ then we say that s *rewrites (in one step) into* t under R.

A TRS R is called terminating, confluent, etc. if $\Rightarrow_R$ is. A TRS R is *convergent* if it is confluent and terminating. For convergent TRS R, every term t has a unique normal form with respect to $\Rightarrow_R$, denoted by $nf_R(t)$. (Exercise: prove this; hint: use Noetherian induction on $\Rightarrow$.) For a convergent system R (where R is seen as a set of equations), $s \Leftrightarrow_R^* t$ if, and only if, $nf_R(s) = nf_R(t)$. In that case, by Birkhoff's theorem, $s \simeq t$ is a logical consequence of R if, and only if, $nf_R(s) = nf_R(t)$. Therefore, if R is finite and convergent, the word problem for R is decidable by rewriting the two terms of any given query into normal form.

Rewriting a term s amounts to finding a subterm $s|_p$ such that $s|_p = l\sigma$ for some rule $l \Rightarrow r$ and substitution σ. Then we say that l *matches* $s|_p$, and that $s|_p$ is an *instance* of l. If a TRS R is terminating, then it must be the case that $vars(l) \supseteq vars(r)$ for every rule $l \Rightarrow r$ in R, so that in each rewrite step the substitution σ is completely determined by matching.

Now we have seen that the two important properties that make TRS very useful for deduction are confluence and termination. Let us now see how they can be assured in practice.

4.3.1 How to Ensure Termination?

Termination of a TRS is undecidable, i.e., there is no algorithm that, when given as input a TRS R, tells us in finite time whether R is terminating or not. This is not surprising, since for every Turing machine there exists a (even one-rule) TRS that simulates it; in other words, rewriting is a Turing-complete computation mechanism. However, in spite of the fact that termination is undecidable, there exist general-purpose methods based on term orderings that frequently suffice in practice.

A (strict partial) *ordering* on $T(\mathcal{F}, \mathcal{X})$ is an irreflexive transitive relation $\succ$. It is a *reduction* ordering if it is well-founded and monotonic, and *stable under substitutions*, that is, $s \succ t$ implies $s\sigma \succ t\sigma$ for all substitutions σ.

It turns out that a TRS R is terminating if, and only if, R is contained in a reduction ordering, that is, there exists some reduction ordering $\succ$ such that $l \succ r$ for every rule $l \Rightarrow r$ of R. The reader may prove this as an exercise (hint: show that if R terminates then $\Rightarrow_R{}^+$ itself is a reduction ordering).

But unconstrained term rewriting cannot deal properly with inherently non-terminating equations such as the commutativity axiom for a binary function symbol. Therefore the concepts have been generalized to ordered term rewriting. An *ordered* TRS is a pair $(E, \succ)$ where E is a set of equations and $\succ$ is a reduction ordering. Rewriting with an ordered TRS is done by applying equations of E in whatever direction agrees with $\succ$: the *ordered rewriting relation* defined by $(E, \succ)$ is the smallest monotonic binary relation $\Rightarrow_{E,\succ}$ on terms such that $s\sigma \Rightarrow_{E,\succ} t\sigma$ whenever $s \simeq t \in E$ (or $t \simeq s \in E$) and $s\sigma \succ t\sigma$. Hence an ordered TRS can handle *any* set of equations E, being terminating by definition. Note that $\Rightarrow_{E,\succ}$ coincides with $\Rightarrow_{E^\succ}$ where $E^\succ$ is the set of rewrite rules $s\sigma \Rightarrow t\sigma$ such that $s \simeq t \in E$ (or $t \simeq s \in E$), and σ a substitution such that $s\sigma \succ t\sigma$. Note also that since reduction orderings are only partial orderings on $T(\mathcal{F}, \mathcal{X})$, the relation $\Leftrightarrow^*_E$ is in general a strict superset of $\Leftrightarrow^*_{E^\succ}$. Except for trivial cases there are no reduction orderings that are total on $T(\mathcal{F}, \mathcal{X})$. Fortunately, reduction orderings that are total on ground terms do exist. For such a *complete reduction ordering* $\succ$, the relations $\Leftrightarrow^*_E$ and $\Leftrightarrow^*_{E^\succ}$ coincide on $T(\mathcal{F})$.

4.3.2 How to Get Reduction Orderings in Practice?

In practical deduction systems, useful reduction orderings can be obtained by easily automatisable, general-purpose methods. Here we will explain one of them, the so-called recursive path ordering (RPO). It is based on a well-founded ordering $\succ_\mathcal{F}$ on the function symbols $\mathcal{F}$.[2] Here $\succ_\mathcal{F}$ is called the *precedence*.

Let $\mathcal{F}$ be the disjoint union of two sets *lex* and *mul*, the symbols with *lexicographic* and *multiset status*, respectively, and let $=_{mul}$ denote the equality of terms up to the permutation of direct arguments of symbols with multiset status; for example, if $f \in mul$ and $g \in mul$, then $f(a, g(b, c, d), e) =_{mul} f(e, a, g(c, b, d))$.

Then RPO is defined as follows: $s \succ_{\text{rpo}} t$ if t is a variable that is a proper subterm of s or $s = f(s_1, \ldots, s_m)$ and $t = g(t_1, \ldots, t_n)$ and

1. $s_i \succ_{\text{rpo}} t$ or $s_i =_{mul} t$, for some i with $1 \leq i \leq m$ or
2. $f \succ_\mathcal{F} g$, and $s \succ_{\text{rpo}} t_j$, for all j with $1 \leq j \leq n$ or
3. $f = g$, $f \in lex$, $\langle s_1, \ldots, s_n \rangle \succ^{\text{lex}}_{\text{rpos}} \langle t_1, \ldots, t_n \rangle$, and $s \succ_{\text{rpo}} t_j$, for all j with $1 \leq j \leq n$
4. $f = g$, $f \in mul$, $\{s_1, \ldots, s_n\} \succ^{\text{mul}}_{\text{rpos}} \{t_1, \ldots, t_n\}$

where $\langle s_1, \ldots, s_n \rangle \succ^{\text{lex}}_{\text{rpos}} \langle t_1, \ldots, t_n \rangle$ if $\exists j \leq n$ s.t. $s_j \succ_{\text{rpo}} t_j$ and $\forall i < j$ $s_i =_{mul} t_i$. Furthermore, $\succ^{\text{mul}}_{\text{rpos}}$ is the multiset extension of $\succ_{\text{rpo}}$, defined as the smallest ordering such that for any $m \geq 0$, $S \cup \{s\} \succ^{\text{mul}}_{\text{rpos}} S' \cup \{t_1, \ldots, t_m\}$ whenever S is equal to S' up to $=_{mul}$ and $s \succ_{\text{rpo}} t_i$ for all i in $1 \ldots n$.

[2] This allows $\mathcal{F}$ to be infinite which is useful when infinitely many Skolem constants are needed.

The *lexicographic path ordering* is the particular case of RPO where $\mathcal{F} = lex$, and the *multiset path ordering* (or RPO without status) is the particular case where $\mathcal{F} = mul$.

For example, if $f \in mul$ and $g \in lex$, and $f \succ_\mathcal{F} g \succ_\mathcal{F} a \succ_\mathcal{F} b$, then $f(b, g(a, b, b), b) \succ_{\text{rpo}} g(a, f(a, g(b, a, a)), a)$. Exercise: explain why.

The RPO is a reduction ordering for every choice of mul, lex and $\succ_\mathcal{F}$. However in practice it is important that these be chosen appropriately so that deduction takes place with appropriate normal forms of terms and clauses.

4.3.3 How to Ensure Confluence?

Once we know that termination can be obtained by working with reduction orderings, we can apply Newman's lemma: terminating locally-confluent relations are confluent. Now let us analyze in which situations local confluence of (ordered) rewriting fails.

Assume a term t can be rewritten in two different ways, with instances by, respectively, substitutions σ and σ' of equations $l \simeq r$ and $l' \simeq r'$, at positions p and p', respectively. We may assume that variables in the two (not necessarily different) equations have been renamed so that the domains of σ and σ' are disjoint. There are three possible situations:

1. p and p' are disjoint positions (no one is prefix of the other). Then we have:

$$\begin{tikzcd} & t[l\sigma]_p[l'\sigma']_{p'} \arrow[dl] \arrow[dr] & \\ t[r\sigma]_p[l'\sigma']_{p'} & & t[l\sigma]_p[r'\sigma']_{p'} \end{tikzcd}$$

   But this does not lead to failure of local confluence, because $t[r\sigma]_p[l'\sigma']_{p'}$ and $t[l\sigma]_p[l'\sigma']_{p'}$ are *joinable*, that is, they can be rewritten into the same term (in this case, in one step):

$$\begin{tikzcd} t[r\sigma]_p[l'\sigma']_{p'} \arrow[dr] & & t[l\sigma]_p[r'\sigma']_{p'} \arrow[dl] \\ & t[r\sigma]_p[r'\sigma']_{p'} & \end{tikzcd}$$

   For example, assume $a \simeq b$ and $c \simeq d$ are equations and $f(a, c)$ rewrites into $f(b, c)$ and into $f(a, d)$. Then we have:

$$\begin{tikzcd} & f(a,c) \arrow[dl] \arrow[dr] & \\ f(b,c) \arrow[dr] & & f(a,d) \arrow[dl] \\ & f(b,d) & \end{tikzcd}$$

2. If, say, p is a prefix of p', then $p' = p.q$ for some position q, and the situation is called an *overlap*. Then

$$\begin{tikzcd} & t[l\sigma[l'\sigma']_q]_p \arrow[dl] \arrow[dr] & \\ t[r\sigma]_p & & t[l\sigma[r'\sigma']_q]_p \end{tikzcd}$$

and q is a position inside $l\sigma$.
(a) If some prefix of q is a variable x of l, then the overlap is called a *variable overlap* which also does not lead to a failure of local confluence. Let us see why both terms are joinable. The step with $l' \to r'$ rewrites $x\sigma$ into some term r''. Let σ' be extended to the variables in l and r such that $\sigma'(x) = r''$ and $\sigma'(y) = \sigma(y)$ if $y \neq x$. Assuming that x occurs n times in r and m times in l, we have:

$$\begin{tikzcd}
t[r\sigma]_p \arrow[rrdd, "n"] & & & & t[l\sigma[r'\sigma']_q]_p \arrow[dl, "m-1"] \\
& & & t[l\sigma']_p \arrow[dl] & \\
& & t[r\sigma']_p & &
\end{tikzcd}$$

For example, if $a \simeq b$ and $f(x,x,x,x) \simeq g(x,x)$ are applied to rewrite $f(a,a,a,a)$ into $g(a,a)$ and into $f(a,b,a,a)$, then we have:

$$\begin{tikzcd}
& & f(a,a,a,a) \arrow[dll] \arrow[dr] & & \\
g(a,a) \arrow[ddr, "2"] & & & f(a,b,a,a) \arrow[dl, "3"] & \\
& & f(b,b,b,b) \arrow[dl] & & \\
& g(b,b) & & &
\end{tikzcd}$$

(b) If no prefix of q is a variable x of l, then $l|_q$ is a function symbol. This case is called a *critical overlap* and can be a situation of failure of local confluence. For example, if $f(a,x) \simeq g(x)$ and $h(f(y,b)) \simeq h(y,y)$ can be applied, according to $\succ$, as follows:

$$\begin{tikzcd}
& h(f(a,b)) \arrow[dl] \arrow[dr] & \\
h(g(b)) & & h(a,a)
\end{tikzcd}$$

then $h(g(b))$ and $h(a,a)$ may not be joinable.

From these consideration it follows that we only have to consider critical overlaps. How can we detect them? How can we check whether they witness a counterexample to local confluence? And what can we do if this is indeed the case?

4.3.4 How Can We Detect Critical Overlaps?

We have to detect all terms that contain overlapped *common instances* of left hand sides of possible rewrite steps. But this can be detected by *unification*: two terms s and t are *unifiable* if there exists some substitution σ such that

$s\sigma = t\sigma$, and then σ is called a *unifier* of s and t. There exist unification algorithms that, when given terms s and t compute their *most general unifier* $mgu(s,t)$, that is, a unifier σ such that any other unifier θ of s and t is an instance of σ (there is some θ' such that $x\theta = x\sigma\theta'$ for all x).

In order to get a bit more intuition about unification, let us now briefly see what unification algorithms do (for more details, see the other courses in this summer school). The following rule set can be applied with different strategies leading to different algorithms. It works on unification problems that are sets of (unordered) pairs of terms $\{s_1 \doteq t_1, \dots, s_n \doteq t_n\}$, for which a simultaneous mgu σ is computed for which $s_i\sigma = t_i\sigma$ for all $i \in 1 \dots n$:

$$\begin{array}{lll}
0. & P \cup \{t \doteq t\} & \Rightarrow P \\
1. & P \cup \{f(t_1, \dots, t_n) \doteq f(s_1, \dots, s_n)\} & \Rightarrow P \cup \{t_1 \doteq s_1, \dots, t_n \doteq s_n\} \\
2. & P \cup \{f(t_1, \dots, t_n) \doteq g(s_1, \dots, s_n)\} & \Rightarrow \text{fail}, \quad \text{if } f \neq g \\
3. & P \cup \{x \doteq t\} & \Rightarrow P\{x \mapsto t\} \cup \{x \doteq t\}, \\
 & & \quad \text{if } x \in vars(P),\ x \notin vars(t), \\
 & & \quad \text{and either } t \text{ is not a variable,} \\
 & & \quad \text{or else } t \in vars(P) \\
4. & P \cup \{x \doteq t\} & \Rightarrow \text{fail} \quad \text{if } x \in vars(t) \text{ and } x \neq t
\end{array}$$

It is easy to see that the rules transform any unification problem into an equivalent one (having the same set of unifiers), with "fail" representing the empty set of unifiers. Any problem in normal form, that is, to which none of the rules can be applied anymore, is a problem of the form $\{x_1 \doteq t_1, \dots, x_k \doteq t_k\}$ with x_i pairwise different and not occurring in any of the terms t_j. Such problems are trivially solved in that the substitution sending x_i to t_i is an mgu. Note that mgu's are unique up to renaming of variables in the following sense: if $x = y$ appears at some stage in the problem, then either all x's elsewhere in the problem are replaced by y's, and $x \mapsto y$ will be in the mgu, or the y's are replaced by x's, and $y \mapsto x$ is in the mgu. Also note that no new variables other than those occurring in initial unification problems are used to represent mgus.

Exercise: it remains to show that any strategy of applying the rules terminates. To do this, consider a well-founded ordering on unification problems comparing (i) their number of unsolved variables and (ii) (in case of the same number of unsolved variables), their number of function symbols.

Reconsidering the critical overlap produced by the rules $h(f(y,b)) \Rightarrow h(y,y)$ and $f(a,x) \Rightarrow g(x)$:

$$\begin{tikzcd}
 & h(f(a,b)) \arrow[dl] \arrow[dr] & \\
h(g(b)) & & h(a,a)
\end{tikzcd}$$

we can see that it corresponds to the unification of $f(a,x)$, which is used as a left hand side, with the subterm $f(y,b)$ of the term used as left hand side of the other step.

It is also clear that, in general, if the rules are $l \Rightarrow r$ and $l' \Rightarrow r'$ respectively, then the two terms obtained from the critical overlap are always $l[r']\theta$ and $r\theta$, for some unifier θ of l' and a non-variable $l|_p$. Moreover, if $\sigma = mgu(l', l_p)$ then the two terms are always an instance of $l[r']\sigma$ and $r\sigma$. The equation $l[r']\sigma \simeq r\sigma$ is then called a *critical pair* between the rules are $l \Rightarrow r$ and $l' \Rightarrow r'$. It represents a real critical overlap only if it corresponds to applications of the two rules in this direction according to the current $\succ$, that is, if there exists some substitution θ such that $l\sigma\theta \succ r\sigma\theta$ and $l'\sigma\theta \succ r'\sigma\theta$.

Now such a a critical pair is nothing else than the conclusion of an inference by *superposition*, the following inference rule between equations, that is parameterized by the reduction ordering $\succ$:

$$\frac{l' \simeq r' \qquad l \simeq r}{(l[r']_p \simeq r)\sigma}$$

where $\sigma = mgu(l', l|_p)$, l_p is not a variable, and there exists some substitution θ such that $l\sigma\theta \succ r\sigma\theta$ and $l'\sigma\theta \succ r'\sigma\theta$. Note that an implicit assumption on all inference rules that we present in this paper is that variables are appropriately renamed so that the premises of an inference rule do not share any variables.

Now we can summarize the results of this subsection: if all critical pairs between the equations of E are joinable by ordered rewriting with $(E, \succ)$, then $(E, \succ)$ is convergent.

4.3.5 What Can We Do If Some Critical Pair Is Not Joinable?

To answer this question, note that by Birkhoff's theorem, every critical pair between equations of E is a logical consequence of E. Hence if we add the critical pair to E, we do not change its logical meaning. Moreover, it restores local confluence for its critical overlap.

Returning to our previous example, adding the critical pair $h(g(b)) \simeq h(a, a)$ solves the situation of non-local confluence

$$\begin{tikzcd} & h(f(a,b)) \arrow[dl] \arrow[dr] & \\ h(g(b)) & & h(a,a) \end{tikzcd}$$

independently of whether $h(g(b)) \succ h(a, a)$ or $h(a, a) \succ h(g(b))$. On the other hand, to not render ordered rewriting incomplete, we have to require that each ground instance of each equation, be it an input equation or an equation generated by a critical pair inference, be orientable under the ordering. Therefore in the following we will make the usual assumption that $\succ$ is a complete reduction ordering, that is, is *total* on $T(\mathcal{F})$. RPO's with no function symbols of multiset status and with a total precedence are examples of complete reduction orderings. Being able to orient ground instances of equations is sufficient for our purposes. We want to solve word problems

$E \models s \simeq t$ by rewriting s and t to their normal forms, and hence may consider the variables of s and t to be (Skolem) constants. In other words, for solving word problems it suffices to rewrite ground terms (with ordered instances of possibly non-ground equations). Consequently, we may reformulate the notion of a critical pair as well as the superposition inference rule by requiring that there exists some *ground* substitution θ such that $l\sigma\theta \succ r\sigma\theta$ and $l'\sigma\theta \succ r'\sigma\theta$.

These ordering restrictions can be used in practice for reducing the number of critical pairs to be considered in several ways. By *ordering constraint* solving techniques, the restrictions can be decided for RPO and its variants. But since this can be expensive, instead of really checking whether there exists a θ such that, say, $l\sigma\theta \succ r\sigma\theta$, one can also use approximations. For instance, if $r\sigma \succ l\sigma$ (in a version of $\succ$ for terms with variables) then the critical pair is not needed.

Now, once we have added all non-joinable critical pairs, is the resulting E confluent? Unfortunately this is not the case (although this is not surprising, because otherwise it would provide a decision procedure for the word problem, which is undecidable in general). The reason is that the critical pairs that have been added can produce new critical pairs, which then have to be added, and so on. The (possibly non-terminating) procedure that does this until it obtains a set E all whose critical pairs are joinable, is called the (unfailing) Knuth-Bendix completion procedure.

Currently, most, if not all, state-of-the-art theorem provers for purely equational logic are variants of such Knuth-Bendix completion procedures. Indeed, they are semi-decision procedures for the word problem: if $E \models s \simeq t$, then completion with input E and a reduction ordering $\succ$ that is total on $T(\mathcal{F})$ will generate in finite time enough equations to rewrite s and t into the same normal form (provided it is applied with a *fair* control, i.e., no superposition inference is postponed forever). We will not prove this fact at this point, because it is a consequence of more general results that will be given later on.

4.3.6 First-Order vs. Inductive Consequences

In many cases, the intended meaning of a specification E is not its standard first-order semantics, i.e., the class of all its models, but rather a more specific model I. Perhaps the best known example is the *initial* or *minimal Herbrand model* semantics, where I is the unique minimal Herbrand model of a set of Horn clauses with or without equality E (an algebraic specification, a logic program, a deductive data base, ...). Other interesting semantics that are used in practice as well include *final* semantics, or, for possibly non-Horn specifications E, *perfect model* semantics or the class of all (minimal) Herbrand models.

For example, let $\mathcal{F}$ be $\{0, s, +\}$ and assume E consists of the equations $0 + x \simeq x$ and $s(x) + y \simeq s(x + y)$. Then I is isomorphic to the algebra of natural numbers with $+$ and hence properties like the commutativity of $+$ are true in I. Exercise: prove $I \models x{+}y \simeq y{+}x$, and prove that $E \not\models x{+}y \simeq y{+}x$.

Properties true in I are called *inductive* theorems, since they usually require induction to prove them. However, there are so-called *implicit* induction methods that do not require the user of a (semi-)automatic inductive theorem prover to provide explicit induction schemes. Instead, induction can be done automatically, using well-founded orderings $\succ$ on terms, such as the ones we have seen earlier in this section.

Some implicit induction methods like the *proof-by-consistency* approach of [Bac88] are closely related to completion. Another simple one that is not based on completion, but rather on the notions of redundancy that we will see in the next section, is given in [Nie99]. A more detailed analysis of inductive proofs is unfortunately out of the scope of these notes, and in what follows we will only consider classical first-order semantics of equational theories.

4.4 Superposition for General Clauses

The ideas on ordering restrictions explained so far for the case of unit equations can be extended to obtain refutationally complete procedures for general clauses. Then superposition becomes a restricted version of the following inference rule of *paramodulation* [RW69]:

$$\frac{C \vee s \simeq t \qquad D}{(\ C \vee D[t]_p\ )\sigma} \qquad \text{if } \sigma = mgu(s, D|_p)$$

In essence, superposition is paramodulation where $D|_p$ is not a variable, and where inferences take place *with* and *on maximal* (with respect to $\succ$) sides of *maximal* literals (in this paramodulation inference rule, we say that the inference takes place *with s on $D|_p$*).

In the following we further develop this idea, starting with the simple case of ground Horn clauses, and introducing the more general techniques as we go along.

4.4.1 Refutational Completeness

An inference system $\mathcal{I}nf$ is called *refutationally complete* if from every inconsistent (unsatisfiable) set S of clauses the empty clause $\Box$ can be inferred by $\mathcal{I}nf$.

Here we will introduce the *model generation method*, one of the main methods for showing the refutational completeness of paramodulation-based inference systems, and the one that has been used for obtaining most results on deduction with symbolic constraints in this context. The main idea is as follows. Assume that S is *closed* under the given inference system $\mathcal{I}nf$ (that is, S contains all conclusions of inferences by $\mathcal{I}nf$ with premises in S) and assume $\square \notin S$. Then it suffices to show that S is satisfiable. This is done by constructing an equality Herbrand model $\Leftrightarrow^*_{R_S}$ for S, where R_S is a ground TRS that is *generated* from S (remember that a congruence on $T(\mathcal{F})$ like $\Leftrightarrow^*_{R_S}$ is an equality Herbrand interpretation). In the following, instead of $\Leftrightarrow^*_R$, we will usually write R^*. Later on, the requirement of S to be closed with respect to the inference system $\mathcal{I}nf$ will be weakened into *saturatedness* with respect to $\mathcal{I}nf$, that is, closure up to *redundant* inferences.

4.4.2 The Model Generation Method

We first consider the simple case of *ground Horn clauses*. In the following, let $\succ$ be a given total reduction ordering on ground terms. We will use the fact that such orderings also enjoy the *subterm property*, that is, $\succ \supseteq \rhd$, where $\rhd$ denotes the strict subterm ordering, defined: $s \rhd t$ if, and only if, $s|_p = t$ for some $p \neq \lambda$. Exercise: prove that any total reduction ordering on $T(\mathcal{F})$ is a simplification ordering, i.e., it has the subterm property.

Let $\Gamma \to l \simeq r$ be a ground clause where Γ is of the form $\{u_1 \simeq v_1, \ldots, u_n \simeq v_n\}$. The term l is called *strictly maximal* in $\Gamma \to l \simeq r$ if it is greater (with respect to $\succ$) than any other term in the clause, that is, if

$$l \succ r \ \wedge \ l \succ u_1 \ \wedge \ l \succ v_1 \ \wedge \ldots \wedge \ l \succ u_n \ \wedge \ l \succ v_n$$

In the following, this expression will be denoted by $l \succ r, \Gamma$. Similarly, l is called *maximal* in $\Gamma \to l \simeq r$ if $l \succeq t$, for each term t in r, Γ, where $\succeq$ denotes the reflexive closure of $\succ$. In the latter case we will also write $l \succeq r, \Gamma$. The inference system $\mathcal{G}$ for ground Horn clauses with equality is the following:

superposition right:

$$\frac{\Gamma' \to l \simeq r \qquad \Gamma \to s \simeq t}{\Gamma', \Gamma \to s[r]_p \simeq t} \quad \text{if} \quad \begin{array}{l} s|_p = l \text{ and} \\ l \succ r, \Gamma' \text{ and } s \succ t, \Gamma. \end{array}$$

superposition left:

$$\frac{\Gamma' \to l \simeq r \qquad \Gamma, s \simeq t \to \Delta}{\Gamma', \Gamma, s[r]_p \simeq t \to \Delta} \quad \text{if} \quad \begin{array}{l} s|_p = l \text{ and} \\ l \succ r, \Gamma',\ s \succ t \text{ and } s \succeq \Gamma, \Delta. \end{array}$$

reflexivity resolution:

$$\frac{\Gamma, s \simeq s \to \Delta}{\Gamma \to \Delta} \quad \text{if} \quad s \succeq \Gamma, \Delta.$$

Given an ordering $\succ$ on a set Z, we define $\succ_{mul}$, the *multiset extension* of $\succ$, to be the smallest ordering on multisets over Z satisfying $M \cup \{s\} \succ_{mul} M \cup \{t_1, \ldots, t_n\}$ whenever $s \succ t_i$, for all $i \in 1 \ldots n$, with $n \geq 0$. In other words, for obtaining a smaller multiset one (repeatedly) replaces elements by finitely many, possibly zero, smaller elements. If $\succ$ is a well-founded and total ordering on Z, so is $\succ_{mul}$ on finite multisets over Z. This will be used to lift orderings $\succ$ on terms to orderings on equations and clauses. Let C be a ground clause, and let $emul(s \simeq t)$ be $\{s, t\}$ if $s \simeq t$ is a positive equation in C, and $\{s, s, t, t\}$ if it is negative. Then, if $\succ$ is an ordering, we define the ordering $\succ_e$ on (occurrences of) ground equations in a clause by $e \succ_e e'$ if $emul(e) \succ_{mul} emul(e')$. Similarly, $\succ_c$ on ground clauses is defined $C \succ_c D$ if $mse(C) \ (\succ_{mul})_{mul} \ mse(D)$, where $mse(C)$ is the multiset of all $emul(e)$ for occurrences e of equations in C. We see that $C \succ_c D$ whenever (i) the maximal term in C is strictly larger then the maximal term in D, or (ii) the maximal term in C and D is the same, and it occurs negatively in C but only positively in D, or (iii) D results from C by replacing an occurrence of a subterm in some literal by a smaller term, or deleting a literal, and adding any (finite, including zero) number of smaller literals. These are three crucial properties[3] of $\succ_c$ which have the effect that the inferences in $\mathcal{G}$ are *monotone* in that the conclusion is smaller than the maximal (the second, in the case of superposition) premise. In fact, reflexivity resolution deletes a literal. Superposition replaces the maximal term in the second premise by a smaller term and appends side literals (in Γ') which are all smaller than the literal $s \simeq t$ in which the replacement occurs.

Definition 1. Let S be a set of ground Horn clauses. We now define, by induction on $\succ_c$, the ground TRS R_S *generated* by S, as follows. A clause C in S of the form $\Gamma \to l \simeq r$ *generates* the rule $l \Rightarrow r$ if

1. $R_C^* \not\models C$,
2. $l \succ r, \Gamma$, and
3. l is irreducible by R_C

where R_C is the set of rules generated by all clauses D in S such that $C \succ_c D$. We denote by R_S (or simply R when S is clear from the context) the set of rules generated by all clauses of S.

Let us analyze the three conditions. The first one states that a clause only contributes to the model if it does not hold in the partial model built so far and hence we are *forced* to extend this partial model. The second one states that a clause can only generate a rule $l \Rightarrow r$ if l is the strictly maximal term of the clause, and hence $l \succ r$, which makes R a terminating TRS. The third condition, stating that l is irreducible by the rules generated so far, is, together with the second one, the key for showing that R is convergent, which is in turn essential for the completeness result.

Lemma 1. *For every set of ground clauses S, the set of rules R generated for S is convergent (i.e., confluent and terminating). Furthermore, if $R_C^* \models C$ then $R^* \models C$, for all C in S.*

[3] The clause ordering also satisfies certain other properties in addition to (i)—(iii). These are not essential. In fact, to obtain completeness results of certain variants of superposition, different clause orderings, but also satisfying (i)–(iii), are required.

Proof. On the one hand, R terminates since $l \succ r$ for all its rules $l \Rightarrow r$. On the other hand, it suffices to show *local* confluence, which in the ground case is well-known (and easily shown) to hold if there are no two different rules $l \Rightarrow r$ and $l' \Rightarrow r'$ where l' is a subterm of l. This property is fulfilled: clearly when a clause C generates $l \Rightarrow r$, no such $l' \Rightarrow r'$ is in R_C; but if $l' \Rightarrow r'$ is generated by a clause D with $D \succ_c C$ then, by definition of $\succ_c$, we must have $l' \succ l$ and hence l' cannot a be subterm of l either.

To show $R_C^* \models C$ implies $R^* \models C$, let C be $\Gamma \to \Delta$, and assume $R_C^* \models C$. If $R_C^* \models \Delta$ then $R^* \models \Delta$ since $R \supseteq R_C$. Otherwise, $R_C^* \not\models \Gamma$. Then $R^* \not\models \Gamma$ follows from the fact that if a term t occurs negatively in a clause that clause is larger than any clause in which t is also maximal but occurs only positively. As all rules in $R \setminus R_C$ are generated by clauses bigger than C with their maximal term occurring only positively, these clauses generate rules that have left hand sides too big to reduce any term occurring in Γ. Since R is convergent this implies $R^* \not\models \Gamma$. □

Theorem 1. *The inference system $\mathcal{G}$ is refutationally complete for ground Horn clauses.*

Proof. Let S be a set of ground Horn clauses that is closed under $\mathcal{G}$ and such that $\Box \notin S$. We prove that then S is satisfiable by showing that R^* is a model for S. We proceed by induction on $\succ_c$, that is, we derive a contradiction from the existence of a minimal (with respect to $\succ_c$) clause C in S such that $R^* \not\models C$. Clauses false in R^* are called *counterexamples* for the *candidate model R*. Therefore we assume that C is a minimal counterexample (and different from the empty clause). There are a number of cases to be considered, depending on the occurrences in C of its maximal term s, i.e., the term s such that $s \succeq u$ for all terms u in C (s is unique since $\succ$ is total on $T(\mathcal{F})$):

1. s occurs only in the succedent and C is $\Gamma \to s \simeq s$. This is not possible since $R^* \not\models C$.
2. s occurs only in the succedent and C is $\Gamma \to s \simeq t$ with $s \neq t$. Then $s \succ t, \Gamma$. Since $R^* \not\models C$, we have $R^* \supseteq \Gamma$ and $s \simeq t \notin R^*$, i.e., C has not generated the rule $s \Rightarrow t$. From Lemma 1 we infer that $R_C^* \not\models C$ so that the reason why C has not generated a rule must be that s is reducible by some rule $l \Rightarrow r \in R_C$. Suppose that $l \Rightarrow r$ has been generated by a clause C' of the form $\Gamma' \to l \simeq r$. Then there exists an inference by superposition right:

$$\frac{\Gamma' \to l \simeq r \qquad \Gamma \to s \simeq t}{\Gamma', \Gamma \to s[r]_p \simeq t}$$

where the conclusion D has only terms u with $s \succ u$, and hence $C \succ_c D$. Moreover, D is in S and $R^* \not\models D$, since $R^* \supseteq \Gamma \cup \Gamma'$ and $s[r]_p \simeq t \notin R^*$ (since otherwise $s[l]_p \simeq t \in R^*$). This contradicts the minimality of C.

3. s occurs in the antecedent and C is $\Gamma, s \simeq s \to \Delta$. Then there exists an inference by reflexivity resolution:

$$\frac{\Gamma, s \simeq s \to \Delta}{\Gamma \to \Delta}$$

   and for its conclusion D we have $C \succ_c D$. Moreover, D is in S and $R^* \not\models D$, which is a contradiction as in the previous case.
4. s occurs in the antecedent and C is $\Gamma, s \simeq t \to \Delta$ with $s \succ t$. Since $R^* \not\models C$, we have $s \simeq t \in R^*$ and since R is convergent, s and t must have the same normal forms with respect to R, so s must be reducible by some rule $l \Rightarrow r \in R$. Assume $l \Rightarrow r$ has been generated by a clause C' of the form $\Gamma' \to l \simeq r$. Then there exists an inference by superposition left:

$$\frac{\Gamma' \to l \simeq r \qquad \Gamma, s[l]_p \simeq t \to \Delta}{\Gamma', \Gamma, s[r]_p \simeq t \to \Delta}$$

   and for its conclusion D we have that $C \succ_c D$. Moreover, D is in S and $R^* \not\models D$, which again contradicts the minimality of C. $\square$

Example 1. In the figure 1 we illustrate how the rewrite system R changes during the process of saturating a set of ground clauses under $\mathcal{G}$. None of the intermediate interpretations is a model. In the end we obtain one, as stated by the theorem.

Consider the lexicographic path ordering generated by the precedence $f > a > b > c > d$. The following table shows in the left column the ground Horn clauses (sorted in ascending order) at each step of the closure process. The first set is the initial clause set, the two other sets are obtained by adding an inference with the minimal counterexample of the preceding set of clauses. In the right column the rewrite systems as generated by the respective clause sets are displayed. The maximal term of every clause is underlined and the subterms of the clauses involved in the inference leading to the next clause set are framed.

4.4.3 Selection of Negative Equations

The inference system $\mathcal{G}$ includes strong ordering restrictions: a superposition inference is needed only if the term involved is maximal in the clause, and even strictly maximal for terms of positive equations. But more constraints can be imposed. If a clause C has a non-empty antecedent, it is possible to arbitrarily *select* exactly one of its negative equations. Then completeness is preserved if C is not used as left premise of any superposition inference and the only inferences involving C are equality resolution or superposition left on its selected equation.

S	R
$\begin{array}{rl} & \to \boxed{\underline{c}} \simeq d \\ \underline{f(d)} \simeq d & \to a \simeq b \\ & \to \underline{f(\boxed{c})} \simeq d \end{array}$	$c \simeq d$
$\begin{array}{rl} & \to \underline{c} \simeq d \\ & \to \boxed{\underline{f(d)}} \simeq d \\ \boxed{\underline{f(d)}} \simeq d & \to a \simeq b \\ & \to \underline{f(c)} \simeq d \end{array}$	$\begin{array}{rcl} c & \simeq & d \\ f(d) & \simeq & d \end{array}$
$\begin{array}{rl} & \to \underline{c} \simeq d \\ d \simeq d & \to \underline{a} \simeq b \\ & \to \underline{f(d)} \simeq d \\ \underline{f(d)} \simeq d & \to a \simeq b \\ & \to \underline{f(c)} \simeq d \end{array}$	$\begin{array}{rcl} c & \simeq & d \\ a & \simeq & b \\ f(d) & \simeq & d \end{array}$

Fig. 4.1. Evolution of the rewrite system during saturation

In the following variant of the inference system $\mathcal{G}$, selected equations are underlined, and it is assumed that any clause $\Gamma \to \Delta$ where the maximal term occurs in Γ has one selected negative equation.

superposition right:

$$\frac{\Gamma' \to l \simeq r \qquad \Gamma \to s \simeq t}{\Gamma', \Gamma \to s[r]_p \simeq t} \quad \text{if} \quad \begin{array}{l} s|_p = l,\ l \succ r, \Gamma',\ s \succ t, \Gamma, \text{ and} \\ \text{no equation is selected in } \Gamma \text{ or } \Gamma'. \end{array}$$

superposition left:

$$\frac{\Gamma' \to l \simeq r \qquad \Gamma, \underline{s \simeq t} \to \Delta}{\Gamma', \Gamma, s[r]_p \simeq t \to \Delta} \quad \text{if} \quad \begin{array}{l} s|_p = l,\ l \succ r, \Gamma',\ s \succ t, \text{ and} \\ \text{no equation is selected in } \Gamma'. \end{array}$$

reflexivity resolution:

$$\frac{\Gamma, \underline{s \simeq s} \to \Delta}{\Gamma \to \Delta}$$

Exercise: adapt the proof of completeness of Theorem 1 to this framework with selection (hint: consider that clauses with selected equations generate no rules).

4.4.4 Clauses with Variables

Up to now, in this section we have only dealt with ground clauses. Since a non-ground clause represents the set of all its ground instances, a refutationally complete method for the non-ground case would be to systematically enumerate all ground instances of the clauses, and to perform inferences by $\mathcal{G}$ between those instances. But fortunately it is possible to perform inferences between non-ground clauses, covering in one step a possibly large number of ground inferences. We now adapt $\mathcal{G}$ according to this view. For example, at the ground level, in the superposition right inference

$$\frac{\Gamma' \rightarrow l \simeq r \qquad \Gamma \rightarrow s \simeq t}{\Gamma', \Gamma \rightarrow s[r]_p \simeq t} \quad \text{if} \quad \begin{array}{l} s|_p = l \text{ and} \\ l \succ r, \Gamma' \text{ and } s \succ t, \Gamma. \end{array}$$

we have required $s|_p$ and l to be the same term. At the non-ground level, this becomes a constraint $s|_p \doteq l$ on the possible instances of the conclusion. Therefore we may view the conclusion of a non-ground inference as a constrained clause $D \;\mid\; T$, representing, semantically, the set of ground instances $D\sigma$ for which the correspondingly instantiated constraint $T\sigma$ is true.

Hence if the conclusion is of the form $D \;\mid\; s|_p \doteq l \wedge \gamma$, the instances $D\sigma$ of D for which $s|_p\sigma \neq l\sigma$ are excluded, and similarly for ordering restrictions. For instance, instead of requiring $l \succ r$ as a meta-level condition of the inference, it becomes part of the constraint of the conclusion, excluding those instances of the conclusion that correspond to ground inferences between instances of the premises for which $l \succ r$ does not hold:

$$\frac{\Gamma' \rightarrow l \simeq r \qquad\qquad \Gamma \rightarrow s \simeq t}{\Gamma', \Gamma \rightarrow s[r]_p \simeq t \quad \mid \; s|_p \doteq l \;\wedge\; l > r, \Gamma' \;\wedge\; s > t, \Gamma}$$

Therefore in our context, a *constraint* is a boolean combination of atomic constraints of the form $s \doteq t$ (an *equality constraint*) or $s > t$ or $s \geq t$ (*inequality constraints*). A substitution σ is called a *solution* of a constraint T if the instantiated constraint $T\sigma$ is ground and evaluates to true when interpreting $\doteq$ and $>$ respectively, as the syntactic equality and $\succ$ on ground terms. We shall also write $\sigma \models T$ whenever σ is a solution of T. Constraints such as $l > r, \Gamma$ are considered to be abbreviations for an obvious conjunction of inequality constraints involving the terms in Γ, l and r. A ground clause $D\sigma$ is called a *ground instance* of a constrained clause $D \;\mid\; T$ whenever σ is a solution of T. If C is the empty clause and T is satisfiable, the empty (unconstrained) clause is an instance of C. If T is unsatisfiable, a clause constrained by T does not have any instances and, hence, can be considered a tautology.

As in the case of the critical pair inference for unit equations, the inference rule may be equipped with the additional restriction excluding the cases where $s|_p$ is a variable. This indicates that certain inferences between ground instances of the premises are redundant, namely those for which the replacement of equals by equals occurs *inside* the substitution part. Such inferences are also called inferences *below variables*), as they occur at positions $s\sigma|_p$ where $s|_{p'}$ is a variable for some prefix p' of p.

The full inference system for non-ground Horn clauses will be called $\mathcal{H}$ (for Horn), where we again assume that variables in premises are renamed to achieve variable disjointness:

superposition right:

$$\frac{\Gamma' \to l \simeq r \qquad \Gamma \to s \simeq t}{\Gamma', \Gamma \to s[r]_p \simeq t \ \mid\ s|_p \doteq l \ \wedge\ l > r, \Gamma' \ \wedge\ s > t, \Gamma} \quad \text{if } s|_p \text{ is not a variable, and nothing is selected in } \Gamma', \Gamma$$

superposition left:

$$\frac{\Gamma' \to l \simeq r \qquad \Gamma, \underline{s \simeq t} \to \Delta}{\Gamma', \Gamma, s[r]_p \simeq t \to \Delta \ \mid\ s|_p \doteq l \ \wedge\ l > r, \Gamma' \ \wedge\ s > t} \quad \text{if } s|_p \text{ is not a variable, and nothing is selected in } \Gamma'$$

reflexivity resolution:

$$\frac{\Gamma, \underline{s \simeq t} \to \Delta}{\Gamma \to \Delta \quad \mid\ s \doteq t}$$

Example 2. Consider the lexicographic path ordering generated by the precedence $h > a > f > g > b$ and the inference

$$\frac{g(x) \simeq x \to f(a,x) \simeq f(x,x) \quad \to h(f(a,g(y))) \simeq h(y)}{g(x) \simeq x \to h(f(x,x)) \simeq h(y) \ \mid\ f(a,x) \doteq f(a,g(y)) \wedge h(f(a,g(y))) > h(y) \wedge f(a,x) > f(x,x), g(x), x}$$

The constraint of the conclusion is satisfiable: using the properties of the ordering and solving the unification problem, the constraint can be simplified to

$$x \doteq g(y) \ \wedge\ a > x$$

which is satisfied, for instance, by the substitution $\{y \mapsto b, x \mapsto g(b)\}$.

On the other hand, the following inference is not needed

$$\frac{\to f(x,x) \simeq f(a,x) \qquad \to f(g(y),z) \simeq h(z)}{\to f(a,x) \simeq h(z) \quad \mid\ f(x,x) \doteq f(g(y),z) \ \wedge\ f(g(y),z) > h(z) \wedge f(x,x) > f(a,x)}$$

since the constraint of its conclusion is unsatisfiable: it can be simplified to

$$x \doteq g(y) \ \wedge\ x \doteq z \ \wedge\ y \geq h(z) \ \wedge\ x > a$$

which implies $y \geq h(g(y))$ and therefore it is unsatisfiable. Note that if the equality constraint and the ordering constraint are considered independently then both are satisfiable.

4.4.5 Completeness without Constraint Inheritance

There are several possible treatments for constrained clauses as generated by the inference system $\mathcal{H}$. The classical view is to make them unconstrained again. That is, clauses of the form $C \mid s \doteq t \wedge OC$, for some ordering constraint OC, are immediately converted into $C\sigma$ where $\sigma = mgu(s,t)$. This will be called the strategy *without constraint inheritance*. Below we will also explain an alternative where constraints are kept and inherited throughout inferences.

Of course, the clause $C\sigma$ has to be generated only if the constraint $s \doteq t \wedge OC$ is satisfiable in $T(\mathcal{F})$. If $\succ$ is the lexicographic path ordering (LPO) the satisfiability of constraints is decidable [Com90, Nie93]. But traditionally in the literature weaker approximations by non-global tests are used; for example, inference systems are sometimes expressed with local conditions of the form $r \not\succeq l$ when in our framework we have $l > r$ as a part of the global constraint OC. Note that such weaker approximations do not lead to unsoundness, but only to the generation of unnecessary clauses.

In the following, we call a set of (unconstrained) Horn clauses S *closed under* $\mathcal{H}$ *without constraint inheritance* if $D\sigma \in S$ for all inferences by $\mathcal{H}$ with premises in S and conclusion $D \mid s \doteq t \wedge OC$ such that $s \doteq t \wedge OC$ is satisfiable and $\sigma = mgu(s,t)$.

Theorem 2. *The inference system $\mathcal{H}$ without constraint inheritance is refutationally complete for Horn clauses.*

Proof. Let S be a set of Horn clauses closed under $\mathcal{H}$ without constraint inheritance such that $\square \notin S$. The proof is very similar to the one for $\mathcal{G}$: we exhibit a model R^* for S. We proceed again by induction on $\succ_c$, but now the rôle of the ground clauses in the proof for $\mathcal{G}$ is played by all *ground instances* of clauses in S, and the generation of rules in R from these ground instances is as for $\mathcal{G}$. Now we derive a contradiction from the existence of a minimal (with respect to $\succ_c$) *ground instance* $C\sigma$ *of a* clause C in S such that $R^* \not\models C\sigma$. The cases considered are the same ones as well, again depending on the occurrences in $C\sigma$ of its maximal term $s\sigma$.

The only additional complication lies in the *lifting* argument. As all the cases are essentially similar, we will only treat one of them in detail. Suppose that $C = \Gamma, \underline{s \simeq t} \rightarrow \Delta$ and $s\sigma \succ t\sigma$. Since $R^* \not\models C\sigma$, we have $s\sigma \simeq t\sigma \in R^*$ and since R is convergent, $s\sigma$ must be reducible by some rule $l\sigma \Rightarrow r\sigma \in R$, generated by a clause C' of the form $\Gamma' \rightarrow l \simeq r$. Now we have $s\sigma|_p = l\sigma$, and there are two possibilities:

An inference. $s|_p$ is a non-variable position of s.
Then there exists an inference by superposition left:

$$\frac{\Gamma' \rightarrow l \simeq r \qquad \Gamma, \underline{s \simeq t} \rightarrow \Delta}{\Gamma', \Gamma, s[r]_p \simeq t \rightarrow \Delta \quad \mid \; s|_p \doteq l \;\wedge\; l > r, \Gamma' \;\wedge\; s > t}$$

where the conclusion $D \mid T$ has an instance $D\sigma$ (i.e., σ satisfies the constraint T) such that $C\sigma \succ_c D\sigma$, and $R^* \not\models D\sigma$, contradicting the minimality of $C\sigma$.

Lifting. $s|_{p'}$ is a variable x for some prefix p' of p.
Then $p = p' \cdot p''$ and $x\sigma|_{p''}$ is $l\sigma$. Now let σ' be the ground substitution with the same domain as σ but where $x\sigma' = x\sigma[r\sigma]_{p''}$ and $y\sigma' = y\sigma$ for all other variables y. Then $R^* \not\models C\sigma'$ and $C\sigma \succ_c C\sigma'$, contradicting the minimality of $C\sigma$. □Notice that the lifting argument relies on the existence of the substitution-reduced ground instance $C\sigma'$ of C. In inference systems without hereditary constraints, clauses in saturated clause sets carry no constraint. Therefore any grounding substitution yields an instance of C. For clauses with constraints the situation can be different. Reducing a solution σ of a constraint by a rewrite system might yield a substitution that is not a solution anymore, so that $C\sigma'$ is not a ground instance of C that could be a counterexample.

4.4.6 General Clauses

Now we consider clauses that may have more than one equation in their succedent. For this we need the notions of *maximal* and *strictly maximal* equation in a clause C. To express maximality and strict maximality we use $gr(s \simeq t, \Delta)$ and $geq(s \simeq t, \Delta)$, which denote, respectively, that the equation $s \simeq t$, considered as a multiset of terms, is strictly greater (with respect to the multiset extension of $\succ$) than and greater than or equal to all equations in a multiset of equations Δ.

The inference system $\mathcal{I}$ for general clauses consists of these inference rules:

superposition right:

$$\frac{\Gamma' \to l \simeq r, \Delta' \qquad \Gamma \to s \simeq t, \Delta}{\Gamma', \Gamma \to s[r]_p \simeq t, \Delta', \Delta \quad \mid \; s|_p \doteq l \;\wedge\; l > r, \Gamma' \;\wedge\; s > t, \Gamma \;\wedge\; gr(l \simeq r, \Delta') \;\wedge\; gr(s \simeq t, \Delta)}$$

superposition left:

$$\frac{\Gamma' \to l \simeq r, \Delta' \qquad \Gamma, \underline{s \simeq t} \to \Delta}{\Gamma', \Gamma, s[r]_p \simeq t \to \Delta', \Delta \quad \mid \; s|_p \doteq l \;\wedge\; l > r, \Gamma' \;\wedge\; s > t \;\wedge\; gr(l \simeq r, \Delta')}$$

reflexivity resolution:

$$\frac{\Gamma, \underline{s \simeq t} \to \Delta}{\Gamma \to \Delta \quad \mid \; s \doteq t}$$

equality factoring:

$$\frac{\Gamma \to s \simeq t, s' \simeq t', \Delta}{\Gamma, t \simeq t' \to s \simeq t', \Delta \;\mid\; s \doteq s' \;\wedge\; s > t, \Gamma \;\wedge\; geq(s \simeq t, (s' \simeq t', \Delta))}$$

where as in the Horn case in both superposition rules $s|_p$ is not a variable and where premises do not have selected negative equations except the ones that are explicitly indicated by the inferences.

Example 3. Consider the lexicographic path ordering generated by the precedence $f > g > h$ and the following inference by superposition right

$$\frac{\to g(z) \simeq h(z) \qquad\qquad \to f(g(x),y) \simeq g(x), f(g(x),y) \simeq y}{\to f(h(z),y) \simeq g(x), f(g(x),y) \simeq y \ \mid\ \begin{array}{l} g(x) \doteq g(z) \ \wedge\ g(z) > h(z) \wedge \\ f(g(x),y) > g(x) \wedge \\ gr(f(g(x),y) \simeq g(x), \{f(g(x),y) \simeq y\}) \end{array}}$$

where $gr(f(g(x),y) \simeq g(x), \{f(g(x),y) \simeq y\})$ represents the ordering constraint $\{f(g(x),y), g(x)\} >_{mul} \{f(g(x),y), y\}$, which can be simplified into $g(x) > y$. Now, simplifying the rest of the constraint the conclusion of the inference can be written as

$$\to f(h(z),y) \simeq g(x), f(g(x),y) \simeq y \quad \mid \quad x \doteq z \ \wedge\ g(x) > y$$

Theorem 3. *The inference system $\mathcal{I}$ is refutationally complete for general clauses.*

The proof is an extension of the proof of Theorem 2. One way to do it is to extend the construction of candidate models in Definition 1 appropriately. Let a ground clause $C = \Gamma \to l \simeq r, \Delta$ generate $l \simeq r$ if, and only if, (i) C is false in the partial interpretation R_C^*, if $l \succ r, \Gamma$, if $geq(l \simeq r, \Delta)$ holds, if l is irreducible by R_C^*, and (ii) no equation in Δ is true in $(R_C \cup \{l \simeq r\})^*$. Then the reasoning about reducing counterexamples is essentially similar. Because of condition (ii) preventing the production of certain rules, an additional form of counterexamples—those satisfying (i) but not (ii)—may occur. Counterexamples of this form can be reduced by equality factoring inferences.

Exercise: Prove the theorem on the basis of these hints.

A consequence of this theorem is compactness. In fact, the theorem makes no assumptions about the cardinality of S. If S is infinite and unsatisfiable, we may consider its closure $S_\infty = \bigcup_{i\geq 1} S^i$ under $\mathcal{I}$, with $S^0 = S$, and $S^{i+1} = S^i \cup \mathcal{I}(S)$, for $i \geq 0$. Then S_∞ is also unsatisfiable hence, by the theorem, contains the empty clause, which must then be an element in one of the S^i. Therefore, the empty clause has a finite superposition proof involving only finitely many clauses from S as initial premises.

4.4.7 Ordered Resolution with Selection

As pointed out in the section 4.2.7, one may view non-equational atoms of the form $p(s_1, \ldots, s_n)$, with p a predicate symbol, as equations of the form $p(s_1, \dots, s_n) \simeq \top$. Ordered resolution is a specialization of superposition to *non-equational clauses*, that is, clauses in which all equations are encodings of non-equational atoms. In that case, superposition inferences are only possible at the top of terms (that encode an atom). Also, superposition, right, generates a positive equation $\top \simeq \top$ so that its conclusion is a tautology, hence redundant.[4] Similarly, superposition, left, produces a negative equation $\top \simeq \top$ which, when selected, can be eliminated by reflexivity resolution. Therefore reflexivity resolution inferences will be built into the resolution inference. Equality factoring specializes to factoring. Moreover, ordering constraints can be greatly simplified. In particular a constraint such as $gr(p(s_1, \dots, s_n) \simeq \top, \Delta)$ simplifies to $p(s_1, \dots, s_n) \succ \Delta$, assuming, as we do, that $\bot$ is minimal in the ordering. Also, $geq(p(s_1, \dots, s_n) \simeq \top, \Delta)$ is true if and only if $p(s_1, \dots, s_n)$ is the maximal atom with respect to Δ, justifying the simplified notation $geq(p(s_1, \dots, s_n), \Delta)$.

The inference system $\mathcal{R}$ of ordered resolution with selection for non-equational clauses consists of these inference rules, where letters A and B denote non-equational atoms:

resolution:

$$\frac{\Gamma' \rightarrow A, \Delta' \qquad \Gamma, \underline{B} \rightarrow \Delta}{\Gamma', \Gamma \rightarrow \Delta', \Delta \qquad \mid\ A \doteq B \ \wedge\ A > \Gamma', \Delta'}$$

factoring:

$$\frac{\Gamma \rightarrow A, B, \Delta}{\Gamma \rightarrow A, \Delta \quad \mid\ A \doteq B \ \wedge\ A > \Gamma \ \wedge\ geq(A, \Delta)}$$

where, as in the case of $\mathcal{I}$, premises do not have selected negative atoms except for the ones that are explicitly indicated in the inferences.

From what we have said before, the refutational completeness of ordered resolution with selection is a special case of Theorem 3.

Theorem 4. *The inference system $\mathcal{R}$ is refutationally complete for general non-equational clauses.*

Note that since in resolution there is no rewriting of proper subterms, the requirements about the ordering can be relaxed. All that is needed is that the ordering is total and well-founded[5] on ground atoms $p(s_1, \dots, s_n)$. The non-atomic terms s_i need not be ordered.

[4] Tautologies can neither be productive nor minimal counterexamples. Therefore they do not participate in essential inferences. For a general notion of redundancy where tautologies appear as a special case we refer to the next section.

[5] Even well-foundedness is not essential as, because of compactness, only finite sets of ground atoms participate in a refutation.

4.5 Saturation Procedures

Simplification and deletion techniques that are compatible with refutational completeness are well-known to be essential for obtaining useful implementations of paramodulation-based theorem proving. These techniques can be uniformly covered by notions of *redundancy* for inferences and clauses. On the one hand, instead of closing the initial set of clauses under a given inference system, we only need to *saturate* it, that is, close it up to *redundant inferences*. Similarly, *redundant clauses* are unnecessary as well, and can be removed during the processes that compute saturated sets, called *saturation* procedures. In this setting, roughly speaking, a clause is redundant if all its instances are logical consequences of other clauses that are smaller. An inference is redundant if, for all its instances, either one of its premises is a strictly redundant clause or else its conclusion follows from clauses smaller than the maximal premise.

Here we only describe in detail the computation of saturated sets for the case of ground clauses. Hence in the remainder of this section all clauses, denoted by $C, D, \ldots$, and sets of clauses, denoted by S, are assumed to be ground. The different notions explained here smoothly extend to the non-ground case (see [BG98, NR01]).

Let S be a set of ground clauses. We denote by $S^{\prec C}$ the set of all D in S such that $C \succ_c D$. A clause C is called *redundant* (in S) if $S^{\prec C} \models C$. An inference with premises $C_1, \ldots, C_n$, maximal premise C and conclusion D is *redundant* (in S) if either one of its premises is redundant in S, or $S^{\prec C} \models D$. S is *saturated* with respect to an inference system $\mathcal{I}nf$ if every inference of $\mathcal{I}nf$ from premises in S is redundant in S.

Theorem 5. *Let S be a set of ground clauses that is saturated with respect to $\mathcal{I}$. Then $R^* \models S$ if, and only if, $\Box \notin S$, and hence S is unsatisfiable if, and only if, $\Box \in S$.*

For the proof of this theorem, one inspects the proof of Theorem 1 to find that inferences essential for reducing counterexamples to the candidate model R for a saturated set S of ground clauses involve rather specific kinds of premises. The maximal premise of such an inference is the (hypothetical) minimal counterexample D, and the side premise (if any) is a productive clause C. The minimal counterexample cannot be implied by smaller (hence true in R^*) clauses in S. For similar reasons, productive clauses C smaller than the minimal counterexample cannot be redundant either. (Otherwise they would already be true in the partial interpretation R^*_C.) Also, as the inference is monotone, the conclusion is a clause smaller than the maximal premise. Therefore, if the inference is redundant the conclusion is a counterexample smaller than D, but is also implied by clauses smaller than D, which together is impossible. Hence if S is saturated up to redundancy, R is a model of S, or else the minimal counterexample D in S must be the empty clause.

Exercise: Fill in the details of this proof sketch.

The previous theorem states that, instead of computing sets closed under the inference system, it suffices to saturate them. Therefore, we may now describe methods for computing saturated sets which are more practical than naive procedures which simply compute all inferences from existing clauses. These methods are abstractly formalized by the notion of a (theorem proving) *derivation* which are sequences of sets of clauses where each set is obtained by either adding some logical consequence to, or by removing some *redundant* clause from, the previous set in the sequence.

A *derivation* is a (possibly infinite) sequence of sets of clauses $S_0, S_1, \ldots$ where each S_{i+1} is either $S_i \cup S$, for some set of clauses S such that $S_i \models S$, or else S_{i+1} is $S_i \setminus S$, for some set of clauses S which are all redundant in S_i. Clauses belonging, from some k on, to all S_i with $i > k$ are called *persistent:* the set $S_\infty = \cup_k \cap_{i>k} S_i$ is the set of all persistent clauses.

A large number of different practical redundancy elimination methods that are applied in theorem provers fit into the notion of derivation. For example, simplifying $P(f(a))$ into $P(a)$ with the equation $f(a) \simeq a$, is modeled by first adding $P(a)$, and then removing $P(f(a))$ which has become redundant in the presence of $P(a)$ and $f(a) \simeq a$.

However, if our aim is to obtain refutationally complete theorem proving procedures by computing derivations, and in the limit, to obtain a saturated set, then a notion of *fairness* is required. A derivation $S_0, S_1, \ldots$ is *fair* with respect to an inference system $\mathcal{I}nf$ if every inference of $\mathcal{I}nf$ with premises in S_∞ is redundant in S_j for some j.

Now what can be done in practice to ensure fairness? On the one hand, for the inference system $\mathcal{I}$, if an inference is not redundant, it can *be made* redundant by adding its conclusion, since in $\mathcal{I}$ the conclusion always is smaller than the maximal premise. On the other hand, this has to be done only for inferences with persistent premises. Since it is not possible to know, at a certain point S_i, whether a given premise is going to be persisent or not, some means to ensure that no non-redundant inference is eternally postponed must be provided. In practice this is usually done by means of size-based priority queues.

Theorem 6. *If $S_0, S_1, \ldots$ is a fair derivation with respect to $\mathcal{I}nf$, then $\cup_i S_i$ is saturated with respect to $\mathcal{I}nf$, and hence if $\mathcal{I}nf$ is $\mathcal{I}$, then S_0 is unsatisfiable if, and only if, $\square \in S_j$ for some j. Furthermore, if $S_0, S_1, \ldots, S_n$ is a fair derivation then S_n is saturated and logically equivalent to S_0.*

For the proof of this theorem it is essential to observe that redundancy of a clause or inference is preserved under adding clauses and removing redundant clauses.

4.5.1 Computing with Saturated Sets

In practice, indeed it is sometimes possible to obtain a finite saturated set S_n (not containing the empty clause) for a given input S_0. In this case hence its satisfiability has been proved. This kind of consistency proving has of course many applications and is also closely related to inductive theorem proving.

Theorem proving in theories expressed by saturated sets S of axioms is also interesting because more efficient proof strategies become (refutationally) complete. For example, if S and S' are two sets of clauses where S is saturated, then, when saturating $S \cup S'$, one can exploit the fact that inferences between premises of S are redundant. Note that such a *set-of-support* strategy is incomplete in general for ordered inference systems and also for paramodulation ([SL91] describe a lazy paramodulation calculus that is complete with set of support).

In some cases, depending on the syntactic properties of the given finite saturated set S, decision procedures for the theory are obtained. This is the case for instance for saturated sets of equations E, i.e., where saturation is in fact (unfailing) Knuth-Bendix completion, and where rewriting with E decides the word problem:

Theorem 7. *Let E be a consistent saturated (with respect to $\mathcal{H}$ and $\succ$) set of equations. Then, for all terms s and t, $E \models s \simeq t$ iff $t \Rightarrow_E \circ \Leftarrow_E s$.*

Decision procedures are also obtained for entailment problems $S \models G$, where S is saturated under $\mathcal{I}$ and G is a variable-free first-order formula, provided the ordering is such that for any ground term there are only finitely many smaller ground terms. In fact, to refute $S \cup \neg G$, only inferences in which one premise is ground (and initially from the set of clauses obtained from clausifying $\neg G$) can possibly be non-redundant. Any such inference produces a conclusion with an ordering constraint that bounds each term in the conclusion from above by a ground term in G. Therefore, clauses derived from non-redundant inferences have only finitely many ground instances where, in addition, all terms are smaller than or equal to a term in G. By a straightforward induction argument and by using König's Lemma, saturation will eventually terminate.

In particular if S is empty (hence trivially saturated), and if G is just a single ground Horn clause $s_1 \simeq t_1, \ldots, s_k \simeq t_k \to s \simeq t$ we obtain the so-called congruence closure problem of deciding whether or not $s_1 \simeq t_1, \ldots, s_k \simeq t_k \models s \simeq t$ is true in FOLE. For this special case one may choose an arbitrary reduction ordering for (finitely) saturating $\{s_1 \simeq t_1, \ldots, s_k \simeq t_k\} \cup \{s \simeq t \to \}$, provided one performs simplification by rewriting eagerly. In this special case, any superposition inference yields a conclusion that renders the maximal premise of the inference redundant. By introducing new constants for any of the subterms in G and by organizing rewrite rules into a union-find data structure, this saturation process can be organized in time $O(n \log n)$ [Kap97, BT00]

Other syntactic restrictions on non-equational saturated sets S that are quite easily shown to lead to decision procedures include *reductive* Horn clauses (also called conditional equations) or *universally reductive* general clauses [BG94a], even in cases where the reduction ordering is not isomorphic to ω. This applies to certain restricted kinds of queries only.

We will expand further on the subject of deciding entailment problems in the section 4.8 for the case of non-equational first-order theories.

4.6 Paramodulation with Constrained Clauses

In this section the power of using constrained clauses is further exploited. By keeping the equality constraints with the conclusion of an inference without solving them we can avoid further inferences on subterms of this conclusion which come from the unification problem. By keeping the ordering constraints we obtain a single constrained clause representing the exact set of ground instances of the conclusion that fulfill the ordering restrictions of the inference. Hence working with constrained clauses and inheriting these constraints in the inferences allows us to avoid many inferences and hence to further restrict the search space. To avoid certain technical complications, in this section we only consider Horn clauses. For the general case we refer to the overview articles [BG98] and [NR01].

4.6.1 Equality Constraint Inheritance: Basic Strategies

We now analyse the first constraint-based restriction of the search space: the so-called *basicness restriction*, where superposition inferences on subterms created by unifiers on ancestor clauses are not performed. This restriction can be conveniently expressed by inheriting the equality constraints without applying (or even computing) any unifiers. What is required is that satisfiability of certain equality constraints is checked. In fact, whenever a constrained clause with an empty skeleton is derived, only if the constraint is satisfiable have we found a contradiction.

In the following, we call a set of Horn clauses S with equality constraints *closed under $\mathcal{H}$ with equality constraint inheritance* if $D \mid T_1 \wedge \ldots \wedge T_n \wedge s \doteq t$ is in S whenever $C_1 \mid T_1$, ... , $C_n \mid T_n$ are clauses in S and there is an inference by $\mathcal{H}$ with premises $C_1, \ldots, C_n$ and conclusion $D \mid s \doteq t \wedge OC$ and $T_1 \wedge \ldots \wedge T_n \wedge s \doteq t \wedge OC$ is satisfiable. In this setting, the ordering constraints OC associated with inferences are only checked for satisfiability but not inherited themselves.

This strategy is incomplete in general:

Example 4. Let $\succ$ be the lexicographic path ordering where $a \succ_{\mathcal{F}} b$. Consider the following unsatisfiable clause set S:

$$\begin{array}{ll} 1. & \rightarrow a \simeq b \\ 2. & \rightarrow P(x) \mid x \doteq a \\ 3. & P(b) \rightarrow \end{array}$$

S is clearly closed under $\mathcal{H}$ with equality constraint inheritance, since no inferences by $\mathcal{H}$ that are compatible with the constraint of the second clause can be made. We have $a \succ_{\mathcal{F}} b$ and hence the first clause could only be used by superposing a on some non-variable subterm, while superposition left (i.e., resolution) between 2 and 3 leads to a clause with an unsatisfiable constraint $x \doteq a \wedge b \doteq x$. However, S does not contain the empty clause. This incompleteness is due to the fact that the usual lifting arguments, like the ones in Theorem 2, do not work here, since they are based on the existence of *all* ground instances of the clauses. Note that this is not the case here: the only instance of the second clause is $P(a)$, whereas the lifting argument in Theorem 2 requires the existence of the instance $P(b)$. $\square$

Fortunately, the strategy is complete for *well-constrained* sets of clauses, which turn out to be adequate for many practical situations. We do not want to define this concept in detail. An important special case is when initial clauses do not carry any constraints, or when clauses are non-equational.

Theorem 8. *Let S be the closure under $\mathcal{H}$ with equality constraint inheritance of a well-constrained set S_0 of Horn clauses. Then either S_0 is satisfiable, or else S contains the empty clause (possibly as an instance of a constrained clause).*

We provide some hints about the proof for the case that clauses in S_0 carry no constraints so that contraints are only generated for clauses produced by inferences in $\mathcal{H}$. The main ideas about constructing candidate models and reducing counterexamples is the same as before. The only modification of the previous completeness proofs is with regard to the ground instances of S from which the rewrite system R is constructed. This time we construct R from the set GS of all ground instances $C\sigma$ of clauses $C \mid T$ in S which are *irreducible at substitution positions*. That means that if p is a position in C for which $C|_p$ is a variable x then $x\sigma$ is irreducible by all rules in $R_{C\sigma}$ (produced by irreducible instances of S smaller than $C\sigma$ in the clause ordering). (Technically this requires to define R and GS simultaneously by induction over the clause ordering.)

Now, clearly, if $C\sigma$ is reducible by a rule in $R_{C\sigma}$, the matching must take place at a non-variable position in the *skeleton* C of the constrained clause $C \mid T$. Ground inferences of $\mathcal{H}$ taking place at such positions in $C\sigma$ can easily be lifted to $C \mid T$. Therefore, if S is saturated under $\mathcal{H}$ with equality constraint inheritance, then GS is saturated under $\mathcal{H}$ without equality constraint inheritance. To see this, one also has to show that ground instances of clauses produced by an inference in $\mathcal{H}$ from premises in GS again belong to GS, that is, are themselves irreducible at substitution positions. This, however, is straightforward for Horn clauses.[6]

Now, by applying Theorem 2, we have shown that either GS is satisfiable or contains the empty clause. Moreover, if GS is satisfiable, then R^* is a model of GS. It remains to be shown that then R^* is also a model of S_0. S contains S_0 and S_0 has no constraints. If $C\sigma$ is ground instance of a clause C in S_0 consider the reduced instance $C\sigma'$ of C where $\sigma'(x)$ is the normal form of $\sigma(x)$ with respect to R. Obviously, $C\sigma'$ is in GS, and therefore true in R^*. As $C\sigma$ can be obtained back from $C\sigma'$ by replacing terms by congruent (with respect to R^*) terms, $C\sigma$ is also true in R^*.

This last step of showing that R^* is not only a model of GS, but also a model of S_0, fails, as shown in the example above, if clauses in S_0 may carry arbitrary constraints. The reduced substitution σ' might not be a solution of the constraint attached to C.

The proof for the case in which S_0 contains only non-equational clauses is even simpler. As there are no equations between non-atomic terms and no predicate variables that could be instantiated with atom terms, any ground instance of a constrained clause is irreducible.

4.6.2 Ordering Constraint Inheritance

The ideas on equality constraint inheritance apply as well to ordering constraint inheritance or to a combination of both:

A set of constrained Horn clauses S *closed under $\mathcal{H}$ with ordering constraint inheritance* if $D\sigma \;\mid\; T_1 \wedge \ldots \wedge T_n \wedge OC$ is in S whenever $C_1 \;\mid\; T_1\;, \;\ldots\;, \;C_n \;\mid\; T_n$ are clauses in S and there is an inference by $\mathcal{H}$ with (unconstrained) premises $C_1, \ldots, C_n$ and conclusion $D \;\mid\; s \doteq t \wedge OC$, $T_1 \wedge \ldots \wedge T_n \wedge s \doteq t \wedge OC$ is satisfiable, and $\sigma = mgu(s,t)$.

A set of constrained Horn clauses S *closed under $\mathcal{H}$ with equality and ordering constraint inheritance* if $D \;\mid\; T_1 \wedge \ldots \wedge T_n \wedge s \doteq t \wedge OC$ is in S whenever $C_1 \;\mid\; T_1\;, \;\ldots\;, \;C_n \;\mid\; T_n$ are clauses in S and there is an inference by $\mathcal{H}$ with premises $C_1, \ldots, C_n$ and conclusion $D \;\mid\; s \doteq t \wedge OC$ and $T_1 \wedge \ldots \wedge T_n \wedge s \doteq t \wedge OC$ is satisfiable.

The following theorems are proved as for the case of equality constraint inheritance:

Theorem 9. *Let S be the closure under $\mathcal{H}$ with ordering constraint inheritance of a well-constrained set S_0 of Horn clauses. Then either S_0 is satisfiable, or else S contains the empty clause (possibly as an instance of a constrained clause).*

[6] Here is where the case of general clauses becomes technically involved.

Theorem 10. *Let S be the closure under $\mathcal{H}$ with equality and ordering constraint inheritance of a well-constrained set S_0 of Horn clauses. Then either S_0 is satisfiable, or else S contains the empty clause (possibly as an instance of a constrained clause).*

4.6.3 Basic Paramodulation

It is possible to further restrict the inference system $\mathcal{H}$ with constraint inheritance, at the expense of weakening the ordering restrictions. Roughly, the improvement comes from the possibility of moving the inserted right hand side in conclusions to the constraint part, thus *blocking* this term for further inferences. On the other hand, paramodulations take place only *with* the maximal term, like in superposition, but *on* both sides of the equation containing the maximal term. More precisely, the inference rule of (ordered, basic) paramodulation right then becomes:

ordered paramodulation right:

$$\frac{\Gamma' \rightarrow l \simeq r \qquad \Gamma \rightarrow s \simeq t}{\Gamma', \Gamma \rightarrow s[x]_p \simeq t \quad \mid \; x \doteq r \;\wedge\; s|_p \doteq l \;\wedge\; l > r, \Gamma' \;\wedge\; (s > \Gamma \vee t > \Gamma)}$$

where $s|_p$ is not a variable, and x is a new variable. The inference rule for paramodulation left is defined analogously. It is clear that these inference rules are an improvement upon superposition only when applied (at least) with inheritance of the part $x \doteq r$ of the equality constraint (since otherwise the advantage of blocking r is lost).

Basic paramodulation was introduced and proved complete in [BGLS95]. The completeness proof is an easy extension of the proof techniques that we have described above.

The completeness results for theorem proving strategies with constraint inheritance that we have formulated in this chapter do not speak about saturation and redundancy. In fact, we have deliberately ignored this subject here, as the technical details are quite involved. As we have indicated in the proof sketch for Theorem 8, constraint inheritance implies that deduction takes place only with reduced instances of clauses. Therefore any suitable notion of redundancy that is based on implication from smaller clauses will be admissible only if the smaller clauses are reduced themselves. The difficulty is that the rewrite system R with respect to which instances have to be reduced is not effectively known. In other words, irreducibility cannot be absolute but rather has to be relative. We again refer to the overview paper [NR01] for details on this subject.

4.7 Paramodulation with Built-in Equational Theories

In principle, the aforementioned superposition methods apply to any set of clauses with equality, but in some cases special treatments for specific equational subsets of the axioms are preferable. On the one hand, some axioms generate many slightly different permuted versions of clauses, and for efficiency reasons it is many times better to treat all these clauses together as a single one representing the whole class. On the other hand, special treatment can avoid non-termination of completion procedures, as would occur with the equation $f(a, b) \simeq c$ in the presence of the laws of associativity and commutativity of f. Also, some equations like the commutativity axiom are more naturally viewed as "structural" axioms (defining a congruence relation on terms) rather than as "simplifiers" (defining a reduction relation). This allows one to extend completion procedures to deal with congruence classes of terms instead of single terms, i.e., working with a *built-in* equational theory E, and performing rewriting and completion with special E-matching and E-unification algorithms.

Special attention has always been given to the case where E consists of axioms of associativity (A) and commutativity (C), which occur very frequently in practical applications, and are well-suited for being built in due to their permutative nature.

Building permutativity into the data structure requires adapting the semantics of the symbol $>$ in the constraints. Now, the ordering $\succ$ has to be an *AC-compatible* reduction ordering (if $s' =_{AC} s \succ t =_{AC} t'$ then $s' \succ t'$) that is total on the AC-congruence classes of ground terms ($s \neq_{AC} t$ implies that either $s \succ t$ or $t \succ s$). Similarly $\doteq$ is interpreted as AC-equality. For simplicity, we will restrict ourselves again to Horn clauses, although all results can be extended to general clauses in the same way as before.

The full inference system for Horn clauses modulo AC, called $\mathcal{H}_{AC}$ includes the rules of $\mathcal{H}$ (under the new semantics for $\succ$ and $\doteq$) *plus* the following three specific rules:

AC-superposition right:

$$\frac{\Gamma' \rightarrow l \simeq r \qquad \Gamma \rightarrow s \simeq t}{\Gamma', \Gamma \rightarrow s[f(r, x)]_p \simeq t \quad \mid \; s|_p \doteq f(l, x) \;\wedge\; l > r, \Gamma' \;\wedge\; s > t, \Gamma}$$

AC-superposition left:

$$\frac{\Gamma' \rightarrow l \simeq r \qquad \Gamma, \underline{s \simeq t} \rightarrow \Delta}{\Gamma', \Gamma, s[f(r, x)]_p \simeq t \rightarrow \Delta \quad \mid \; s|_p \doteq f(l, x) \;\wedge\; l > r, \Gamma' \;\wedge\; s > t, \Gamma}$$

AC-top-superposition:

$$\frac{\Gamma' \rightarrow l' \simeq r' \qquad \Gamma \rightarrow l \simeq r}{\Gamma', \Gamma \rightarrow f(r', x') \simeq f(r, x) \quad \mid \; f(l', x') \doteq f(l, x) \;\wedge\; l' > r', \Gamma' \;\wedge\; l > r, \Gamma}$$

In these rules, where x and x' are new variables, the term l can be restricted to terms that carry the AC-symbol f at their top. This property is expressible in the constraint language and could be added to the constraint. The new variable x introduced in an inference by AC-superposition (right or left) is a so-called *extension variable* for f. AC-top-superposition is only needed if both l and l' have head symbol f. Finally, as in the non-AC case, the superposition inferences are needed only if $l|_p$ is non-variable, and AC-superposition (left or right) can be restricted to f positions p in l which do not occur as arguments of an f occurrence.

Theorem 11. *The inference system $\mathcal{H}_{AC}$ is refutationally complete for sets of Horn clauses with built-in AC-theories.*

4.7.1 Constraint Inheritance and Built-in Theories

By inheriting equality constraints one can avoid some of the complexity of E-unification, which otherwise is one of the main drawbacks of working with built-in theories. In the AC-case there may be doubly exponentially many AC-unifiers for any two terms e.g. a minimal complete set for $x + x + x$ and $y_1 + y_2 + y_3 + y_4$ contains more than a million unifiers. Therefore the number of possible conclusions of an inference can be extremely high.

Theorem 12. *The inference system $\mathcal{H}_{AC}$ is refutationally complete with ordering constraint inheritance for well-constrained sets S of Horn clauses with built-in AC-theories.*

Again this result can be extended to general clauses using the same techniques as in the free case (see [NR97] for details).

As said before, by inheriting constraints, the computation of unifiers and the generation of multiple conclusions in an inference becomes unnecessary. But it is possible to go beyond. By using constraints it is possible to deal, in an effective way, with theories E with an *infinitary* E-unification problem, i.e. theories where for some unification problems any complete set of unifiers is infinite. This is the case of theories containing only associtivity axioms for some function symbols, which is developed in a similar way as done here for the AC-case in [Rub96].

Finally, by inheriting equality constraints one can consider any built-in theory E, even when the E-unification problem is undecidable, if an adequate inference system and ordering are found (although these ideas require a further development for concrete E). Incomplete methods can then be used for detecting cases of unsatisfiable constraints, and only when a constrained empty clause $\square \mid T$ is found, one has to start (in parallel) a semidecision procedure proving the satisfiability of T. This method is interesting as well if decision procedure for the E-unification problem is available, since the incomplete methods can be more efficient and hence more efective in practice.

4.8 Effective Saturation of First-Order Theories

Saturation up to redundancy also terminates for many consistent theories. This is particularly the case if ordering constraints are inherited throughout inferences and if sufficiently powerful techniques for simplification and the elimination of redundancies are employed. In this section we briefly describe two of the applications in which finitely saturated theories play a central rôle. We intend to demonstrate that saturation can be understood as a (partial) compilation process through which, when it terminates, certain objectives with regard to efficiency can be achieved in an automated way.

In this section we expand on the material provided in Section 4.5.1 above, but—for reasons the discussion of which is beyond the scope of these notes—restrict ourselves to non-equational clauses where superposition specializes to ordered resolution as defined in Section 4.4.7.

Remember that we formally view non-equational atoms $p(s_1, \ldots, s_k)$ as special equations of the form $p(s_1, \ldots, s_k) \simeq \top$, so that formally $p(s_1, \ldots, s_k)$ and the s_j are terms, which we have called, respectively, atomic terms and non-atomic terms. In this section when we speak of an *atom*, we mean an atomic term, and when we speak of a *term* we mean a non-atomic term. Consequently, a *term ordering* will be any well-founded ordering on non-atomic terms, whereas an *atom ordering* will be any well-founded ordering on atomic terms. Also, atom orderings will usually be *compatible* with some term ordering in that $p(s_1, \ldots, s_k) \succ q(t_1, \ldots, t_n)$ in the atom ordering, whenever for all $1 \leq i \leq n$ there exists a $1 \leq j \leq k$ such that $s_j \succ t_i$ in the term ordering. A term s is said to *occur* in a clause C, if s occurs at an argument position of a predicate in C. Note that, by this definition, a proper subterm of a term occurring in C will not necessarily be considered as occurring in C.

4.8.1 Automated Complexity Analysis

Motivated by work on "local theories" by [McA93], in [BG96] the relation between saturation and decidability issues has been refined to complexity issues. It turns out that the complexity of the ordering used for saturation is directly related to the complexity of the entailment problem for a theory.

Let S be a set of clauses (with variables). The *(ground) entailment problem* for the *theory* S consists in checking whether or not a *query* C, where C is a variable-free clause, is logically implied by S. A local theory is one where this problem can be decided merely by considering ground instances of S derived, by means of a term ordering from the ground terms in S and C. In this section, a *term ordering* $\succ$ can be any partial, well-founded ordering on ground terms.

More precisely, suppose that the *complexity* of a term ordering $\succ$ can be bounded by functions f, g in that there are at most $O(f(n))$ ground terms smaller than or equal to some ground term in any given finite set of terms of size n,[7] and such that these smaller terms can be enumerated in time $O(g(n))$.

[7] This is the number of symbols needed to write down the terms in T in (non-sharing) prefix notation.

Given such an ordering $\succ$ of finite complexity and a set of ground terms T, by $T_{\preceq}$ we denote the set of ground terms smaller than (in $\succ$) or equal to a term in T. Moreover, given a set of ground terms T, by S_T we denote the set of ground instances of S that contain terms in T only. We call S (order-) *local* (with respect to $\succ$), if for each ground query C, $S \models C$ if, and only if, $S_{T_{\preceq}} \models C$, where T is the set of ground terms occurring in C or in S. Applying the result of Dowling and Gallier [DG84], if S contains m clauses and is local with respect to $\succ$—an ordering of complexity f, g—then the entailment problem for queries of size n is decidable in time $O(m * f(n)^k + g(n))$ where k depends only on the theory S. Often the theory S is considered a constant so that we obtain $O(f(n)^k + g(n))$, with k a constant, as an upper bound for the time complexity.

In particular, the entailment problem is of polynomial time complexity, if the ordering is of polynomial complexity. For the constant k one may take the *degree* of S, where the degree of a clause is the minimal number of different terms containing all the variables in the clause, and where the degree of a finite set of clauses is the maximum of the degrees of its element. Order locality extends the concept of locality as introduced by McAllester which appears as the special case in which the term ordering is the subterm ordering. It turns out there is a close relationship between order localiy and saturation with respect to ordered resolution.

Given a *total* term ordering $\succ$, we call S *saturated* (with respect to $\succ$) if S is saturated up to redundancy under ordered resolution with selection based on $\succ$ and with respect to some atom ordering $\succ_0$ that is compatible with $\succ$.[8] If the term ordering $\succ$ is only partial, we call N *saturated* (with respect to $\succ$) whenever N is saturated with respect to *every* total term ordering containing $\succ$.

Theorem 13 ([BG96]). *Suppose $\succ$ is a term ordering, and S is a set of Horn clauses that is saturated with respect to $\succ$. Then S is local with respect to $\succ$.*

Proof. (Sketch) Suppose C is a ground clause. Then we may define a total term ordering $\succ'$ extending $\succ$ such that $s \succ' t$ whenever s is a term for which there is no term t' in C such that $t' \succeq s$. Also, let $\succ_0$ be a total, well-founded atom ordering that is compatible with $\succ'$. By assumption, S is saturated under ordered (with respect to $\succ_0$) resolution with selection based on $\succ'$. Since S is saturated, inferences in which both premises are clauses in S are redundant. To decide whether the ground query C is entailed, one negates C,

[8] Selection of negative literals is said to be *based* on a term ordering $\succ$, whenever a negative literal is selected if, and only if, it contains the maximal term of the clause. One notices that with ordering-based selection, the particular choice of a compatible atom ordering $\succ_0$ does not affect the ordering constraints associated with the resolution inferences. The choice may have an effect on redundancy, though.

adds the resulting unit clauses to S, and saturates the resulting set. Clauses D generated by ordered inferences with one premise in S and one premise from $\neg C$ cannot generate clauses with a term not smaller in $\succ$ than, or equal to, some term in C. Atoms containing such terms would be larger in $\succ_0$ than any atom in $\neg C$. By induction this property is maintained for any clause derived subsequently from S, $\neg C$, and D. Thus, if C is a logical consequence of S, it already follows from the set of ground instances of S in which all terms are smaller than (in $\succ$), or equal to, a term in C.

Consequently, the entailment problem for a finite Horn theory S (considered constant) is of complexity $O(f^k + g)$, for some constant k, if S is saturated with respect to a term ordering of complexity f, g.

Example 5. The following represents a tractable (incomplete) fragment of propositional logic.

$$\begin{aligned}
true(p) &\rightarrow true(or(p,q))\\
true(p) &\rightarrow true(or(q,p))\\
true(or(p,q)), true(not(p)) &\rightarrow true(q)\\
true(or(p,q)), true(not(q)) &\rightarrow true(p)\\
true(not(p)), true(not(q)) &\rightarrow true(not(or(p,q)))\\
true(not(or(p,q))) &\rightarrow true(not(p))\\
true(not(or(p,q))) &\rightarrow true(not(q))\\
true(not(p)), true(p) &\rightarrow true(\mathsf{ff})\\
true(\mathsf{ff}) &\rightarrow true(p)\\
true(not(not(p))) &\rightarrow true(p)\\
true(p) &\rightarrow true(not(not(p)))
\end{aligned}$$

This set of Horn clauses is saturated under all total and well-founded extension of the least partial ordering on terms satifying (i) $s \succ t$ if t is a proper subterm of s, (ii) $or(s,t) \succ not(s)$, (iii) $or(s,t) \succ not(t)$, and (iii) $s \succ \mathsf{ff}$, for $s \neq \mathsf{ff}$. This fact can be checked automatically by the Saturate system [GNN94]. In this partial ordering, any term s has at most linearly many (in the size of s) smaller terms. Hence ground entailment and, thus, provability for the encoded fragment of propositional logic, is decidable in polynomial time. In fact it is decidable in linear time since the degree of any clause is 1.

Sometimes the natural presentation of a theory is not saturated with respect to a desired ordering, but can be finitely saturated, see [BG96] for examples that have also been run on Saturate. In such cases saturation can be viewed as an optimizing compiler that adds sufficiently many “useful” consequences to a theory presentation so as to achieve a certain complexity bound for its entailment problem.

In order for saturation to terminate for these weak orderings (that is, to simultaneously saturate a theory with respect to all total, well-founded extensions of the partial ordering), ordering constraints have to be inherited throughout inferences (as explained in the section 4.6.2 for the general case of equational clauses). Moreover, redundancy elimination techniques have to be implemented. In this context it is important to note that while in the equational case the inheritance of ordering constraints comes at the expense of a weaker concept of redundancy, resolution with constraint inheritance is compatible with the standard notion of redundancy (as defined in the section 4.5).

4.8.2 Deduction with Saturated Presentations

Saturated presentations of a theory may give rise to improved inference systems for that theory. Again, deduction with constrained clauses is essential for both obtaining the saturated presentation as well as for defining the resulting specialized inference system.

The following constrained clause set T defines the transitive-reflexive closure p^* of a binary predicate p:

$$\begin{array}{rlll}
 & \to p^*(x,x) & & (1)\\
p(x,y) & \to p^*(x,y) & & (2)\\
p^*(x,y)\ ,\ p^*(y,z) & \to p^*(x,z) & & (3)\\
p^*(x,y)\ ,\ p(y,z) & \to p^*(x,z) & \mid\ y > z, y > x & (4)\\
p(x,y)\ \ ,\ p^*(y,z) & \to p^*(x,z) & \mid\ x > y, x > z & (5)\\
p^*(x,y)\ ,\ p(y,z) & \to p^*(x,z) & \mid\ z > x, z > y & (6)\\
p(x,y)\ \ ,\ p(y,z) & \to p(x,z) & \mid\ y > x, y > z & (7)
\end{array}$$

The presentation is saturated under ordered resolution with selection. The ordering has to be chosen such that it is well-founded and total on ground terms and such that (i) $A \succ B$ whenever the maximal term in A is greater then the maximal term in B, and (ii) $p^*(s,t) \succ p(u,v)$, whenever the maximal term of s and t is the same as the maximal term of u and v. The selection function is assumed to select the maximal negative atom, if the maximal term of the clause occurs either in a negative p^*-atom or else in a negative p-atom but not in a positive p^*-atom. In all other cases, no atom is selected. Checking that the system is saturated is not difficult, but tedious. A sufficiently powerful saturation procedure such as the one implemented in the Saturate system [GNN94] will produce the set automatically from the first three clauses.[9]

[9] Clause (7) is not a consequence of (1)–(3). The actual clause obtained with Saturate was $p(x,y),\ p(y,z) \to p^*(x,z)\ \mid\ y > x, y > z$. But in the application we are interested in, non-theory clauses do not contain negative occurrences of p, so that the use of clause (7) is justified.

For instance, suppose

$$\frac{p(x,y),\ p^*(y,z) \to p^*(x,z) \ \mid\ x > y, x > z \quad p^*(x,z),\ p^*(z,u) \to p^*(x,u)}{p(x,y),\ p^*(y,z),\ p^*(z,u) \to p^*(x,u) \ \mid\ x > y, x > z, x \geq u}$$

is an inference by ordered resolution with selection (and with constraint inheritance) from the fifth and the third clause. Then $p^*(x,z)$ must be selected in the second premise, and therefore $x \succeq u$ must be true, as is reflected in a constraint in the conclusion. If $x = u$, then the conclusion of the inference is subsumed by clause (1). If $x \succ u$, and $x \succ y$ and $x \succ z$, then the conclusion follows from the two clauses $p^*(y,z),\ p^*(z,u) \to p^*(y,u)$ and $p(x,y),\ p^*(y,u) \to p^*(x,u)$, both of which are smaller than the main premise of the above inference. We may conclude that the inference is redundant.

Now consider the saturation of a clause set $T \cup N$ that contains T as a sub-theory. We may assume that atoms with predicate symbol p occur only positively, and atoms with p^* occur only negatively, in N. (Any negative [positive] occurrence of p [p^*] in N can be replaced by p^* [p] without affecting the satisfiability or unsatisfiability of $T \cup N$.) The following observations apply to ordered inferences in which one premise is from T and one from N.

(a) Clauses (3) or (4) can be ruled out as premises from T, because a (negative) atom with predicate p^* is selected in them, while N contains no positive occurrences of such atoms.

(b) Hyper-resolution inferences[10] with clauses in N as side premises (electrons) and clause (7) as main premise (nucleus) can be written as follows (in the ground case, with the theory clause omitted):

Ordered chaining, right

$$\frac{\Gamma \to p(s,t), \Delta \quad \Gamma' \to p(t,u), \Delta'}{\Gamma, \Gamma' \to p(s,u), \Delta, \Delta' \ \mid\ t > s, u \wedge p(s,t) > \Gamma, \Delta \wedge p(t,u) > \Gamma', \Delta'}$$

(c) In clauses (5) and (6) the positive atom is maximal. Ordered resolution with side premise (5) or (6) and a clause N as main premise, followed by resolving the resulting negative atom with predicate p by another clause in N, is captured by the following inferences:

Ordered chaining, left (I):

$$\frac{\Gamma \to p(s,t), \Delta \quad \Gamma', \underline{p^*(s,u)} \to \Delta'}{\Gamma, \Gamma', p^*(t,u) \to \Delta, \Delta' \ \mid\ s > t, u \wedge p(s,t) > \Gamma, \Delta}$$

Ordered chaining, left (II):

$$\frac{\Gamma \to p(t,s), \Delta \quad \Gamma', \underline{p^*(u,s)} \to \Delta'}{\Gamma, \Gamma', p^*(u,t) \to \Delta, \Delta' \ \mid\ s > t, u \wedge p(t,s) > \Gamma, \Delta}$$

[10] Hyper-resolution is resolution where simultanously *all* selected negative literals are resolved Hyper-resolution can be simulated by iterated ordered resolution with selection. The optimization in hyper-resolution over iterated ordered resolution is that the intermediate results need not be kept.

As before, selection is indicated by underlining, and nothing is assumed to be selected in clauses which do not have anything underlined.

Inferences involving clauses (1) or (2) need not be dealt with specifically.

In summary, when one extends ordered resolution with selection by ordered chaining one obtains a refutationally complete specialization of resolution for theories with transitive and reflexive relations p^* in which no resolution inferences with the transitivity axiom for p^* are required.[11] In [BG94b] specific methods from term rewriting have been employed to obtain closely related results. The above considerations demonstrate that saturation of a theory, which can be automated to a large extent, may produce a practically useful inference system, the completeness of which might otherwise require nontrivial meta-level arguments.

It seems that many other theories, including congruence relations, orderings, and distributive lattices, can be engineered in a similar way. This sheds some new light on how theory resolution can be efficiently implemented in practice for theories that can effectively be saturated in such a way that "problematic" clauses such as transitivity are eliminated to a large extent. Pre-saturation of theory modules contributes to an efficient handling of large, but structured theories. Similar ideas have been pursued in [Non95] in the context of theorem proving for modal logics, where specific saturations of the respective "background theories" form a core part of similar methods.

Acknowledgements. We are grateful to Chris Lynch for his detailed comments on a previous version of this paper.

References

[Bac88] Leo Bachmair. Proof by consistency in equational theories. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 228–233, Edinburgh, Scotland, 5–8 July 1988. IEEE Computer Society.

[BG94a] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[BG94b] L. Bachmair and H. Ganzinger. Rewrite techniques for transitive relations. In *Proc. 9th IEEE Symposium on Logic in Computer Science*, pages 384–393. IEEE Computer Society Press, 1994.

[BG96] D. Basin and H. Ganzinger. Complexity analysis based on ordered resolution. In *Proc. 11th IEEE Symposium on Logic in Computer Science*, pages 456–465. IEEE Computer Society Press, 1996.

[BT00] L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In *Proceedings of the Seventeenth International Conference on Automated Deduction (CADE-00)*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 64–78, Berlin, 2000. Springer-Verlag.

[11] The chaining inferences encode certain ordered inferences with transitivity. The advantage over unordered hyper-resolution strategies with the transitivity clause is a better balancing between forward and backward computation.

[BG98] Leo Bachmair and Harald Ganzinger. Equational reasoning in saturation-based theorem proving. In W. Bibel and P. Schmitt, editors, *Automated Deduction: A Basis for Applications.* Kluwer, 1998.

[BGLS95] L. Bachmair, H. Ganzinger, Chr. Lynch, and W. Snyder. Basic paramodulation. *Information and Computation*, 121(2):172–192, 1995.

[Com90] Hubert Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–411, 1990.

[DG84] W. F. Dowling and J. H. Gallier Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. Logic Programming*, 1:267-284, 1984.

[Fit90] Melvin Fitting. First-order logic and automated theorem proving. Springer-Verlag, 1990.

[GNN94] H. Ganzinger, R. Nieuwenhuis, and P. Nivela. The Saturate system. User manual maintained on the Web, 1994.

[Kap97] D. Kapur. Shostak's congruence closure as completion. In H. Comon, editor, *Proceedings of the 8th International Conference on Rewriting Techniques and Applications*, pages 23–37, 1997. Vol. 1232 of *Lecture Notes in Computer Science*, Springer, Berlin.

[McA93] D. A. McAllester. Automated recognition of tractability in inference relations. *J. Association for Computing Machinery*, 40(2):284–303, 1993.

[Nie93] Robert Nieuwenhuis. Simple LPO constraint solving methods. *Information Processing Letters*, 47:65–69, August 1993.

[Nie99] Robert Nieuwenhuis. Rewrite-based deduction and symbolic constraints. In Harald Ganzinger, editor, *Automated Deduction—CADE-16*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 302–313, Berlin, 1997. Springer-Verlag.

[Non95] Andreas Nonnengart. *A Resolution-Based Calculus for Temporal Logics.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, December 1995.

[NR97] Robert Nieuwenhuis and Albert Rubio. Paramodulation with Built-in AC-Theories and Symbolic Constraints. *Journal of Symbolic Computation*, 23(1):1–21, May 1997.

[NR01] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In J.A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning.* Elsevier Science Publishers (to appear), 2001.

[Rub96] Albert Rubio. Theorem proving modulo associativity. In *1995 Conference of the European Association for Computer Science Logic*, LNCS 1092, Paderborn, Germany, September 1996. Springer-Verlag.

[RW69] G. A. Robinson and L. T. Wos. Paramodulation and theorem-proving in first order theories with equality. *Machine Intelligence*, 4:135–150, 1969.

[SL91] Wayne Snyder and Christopher Lynch. Goal directed strategies for paramodulation. In Ronald V. Book, editor, *Rewriting Techniques and Applications, 4th International Conference*, LNCS 488, pages 150–161, Como, Italy, April 10–12, 1991. Springer-Verlag.

5 Functional and Constraint Logic Programming[⋆]

Mario Rodríguez-Artalejo[1]

Dpto. Sistemas Informáticos y Programación, UCM, Madrid

5.1 Introduction

Starting at a seminal paper published by J. Jaffar and J.L. Lassez in 1987 [JL87], Constraint Logic Programming (CLP) has developed as a powerful programming paradigm which supports a clean combination of logic (in the form of Horn clauses) and domain-specific methods for constraint satisfaction, simplification and optimization. The well established mathematical foundations of logic programming [Llo87, Apt90] have been succesfully extended to CLP [JMMS96]. Simultaneously, practical applications of CLP have arisen in many fields. Good introductions to the theory, implementation issues and programming applications of CLP languages can be found in [JM94, MS98]. On the other hand, the combination of logic programming with other declarative programming paradigms (especially functional programming) has been widely investigated during the last decade, leading to useful insights for the design of more expressive declarative languages. The first attempt to combine functional and logic languages was done by J.A. Robinson and E.E. Sibert when proposing the language LOGLISP [RS82]. Some other early proposals for the design of functional + logic languages are described in [De86]. A more recent survey of the operational principles and implementation techniques used for the integration of functions into logic programming can be found in [Han94b].

Declarative programming, in the wide sense, should provide a logical semantics for programs. Nevertheless, many of the early proposals for the integration of functional and logic programming had no clear logical basis. Among the logically well-founded approaches, most proposals (as e.g. [JLM84, GM87, Höl89]) have suggested to use *equational logic*. This allows to represent programs as sets of oriented equations, also known as *rewrite rules*, whose computational use can rely on a well established theory [DJ90, Klo92, BN98]. *Goals* (also called *queries*) for a functional logic program can be represented as systems of equations, and *narrowing* (a natural combination of rewriting and unification, originally proposed as a theorem proving tool [Sla74, Lan75, Fay79, Hul80]) can be used as a goal solving mechanism.

[⋆] This research has been partially supported by the Spanish National Project TIC98-0445-C03-02 "TREND" and the Esprit BRA Working Group EP-22457 "CCLII".

Under various more or less reasonable conditions, several narrowing strategies are known to be complete for goal solving; see [DO90, Han94b, MH94].

This paper has been written as background material for a three hours lecture given at the *International Summer School on Constraints in Computational Logics*, held at Gif-sur-Yvette, France, on September 5–8, 1999. Our aim is to deal with the combination of functional, logic and constraint programming. Regarding the functional component, we are interested in *non-strict* functional languages such as Haskell [Pe99]. Due to *lazy evaluation*, functions in a non-strict language may return a result even if the values of some arguments are not known, or known only partially. In Section 5.2 we will argue that equational logic is not adequate to characterize the semantics of lazy functions, and we will describe quite in detail an approach to the integration of first-order functional programming and logic programming, based on a rewriting logic called CRWL. This logic supports non-deterministic functions, whose combination with lazy evaluation turns out to be helpful. We will present the formalization of program semantics and goal solving in CRWL, as well as $\mathcal{TOY}$ [LFSH99], an experimental language and system that implements the CRWL paradigm.

In the rest of the paper we will introduce various extensions of the CRWL approach in a less detailed way. In Section 5.3 we will consider extensions of CRWL to support programming with algebraic datatypes and higher-order functions. Algebraic datatypes are not yet fully implemented in $\mathcal{TOY}$, but we will show executable examples to illustrate some interesting higher-order programming techniques. In Section 5.4 we will briefly sketch further extensions of CRWL to allow for primitive datatypes and constraints, in the spirit of the CLP scheme. Some of these extensions are not yet implemented, but we will show programming examples using symbolic disequality constraints and arithmetic constraints over the real numbers, which are available in the $\mathcal{TOY}$ system. All along the paper, we will try to justify the adventages of CRWL as a logical framework for the design of declarative programming languages, and we will present comparisons with some other related approaches. In Section 5.5 we will summarize our conclusions and we will point to some lines of on-going work.

The ideas and results been presented here are borrowed from a number of previous works of the Declarative Programming Group at Universidad Complutense de Madrid [GMHGRA96, GMHGLFRA99, GMHGRA97, GMHGRA99, ASRA97b, ASRA97a, ASRAar, ASLFRA98, ASLFRA99], where the mathematical technicalities regarding CRWL and its various extensions have been preented in detail. By contrast, in the current presentation we will try to emphasize the motivation and to provide more intuitive ideas and examples.

5.2 A Rewriting Logic for Declarative Programming

The aim of this section is to present the integration of functional and logic programming in the framework of a rewriting logic, as well as its realization in the $\mathcal{TOY}$ language. We will start by presenting programs as rewrite systems, and motivating the need for a special rewriting logic.

5.2.1 Programming with Rewrite Systems

For terms and rewrite rules, we will use a notation compatible with that in [DJ90,Klo92,BN98]. We will use constructor-based signatures $\Sigma = \langle DC, FS \rangle$, where $DC = \bigcup_{n \in \mathbb{N}} DC^n$ resp. $FS = \bigcup_{n \in \mathbb{N}} FS^n$ are sets of *data constructors* resp. *defined function symbols* with associated arities. As notational conventions, we will assume $c, d \in DC$, $f, g \in FS$ and $h \in DC \cup FS$. We also assume that countably many variables (noted as X, Y, Z, etc.) are available. Given any set $\mathcal{X}$ of *variables*, we will consider the set $Exp_\Sigma(\mathcal{X})$ of all terms built from symbols in $\mathcal{X} \cup DC \cup FS$, and also the set $Term_\Sigma(\mathcal{X})$ of all terms built from symbols in $\mathcal{X} \cup DC$. Terms $l, r, e \in Exp_\Sigma(\mathcal{X})$ will be called *expressions*, while terms $s, t \in Term_\Sigma(\mathcal{X})$ will be called *constructor terms* or also *data terms*. Expressions without variables will be called *ground* or *closed*. Moreover, we will say that an expression e is in *head normal form* iff e is a variable X or has the form $c(\overline{e}_n)$ for some data constructor $c \in DC^n$ and some n-tuple of expressions $\overline{e}_n$. The notion of head normal form comes from functional programming, where it has proved useful for the implementation of lazy evaluation.

For any expression e, $var(e)$ will stand for the set of all variables occurring in e. An analogous notation will be used for data terms and other syntactic objects, whenever appropiate. For given sets of variables $\mathcal{X}$ and $\mathcal{Y}$, we will also consider *constructor-based* substitutions, also called *data substitutions*, as mappings $\theta : \mathcal{X} \to Term_\Sigma(\mathcal{Y})$, extended to $\theta : Exp_\Sigma(\mathcal{X}) \to Exp_\Sigma(\mathcal{Y})$ in the natural way. $Subst_\Sigma(\mathcal{X}, \mathcal{Y})$ will denote the set of all data substitutions $\theta : \mathcal{X} \to Term_\Sigma(\mathcal{Y})$, and the notation "$e\theta$" will be used in place of "$\theta(e)$". For finite $\mathcal{X} = \{X_1, \ldots, X_n\}$, a substitution $\theta \in Subst_\Sigma(\mathcal{X}, \mathcal{Y})$ such that $\theta(X_i) = t_i$ for $1 \le i \le n$, will be noted as a set of *variable bindings* $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$.

We will represent programs as term rewrite systems (shortly, TRSs), i.e. sets $\mathcal{P}$ of rewrite rules of the form $l \to r$, obeying two restrictions which are common in functional programming. Firstly, rules must be *constructor-based*; i.e. l must be of the form $f(\overline{t}_n)$ with $f \in FS^n$ and $t_i \in Term_\Sigma(\mathcal{X})$. Secondly, rules must be *left-linear*, i.e. no variable is allowed to have more than one occurrence in l. In functional programming, data terms are viewed as representations of *values*, while expressions must be evaluated by rewriting, expecting to obtain a value as result. For instance, the following rewrite rules define some simple functions related to lists and natural numbers:

Example 1. Lists and numbers.

```
head([X|Xs]) → X           repeat1(X) → [X|repeat1(X)]
tail([X|Xs]) → Xs          repeat2(X) → [X,X|repeat2(X)]

void(z)     → [ ]           single(z)     → [z]
void(s(N)) → void(N)        single(s(N)) → single(N)
```

In Example 1, the data constructors `z` (for *zero*) and `s` (for *successor*) can be used to represent natural numbers in Peano notation. We also find a nullary data constructor `[ ]` for the empty list, and a binary data constructor used in mixfix notation to construct lists of the form `[X|Xs]` with head `X` and tail `Xs`. The expression `[X,X|repeat2(X)]` is meant as an abbreviation for `[X|[X|repeat2(X)]]`. We use similar abbreviations in the sequel. Functions `void` resp. `single` are intended to take arbitrary numbers as arguments, returning the empty list `[ ]` resp. the singleton list `[z]`. Functions `repeat1` and `repeat2` are intended to return infinite lists. In particular, the expected value of the expression `repeat1(z)` is the *infinite list* consisting of repeated occurrences of `z`. Since we are assuming a lazy semantics, the expected value of the expression `head(repeat1(z))` is `z`, even if the infinite value of `repeat1(z)` cannot be computed in finitely many steps. The expected value of `repeat2(z)` is the same as that of `repeat1(z)`. However, if we look to the program as an equational theory (by forgetting about the left-to-right ordering of the rewrite rules), the equation `repeat1(z)≈repeat2(z)` cannot be deduced by equational reasoning. Moreover, there is no data term in the signature of the program which represents the common value of both expressions.

As a consequence of the previous observations, we cannot claim that the semantics of a lazy functional program $\mathcal{P}$ is characterized in terms of deducibility in the corresponding equational theory. In the case of pure logic programs, there is a well known logical characterization of program semantics. A program can be presented as a set $\mathcal{P}$ of *definite Horn clauses* $a \Leftarrow b_1, \dots, b_n$, where a, b_i are atomic formulae (also called *atoms)* of the form $p(t_1, \dots, t_n)$, built from some n-ary *predicate symbol* p and data terms t_i. Horn logic allows one to deduce atomic formulas from a given program $\mathcal{P}$ by means of a single inferrence, namely to infer $a\theta$ from $b_1\theta, \dots, b_n\theta$ whenever $a \Leftarrow b_1, \dots, b_n \in \mathcal{P}$, $\theta \in Subst_\Sigma(\mathcal{X}, \mathcal{Y})$. The semantics of a pure logic program $\mathcal{P}$ is given by the set of atomic formulas that can be deduced from $\mathcal{P}$ in Horn logic. This set characterizes the *least Herbrand model* of $\mathcal{P}$ over the Herbrand universe with variables, and this fact is very helpful for proving soundness and completeness of SLD resolution [Llo87, Apt90] as a goal solving mechanism. For similar reasons, it is desirable to find a logical characterization of the semantics of lazy functions, as we will do in the next subsection. For the moment, note that we can unify functional and logic programming at the syntactic level, by using conditional rewrite rules in place of Horn clauses. This is illustrated in Example 2, where we consider the problem of computing a list `Zs` as the result of (non-deterministically) merging two given lists `Xs` and `Ys`. We show the natural definition of a `merge` predicate by means of Horn clauses, along with its translation to rewrite rules, where `merge` becomes a function that returns the boolean value `true` (a nullary data constructor).

Example 2. A pure logic program for merging lists:

```
merge([ ],Ys,Ys)
merge([X|Xs],[ ],[X|Xs])
merge([X|Xs],[Y|Ys],[X|Zs]) ⇐ merge(Xs,[Y|Ys],Zs)
merge([X|Xs],[Y|Ys],[Y|Zs]) ⇐ merge([X|Xs],Ys,Zs)
```

And its translation to conditional rewrite rules:

```
merge([ ],Ys,Zs)            → true ⇐ Zs == Ys
merge([X|Xs],[ ],[Z|Zs])    → true ⇐ Z == X, Zs == Xs
merge([X|Xs],[Y|Ys],[Z|Zs]) → true ⇐ Z == X,
                                     merge(Xs,[Y|Ys],Zs) == true
merge([X|Xs],[Y|Ys],[Z|Zs]) → true ⇐ Z == Y,
                                     merge([X|Xs],Ys,Zs) == true
```

Example 2 illustrates a typical logic programming ability, namely to express *don't know non-determinism.* In any Prolog system, the goal

```
merge([1,2],[3,4],Zs)
```

would return different values for `Zs`, via a backtracking mechanism. As an alternative, it is possible to define *non-deterministic functions* by means of rewrite rules. For instance, non-deterministic list merge can be expressed as follows:

Example 3. A non-deterministic function for merging lists:

```
merge([ ],Ys)         → Ys
merge([X|Xs],[ ] )    → [X|Xs]
merge([X|Xs],[Y|Ys]) → [X|merge(Xs,[Y|Ys])]
merge([X|Xs],[Y|Ys]) → [Y|merge([X|Xs],Ys)]
```

Using the rewrite rules in Example 3, the expression `merge([1,2],[3,4])` can be reduced to different data terms, corresponding to all possible ways of merging the two lists `[1,2]` and `[3,4]`. Therefore, the TRS in this example is not confluent. Typical functional languages such as Haskell [Pe99] assume confluent TRSs as programs. However, as we will see later, non-deterministic functions behave better than Prolog-like predicates in some situations.

All the rewrite rules in the rest of the paper will be implicitly assumed to be constructor-based and left-linear. Moreover, the word "program" will always mean a set $\mathcal{P}$ of conditional rewrite rules $l \to r \Leftarrow C$, with conditions of the form $C = l_1 == r_1, \ldots, l_k == r_k$, understood as a *conjunction* of k atomic statements $l_i == r_i$, whose meaning will be explained below.

5.2.2 The Rewriting Logic CRWL

As shown by Example 1, equational logic is not adequate for the semantics of lazy functions. There are two difficulties: first, we have to distinguish between unevaluated expressions and their values, represented as data terms; and second, for some expressions such as `repeat1(z)`, the corresponding value is infinite and cannot be represented by any data term. An idea borrowed from denotational semantics [GS90, Mos90] is helpful at this point. We can introduce a special symbol $\perp$ to represent the *undefined value*. For any signature Σ, we will write $\Sigma_\perp$ for the result of adding $\perp$ as a new nullary data constructor. The notations $Exp_{\Sigma_\perp}(\mathcal{X})$, $Term_{\Sigma_\perp}(\mathcal{X})$ and $Subst_{\Sigma_\perp}(\mathcal{X},\mathcal{Y})$ will stand for so-called *partial* expressions, data terms and data substitutions, where the symbol $\perp$ can occur. Partial data terms naturally play the role of finite approximations of possibly infinite values. For instance, `[z, z, z, z |`$\perp$`]` denotes a list formed by four occurrences of `z`, followed by an undefined rest, which is a finite approximation of the value of `repeat1(z)`. Expressions, terms and substitutions without any occurrence of $\perp$ will be called *total*.

Following [GMHGRA96, GMHGLFRA99], we will define a Constructor-based conditional ReWriting Logic (CRWL, for short) where two different kinds of atomic statements can be deduced: *approximation statements* $e \to t$, intended to mean that the data term t approximates the value of the expression e; and *joinability statements* $l == r$, intended to mean that the expressions l and r have a common *total* approximation $t \in Term_\Sigma(\mathcal{X})$ such that both $l \to t$ and $r \to t$ hold. Note that statements $l == r$ differ from algebraic equalities $l \approx r$ in two respects: firstly, $l == r$ requires a total data term t as common value for l and r; secondly, t is not required to be unique. Requiring unicity of t would lead to a deterministic version of $==$ which has been used under the name *strict equality* in some lazy functional logic languages [GLMP91, MNRA92]. The original motivation for introducing strict equality was to enable better completeness results for conditional narrowing, as we will see in Subsection 5.2.4. On the other hand, the motivation for introducing approximation statements is to characterize the semantics of an expression e w.r.t. a given program $\mathcal{P}$ as the set of all partial data terms $t \in Term_{\Sigma_\perp}(\mathcal{X})$ such that $e \to t$ can be proved from $\mathcal{P}$ in CRWL. A CRWL proof of a statement $e \to t$ roughly corresponds to a finite sequence of rewrite steps going from e to t in the TRS $\mathcal{P} \cup \{X \to \perp\}$. Unfortunately, this simple idea does not work directly, because it leads to an unconvenient treatment of non-determinism. This is shown by the next example.

Example 4. Call-time choice semantics.

```
coin → z           X + z    → X
coin → s(z)        X + s(Y) → s(X + Y)         double(X) → X + X
```

In order to evaluate the expression `double(coin)`, it seems convenient to choose some possible value for the argument expression `coin`, and then apply the function `double` to that value. More generally, the evaluation of any expression of the form $f(e_1, \dots, e_n)$ can proceed by *first* choosing some approximation $t_i \in Term_{\Sigma_\perp}(\mathcal{X})$ of e_i's value (for all $1 \leq i \leq n$) and *then* computing $f(t_1. \dots, t_n)$. This particular treatment of non-determinism is known as *call-time choice*. The rewrite sequence

$$\texttt{double(coin)} \to \texttt{coin + coin} \to^* \texttt{z + s(z)} \to \texttt{s(z)}$$

shows that arbitrary rewriting can violate call-time choice. Operationally, respecting call-time choice in this and other similar examples corresponds to a *sharing* mechanism for multiple occurrences of variables in the right-hand sides of rewrite rules. Following [GMHGLFRA99,Hus92,Hus93], we will adopt call-time choice semantics for non-determinism, so that the statement `double(coin)` $\to$ `s(z)` will be not CRWL-deducible from the program in Example 4. We formally define CRWL by means of the following inference rules:

Definition 1. Inference rules of CRWL.
BT Bottom: $e \to \perp$ for any $e \in Exp_{\Sigma_\perp}(\mathcal{X})$.
RR Restricted Reflexivity: $X \to X$ for any variable X.
DC Decomposition: $\dfrac{e_1 \to t_1 \ \cdots \ e_n \to t_n}{c(\overline{e}_n) \to c(\overline{t}_n)}$ for $c \in DC^n$, $t_i \in Term_{\Sigma_\perp}(\mathcal{X})$.
OR Outer Reduction: $\dfrac{e_1 \to t_1\theta \ \cdots \ e_n \to t_n\theta \qquad C\theta \qquad r\theta \to t}{f(\overline{e}_n) \to t}$

for $t \in Term_{\Sigma_\perp}(\mathcal{X}),\ t \neq \perp,\ f(\overline{t}_n) \to r \Leftarrow C\ \in \mathcal{P},\ \theta \in Subst_{\Sigma_\perp}(\mathcal{Y}, \mathcal{X})$.
JN Joinability: $\dfrac{l \to t \qquad r \to t}{l == r}$ for *total* $t \in Term_{\Sigma}(\mathcal{X})$. □

Rule (BT) corresponds to an implicit rewrite rule $X \to \perp$, meaning that $\perp$ approximates the value of any expression. Rules (RR), (DC) and (OR) implicitly encode reflexivity, monotonicity and transitivity of the rewriting relation, while rule (JN) is used to establish the validity of conditions in the case of conditional rewrite rules. Note that rule (OR) uses *partial data substitutions* θ to instatiate rewrite rules from $\mathcal{P}$, while classical rewriting [DJ90, Klo92, BN98] would use arbitrary substitutions of *total expressions* for variables. It is because of this difference that CRWL expresses *call-time choice* semantics. Moreover, (BT) and (OR) (in cooperation with the other inference rules) also ensure that CRWL expresses a non-strict semantics for function evaluation. As an illustration, we show a CRWL proof of `head(repeat1(z))` $\to$ `z`, using the program from Example 1. From a computational viewpoint, the proof corresponds to an evaluation of the argument expression `repeat1(z)` to *head normal form*, which suffices for the outer function `head` to obtain its value `z`.

Example 5. A proof in CRWL.

1. `head(repeat1(z))` $\to$ `z` by (OR), 2, 3.
2. `repeat1(z)` $\to$ `[z|`$\bot$`]` by (OR), 3, 4.
3. `z` $\to$ `z` by (DC).
4. `[z|repeat1(z)]` $\to$ `[z|`$\bot$`]` by (DC), 3, 5.
5. `repeat1(z)` $\to \bot$ by (BT).

An alternative formalization of CRWL, where transitivity and monotonicity become explicit, can be found in [GMHGLFRA99]. The presentation chosen here will be more convenient for the rest of this paper. In the sequel we will also use the notations $\mathcal{P} \vdash_{CRWL} e \to t$ resp. $\mathcal{P} \vdash_{CRWL} l == r$ to indicate that $e \to t$ resp. $l == r$ can be deduced from the program $\mathcal{P}$ in CRWL. For a compound joinability condition C, we will also use the notation $\mathcal{P} \vdash_{CRWL} C$ to indicate that $\mathcal{P} \vdash_{CRWL} l == r$ holds for all $l == r \in C$.

At this point we can compare CRWL with another well known approach to rewriting as logical deduction, namely Meseguer's *Rewriting Logic* [Mes92] (shortly RWL, in what follows), which has been also used as a basis for computational systems and languages such as Maude [CDE+99], Elan [BKK+96] and CafeOBJ [DFI+96]. In spite of some obvious analogies, there are several clear differences regarding both the intended applications and the semantics. CRWL is intended as a logical basis for declarative programming languages involving lazy evaluation, while RWL has been designed with broader aims, as a logical framework in which other logics can be represented, as well as a semantic framework for the specification of languages and (possibly concurrent) systems. Accordingly, RWL is not constructor-based and lacks the analogon to our rule (BT). Moreover, RWL adopts run-time choice rather than call-time choice. We believe that call-time choice is a more convenient option for programming purposes. Let us motivate this claim with a programming example, where call-time choice non-determinism and lazy evaluation will cooperate. *Permutation sort* is an algorithm that sorts a given list `Xs` by choosing some permuation `Ys` non-deterministically, and checking that `Ys` is sorted. This is considered a typical example of the conciseness and clarity of declarative programming. In Example 6 we find a CRWL formulation of permutation sort in logic programming style, where we assume that the ordering function `<=` and the boolean conjunction function `/\` are defined elsewhere.

Example 6. Permutation sort using naïve generate and test.

```
npermSort(Xs,Ys)        → true ⇐ permutation(Xs,Ys) == true,
                                 isSorted(Ys) == true

permutation([ ],[ ])    → true
permutation([X|Xs],Zs) → true ⇐ permutation(Xs,Ys) == true,
                                 insertion(X,Ys,Zs) == true

insertion(X,[ ],Zs)     → true ⇐ Zs == [X]
insertion(X,[Y|Ys],Zs) → true ⇐ Zs == [X,Y|Ys]
insertion(X,[Y|Ys],Zs) → true ⇐ insertion(X,Ys,Us) == true,
                                 Zs == [Y|Us]
```

```
isSorted([ ])       → true
isSorted([X])       → true
isSorted([X,Y|Zs]) → X <= Y /\ isSorted([Y|Zs])
```

Predicate `npermSort` follows a naïve *generate and test* scheme, that leads to extreme inefficiency when executed in Prolog. Due to the left-most selection strategy, each permutation `Ys` of `Xs` is completely computed before checking if it is sorted. An extension of Prolog's computation model by a coroutining mechanism [Nai87] has been proposed to solve this kind of problems. This approach requires the explicit introduction of "wait" declarations for some predicates.

On the other hand, in order to solve generate-and-test problems in a lazy functional language (where there is no built-in search for solutions), one would typically follow the 'list of successes' approach [Wad85]: generate the list of all candidate solutions (all permutations, in this case) and filter it by means of a testing function. Although lazy evaluation ensures that the list of candidates is generated only to the extent required by the tester (which can reject a partially generated solution), in any case it can be a very large structure. Moreover, some special language constructions, such as *list comprehensions* [Pe99], are usually needed to program the generation of the candidates' list in a declaratively neat way. In CRWL, using a non-deterministic function as generator leads naturally to an alternative formulation of generate-and-test problems, where no explicit list of candidate solutions is needed. Example 7 applies this idea to the permutation sort algorithm.

Example 7. Permutation sort using lazy generate and test.

```
permSort(Xs) → checkSorted(permute(Xs))

checkSorted(Ys) → Ys ⇐ isSorted(Ys) == true

permute([ ])      → [ ]
permute([X|Xs])  → insert(X,permute(Xs))

insert(X,[ ])     → [X]
insert(X,[Y|Ys]) → [X,Y|Ys]
insert(X,[Y|Ys]) → [Y|insert(X,Ys)]
```

Note that a non-deterministic function `permute` is used here as a generator for the different permutations of `Xs`, while the testing function `checkSorted` will check each of these permutations, trying to return as result a sorted one. Since CRWL implementations are expected to use lazy evaluation, `checkSorted` can fail without completing the evaluation of its argument. In this case, `permute` will backtrack and generate another permutation as a new argument for `checkSorted`. This combination of a non-deterministic generator and a lazy tester leads to a *lazy generate and test* scheme that can be applied to many problems, improving the efficiency of the naïve generate and test method. Call-time choice is needed to ensure that the two occurrences of `Ys` in the right-hand side of the rewrite rule for `checkSorted` refer to the same permutation of `Xs`; otherwise, the algorithm would be incorrect. Therefore, call-time choice not only reflects the adventages of sharing as an implementation technique, but is also semantically relevant.

Before closing this subsection, we will briefly discuss the relationship between CRWL, Horn logic and equational logic. First, we claim that Horn logic can be easily encoded into CRWL. For any given Horn logic program $\mathcal{P}$, let $\mathcal{P}^\star$ be the result of translating $\mathcal{P}$ into rewrite rules, by the method suggested by Example 2. Then, the following result is easy to prove, using induction on the lengths of formal proofs:

Proposition 1. *Horn logic reduces to CRWL.*
For every atomic formula a in $\mathcal{P}$'s signature, a is deducible from $\mathcal{P}$ in Horn logic if and only if $a \to true$ is deducible from $\mathcal{P}^\star$ in CRWL. ■

On the other hand, it is well known that equational logic can be encoded into Horn logic. Given any equational theory $\mathcal{E}$, it suffices to view the operation symbols as constructors, to regard the equality symbol $\approx$ as a predicate, and to add Horn clauses to express the standard equality axioms: reflexivity, symmetry, transitivity and congruence w.r.t. all the operations. The resulting Horn theory $\mathcal{E}^\star$ encodes $\mathcal{E}$ in the following sense:

Proposition 2. *Equational logic reduces to Horn logic.*
For every equation $l \approx r$ in $\mathcal{E}$'s signature, $l \approx r$ is deducible from $\mathcal{E}$ in equational logic if and only if $l \approx r$ is deducible from $\mathcal{E}^\star$ in Horn logic. ■

From Propositions 1 and 2 it follows that equational logic can be also encoded in CRWL. On the other hand, no obvious encoding of CRWL into equational logic or Horn logic exists, to the best knowledge of the author.

5.2.3 Model-Theoretic Semantics

Program semantics in declarative languages is considered useful for different purposes; see e.g. [BGLM94]. In the case of a pure logic program $\mathcal{P}$, the least Herbrand model of $\mathcal{P}$ over the Herbrand universe with variables (also known as $\mathcal{C}$-semantics [FLMP93]) can be characterized as the set of all atomic formulas that are deducible from $\mathcal{P}$ in Horn logic. In this subsection we will present similar results for CRWL. Models of CRWL programs with signature Σ will be algebras $\mathfrak{A}$ consisting of a domain $D_\mathfrak{A}$ and suitable interpretations for the symbols in Σ. Partial data terms $t \in Term_{\Sigma_\perp}(\mathcal{X})$ are intended to represent more or less defined values. As a semantic counterpart for this, one needs a partial order over the domain $D_\mathfrak{A}$ of any algebra $\mathfrak{A}$. Let us recall some basic notions from the theory of semantic domains [GS90]. As usual, we will overload the symbol $\perp$ to denote the minimum element of any semantic domain.

A *poset* with bottom $\perp$ is any set S partially ordered by $\sqsubseteq$, with least element $\perp$. $Def(S)$ denotes the set of all maximal elements $u \in S$, also called *totally defined*. Assume $X \subseteq S$. X is a *directed set* iff for all $u, v \in X$ there exists $w \in X$ s.t. $u, v \sqsubseteq w$. X is a *cone* iff $\perp \in X$ and X is downwards closed w.r.t. $\sqsubseteq$. Directed cones are called *ideals*. We write $\mathcal{C}(S)$ and $\mathcal{I}(S)$ for the sets of cones and ideals of S, respectively. $\mathcal{I}(S)$ ordered by set inclusion $\subseteq$ is a poset with bottom $\{\perp\}$, called the *ideal completion* of S. Mapping each $u \in S$ into the *principal ideal* $\langle u \rangle = \{v \in S \mid v \sqsubseteq u\}$ gives an order preserving embedding.

A poset D with bottom is a *complete partial order* (in short, *cpo*) iff D has a least upper bound $\bigsqcup X$ (also called *limit*) for every directed set $X \subseteq D$. An element $u \in D$ is called *finite* iff whenever $u \sqsubseteq \bigsqcup X$ for a non-empty directed set X, there exists $x \in X$ such that $u \sqsubseteq x$. A cpo D is called *algebraic* iff any element of D is the limit of a directed set of finite elements. It is known that, for any poset with bottom S, $\mathcal{I}(S)$ is the least cpo containing S. Moreover, $\mathcal{I}(S)$ is an algebraic cpo whose finite elements correspond to the principal ideals $\langle x \rangle$, $x \in S$; see for instance [Möl85]. In particular, elements $x \in Def(S)$ correspond to finite and total elements $\langle x \rangle$ in the ideal completion $\mathcal{I}(S)$.

Algebraic cpos are commonly used as semantic domains for the denotational semantics of programming languages [Mos90]. The partial order is understood as an information ordering between partially defined values, thinking of $x \sqsubseteq y$ as the statement that y is more defined than x. We will use posets S, intended as representatives of their ideal completions. In particular, for any set of variables $\mathcal{X}$ we will consider the *Herbrand* poset over $\mathcal{X}$. This is given by the set $Term_{\Sigma_\perp}(\mathcal{X})$ of all partial data terms, equiped with the least partial ordering $\sqsubseteq$ such that $\perp \sqsubseteq t$ for all $t \in Term_{\Sigma_\perp}(\mathcal{X})$ and $c(\overline{t}_n) \sqsubseteq c(\overline{t'}_n)$ whenever $c \in DC^n$ and $t_i \sqsubseteq t'_i$ for all $1 \le i \le n$. The ideal completion of $Term_{\Sigma_\perp}(\mathcal{X})$ is isomorphic to a cpo whose elements are possibly infinite trees with nodes labelled by symbols from $DC \cup \{\perp\}$ in such a way that the arity of each label corresponds to the number of sons of the node; see [GTWJ77]. For instance, assumming data constructors for lists and natural numbers as in Example 1, we can obtain the infinite list of all natural numbers as limit of the following chain of partial data terms:

$$\perp \sqsubseteq \texttt{[z|}\perp\texttt{]} \sqsubseteq \texttt{[z,s(z)|}\perp\texttt{]} \sqsubseteq \texttt{[z,s(z),s(s(z))|}\perp\texttt{]} \sqsubseteq \ldots$$

CRWL-deducibility is compatible with $\sqsubseteq$ in the sense that $\emptyset \vdash_{CRWL} t' \to t \iff t \sqsubseteq t'$ holds for all $t, t' \in Term_{\Sigma_\perp}(\mathcal{X})$. This property is easy to prove by induction. After all these preliminaries, we are ready to define the class of algebras that will play the rôle of models for CRWL programs.

Definition 2. A CRWL Algebra is a structure

$$\mathfrak{A} = \langle D_{\mathfrak{A}},\ \{c^{\mathfrak{A}}\}_{c \in DC},\ \{f^{\mathfrak{A}}\}_{f \in FS} \rangle$$

where

1. $D_{\mathfrak{A}}$ is a poset, called the *domain* of $\mathfrak{A}$.

2. For each $c \in DC^n$, $c^{\mathfrak{A}} : D_{\mathfrak{A}}^n \to \mathcal{I}(D_{\mathfrak{A}})$ is a monotonic mapping returning ideals as results.
3. For each $c \in DC^n$ and for any tuple $\overline{u}_n \in D_{\mathfrak{A}}^n$, $c^{\mathfrak{A}}(\overline{u}_n) = \langle v \rangle$ for some v. Moreover, if $u_i \in Def(D_{\mathfrak{A}})$ for all $1 \leq i \leq n$, then $v \in Def(D_{\mathfrak{A}})$.
4. For each $f \in FS^n$, $f^{\mathfrak{A}} : D_{\mathfrak{A}}^n \to \mathcal{C}(D_{\mathfrak{A}})$ is a monotonic mapping returning cones as results. If it happens that $f^{\mathfrak{A}} : D_{\mathfrak{A}}^n \to \mathcal{I}(D_{\mathfrak{A}})$, we will say that $f^{\mathfrak{A}}$ is *deterministic*. □

Item 4. of the definition reflects the fact that we allow non-deterministic functions in our semantics. Technically, cones are the elements of Hoare's powerdomain [GS90], used in denotational semantics for capturing *don't know* non-determinism. Item 2. means that data constructors must be interpreted as deterministic operations. This is so because ideals in $\mathcal{I}(D_{\mathfrak{A}})$ correspond to single elements in the ideal completion of $D_{\mathfrak{A}}$. The monotonicity requirement in items 2. and 4. ensures that $c^{\mathfrak{A}}$ and $f^{\mathfrak{A}}$ would become *continuous* cpo mappings over the ideal completion $\mathcal{I}(D_{\mathfrak{A}})$. Finally, item 3. means that data constructors must preserve totally defined elements. All these requirements are motivated by the properties of an important subclass of CRWL algebras, so called *Herbrand* algebras.

Definition 3. Herbrand Algebras.
A CRWL algebra $\mathfrak{A}$ of signature Σ is called a *Herbrand algebra* iff the two following conditions are satisfied:
1. $D_{\mathfrak{A}} = Term_{\Sigma_\perp}(\mathcal{X})$, for some set of variables $\mathcal{X}$.
2. For all $c \in Term_{\Sigma_\perp}(\mathcal{X})$, $\overline{t}_n \in Term_{\Sigma_\perp}(\mathcal{X})^n$: $c^{\mathfrak{A}}(\overline{t}_n) = \langle c(\overline{t}_n) \rangle$ □

Example 8. Assume a Herbrand algebra $\mathfrak{A}$ with the signature of Example 4. Then we may expect:
1. $\mathtt{s}^{\mathfrak{A}}(\mathtt{z}) = \langle \mathtt{s}(\mathtt{z}) \rangle$ (a principal ideal).
2. $\mathtt{coin}^{\mathfrak{A}} = \{\mathtt{z}, \mathtt{s}(\mathtt{z}), \mathtt{s}(\perp), \perp\}$ (a cone).
3. $\mathtt{double}^{\mathfrak{A}}(\mathtt{s}(\mathtt{z})) = \langle \mathtt{s}(\mathtt{s}(\mathtt{z})) \rangle$ (a principal ideal).

In fact, item 1. is enforced by the concept of Herbrand algebra, while items 2. and 3. hold if $\mathfrak{A}$ is a least Herbrand model of the program from Example 4, as we will see below.

Next, we explain how to evaluate expressions in a given algebra. To simplify the notation, we write $\{h^{\mathfrak{A}}(u_1, \ldots, u_n) \mid u_1 \in C_1, \ldots, u_n \in C_n\}$, where C_i are certain cones, and $h \in DC^n \cup FS^n$, as $h^{\mathfrak{A}}(C_1, \ldots, C_n)$.

Definition 4. Expression evaluation.
Assume a CRWL algebra $\mathfrak{A}$ and a set of variables $\mathcal{X}$. *Valuations* for $\mathcal{X}$ over $\mathfrak{A}$ are mappings $\eta : \mathcal{X} \to D_{\mathfrak{A}}$. We will say that η is *totally defined* iff $\eta(X) \in Def(D_{\mathfrak{A}})$ for all $X \in \mathcal{X}$. We will write $Val(\mathcal{X}, \mathfrak{A})$ resp. $DefVal(\mathcal{X}, \mathfrak{A})$ for the sets of all valuations resp. totally defined valuations for $\mathcal{X}$ over $\mathfrak{A}$. For given $e \in Exp_{\Sigma_\perp}(\mathcal{X})$, the evaluation of e under $\eta \in Val(\mathcal{X}, \mathfrak{A})$ yields a cone $[\![e]\!]^{\mathfrak{A}}\eta \in \mathcal{C}(D_{\mathfrak{A}})$, defined by structural recursion over e:

1. $[\![\bot]\!]^{\mathfrak{A}}\eta = \langle\bot\rangle \;=\; \{\bot\}$.
2. $[\![X]\!]^{\mathfrak{A}}\eta = \langle\eta(X)\rangle$, for all $X \in \mathcal{X}$.
3. $[\![h(e_1,\dots,e_n)]\!]^{\mathfrak{A}}\eta = h^{\mathfrak{A}}([\![e_1]\!]^{\mathfrak{A}}\eta,\dots,[\![e_n]\!]^{\mathfrak{A}}\eta)$, for all $h \in DC^n \cup FS^n$. □

Note that item 3. in the previous definition reflects call-time choice. Note also that in the case of a Herbrand algebra $\mathfrak{A}$ with $D_{\mathfrak{A}} = Term_{\Sigma_\bot}(\mathcal{Y})$, one has $Val(\mathcal{X},\mathfrak{A}) = Subst_{\Sigma_\bot}(\mathcal{X},\mathcal{Y})$, i.e., valuations are the same as partial data substitutions. The following simple properties of expression evaluation are easily proved by induction; see [GMHGLFRA99].

Proposition 3. *Properties of expression evaluation.*
Assume $t \in Term_{\Sigma_\bot}(\mathcal{X})$, $e \in Exp_{\Sigma_\bot}(\mathcal{X})$, $\eta \in Val(\mathcal{X},\mathfrak{A})$. Then:
1. $[\![e]\!]^{\mathfrak{A}}\eta \in \mathcal{C}(D_{\mathfrak{A}})$.
2. If $f^{\mathfrak{A}}$ is deterministic for every $f \in FS$ occurring in e, then $[\![e]\!]^{\mathfrak{A}}\eta \in \mathcal{I}(D_{\mathfrak{A}})$.
In particular, $[\![t]\!]^{\mathfrak{A}}\eta \in \mathcal{I}(D_{\mathfrak{A}})$.
3. $[\![t]\!]^{\mathfrak{A}}\eta = \langle v\rangle$, for some $v \in D_{\mathfrak{A}}$.
Moreover, if $t \in Term_{\Sigma}(\mathcal{X})$ and $\eta \in DefVal(\mathcal{X},\mathfrak{A})$, then $v \in Def(D_{\mathfrak{A}})$.
4. If $\mathfrak{A}$ is a Herbrand algebra with $D_{\mathfrak{A}} = Term_{\Sigma_\bot}(\mathcal{Y})$, then $[\![t]\!]^{\mathfrak{A}}\eta = \langle t\eta\rangle$. ■

Those CRWL algebras that satisfy all the rewrite rules belonging to a given program $\mathcal{P}$ are called *models* of $\mathcal{P}$. We will say that $\mathfrak{A}$ satisfies a rewrite rule $l \to r \Leftarrow C$ iff l's value includes r's value under any valuation that satisfies C. This makes sense because expression values are cones. More formally:

Definition 5. Models of a program.
Assume an algebra $\mathfrak{A}$ and a program $\mathcal{P}$ of the same signature.

1. $\mathfrak{A}$ satisfies an approximation statement $e \to t$ under $\eta \in Val(\mathcal{X},\mathfrak{A})$ iff $[\![e]\!]^{\mathfrak{A}}\eta \supseteq [\![t]\!]^{\mathfrak{A}}\eta$.
2. $\mathfrak{A}$ satisfies a compound joinability condition C under $\eta \in Val(\mathcal{X},\mathfrak{A})$ iff for every $l == r \in C$ there is some $u \in Def(D_{\mathfrak{A}})$ s.t. $u \in [\![l]\!]^{\mathfrak{A}}\eta \cap [\![r]\!]^{\mathfrak{A}}\eta$.
3. $\mathfrak{A}$ satisfies a rewrite rule $l \to r \Leftarrow C$ iff $[\![l]\!]^{\mathfrak{A}}\eta \supseteq [\![r]\!]^{\mathfrak{A}}\eta$ holds for every $\eta \in Val(\mathcal{X},\mathfrak{A})$ s.t. $\mathfrak{A}$ satisfies C under η.
4. $\mathfrak{A}$ is a model of $\mathcal{P}$ iff $\mathfrak{A}$ satisfies all the rewrite rules belonging to $\mathcal{P}$. □

In the sequel, we write $(\mathfrak{A},\eta) \models \varphi$ to note that $\mathfrak{A}$ satisfies φ under η (where φ maybe an approximation statement or a joinability statement), and $\mathfrak{A} \models \mathcal{P}$ to note that $\mathfrak{A}$ is a model of $\mathcal{P}$. We are mainly interested in *least Herbrand models*, that are defined as follows:

Definition 6. Least Herbrand models.
Assume a program $\mathcal{P}$ of signature Σ and a set of variables $\mathcal{X}$. The least Herbrand model of $\mathcal{P}$ over $\mathcal{X}$ is the Herbrand algebra $\mathfrak{M}_{\mathcal{P}}(\mathcal{X})$ with domain $Term_{\Sigma_\bot}(\mathcal{X})$ s.t., for all $f \in FS^n$, $\bar{t}_n \in Term_{\Sigma_\bot}(\mathcal{X})^n$, $f^{\mathfrak{M}_{\mathcal{P}}(\mathcal{X})}(\bar{t}_n)$ is defined as the set $\{t \in Term_{\Sigma_\bot}(\mathcal{X}) \mid \mathcal{P} \vdash_{CRWL} f(\bar{t}_n) \to t\}$. □

According to this definition, the interpretation of data constructors in least Herbrand models is free (as in any Herbrand algebra), while the interpretation of defined functions is made according to formal deducibility in CRWL. For instance, for the Herbrand algebra $\mathfrak{A}$ in Example 8 we had in mind a least Herbrand model of the program $\mathcal{P}$ from Example 4. In that example we found that the values returned by function `double` were ideals. In general, given $f \in FS$, we say that f is defined as a *deterministic* function by a program $\mathcal{P}$ iff $f^{\mathfrak{M}_{\mathcal{P}}(\mathcal{X})}(\bar{t}_n)$ is an ideal for all possible $t_i \in Term_{\Sigma_\perp}(\mathcal{X})$. Many commonly used functions are deterministic. Determinism is an undecidable property, but some decidable sufficient conditions are known which work well enough in practice [GMHGRA92].

As we have stated in Proposition 1, any pure logic program $\mathcal{P}$ can be translated into an equivalent CRWL program $\mathcal{P}^\star$. From Proposition 1 and Definition 6, it is clear that the least Herbrand models of $\mathcal{P}$ and $\mathcal{P}^\star$ bear the same information. This shows that least Herbrand models in CRWL are a natural generalization of an analogous notion in pure logic programming. The analogy is confirmed by a known characterization of least Herbrand models of a CRWL program $\mathcal{P}$ as least fixpoints of an immediate consequences operator associated to $\mathcal{P}$ [MBP97]. Regarding functional programming, the comparison is more difficult because the usual denotational semantics for functional languages knows no analogon of Herbrand models. In [GMHGRA92] (a paper that was written before developping CRWL) there are results showing that cpo-based Herbrand models are better related to operational semantics of functional programming. Therefore, we claim that least Herbrand models in CRWL are a convenient semantics, both for purely functional programming and for functional logic programming. This is confirmed by a categorical characterization of $\mathfrak{M}_{\mathcal{P}}(\mathcal{X})$ as an initial object in the category of all models of $\mathcal{P}$ [GMHGLFRA99], as well as by the next two results, whose proofs can be found in [GMHGLFRA99].

Lemma 1. *Characteristic Properties of Least Herbrand Models.*
Assume a CRWL program $\mathcal{P}$ and a set of variables $\mathcal{Y}$. Then, $\mathfrak{M}_{\mathcal{P}}(\mathcal{Y})$ is a well defined Herbrand algebra such that:
1. For all $e \in Exp_{\Sigma_\perp}(\mathcal{X})$, $\theta \in Subst_{\Sigma_\perp}(\mathcal{X},\mathcal{Y})$:
$[\![e]\!]^{\mathfrak{M}_{\mathcal{P}}(\mathcal{Y})}\theta = \{t \in Term_{\Sigma_\perp}(\mathcal{Y}) \mid \mathcal{P} \vdash_{CRWL} e\theta \to t\}$.
2. For every approximation or joinability statement φ with variables in $\mathcal{X}$ and
for all $\theta \in Subst_{\Sigma_\perp}(\mathcal{X},\mathcal{Y})$: $(\mathfrak{M}_{\mathcal{P}}(\mathcal{Y}),\theta) \models \varphi$ iff $\mathcal{P} \vdash_{CRWL} \varphi\theta$. ■

Theorem 1. *Canonicity of Least Herbrand Models.*
Assume a CRWL program $\mathcal{P}$ and a set of variables $\mathcal{X}$. Then:

1. *$\mathfrak{M}_{\mathcal{P}}(\mathcal{X}) \models \mathcal{P}$.*
2. *Let φ be an approximation statement or a joinability statement with variables in $\mathcal{X}$, and let $id \in Subst_{\Sigma_\perp}(\mathcal{X},\mathcal{X})$ be the identity substitution. The three following statements are equivalent:*

(a) $\mathcal{P} \vdash_{CRWL} \varphi$.
(b) $(\mathfrak{A}, \eta) \models \varphi$, *for all* $\mathfrak{A} \models \mathcal{P}$ *and all* $\eta \in DefVal(\mathcal{X}, \mathfrak{A})$.
(c) $(\mathfrak{M}_{\mathcal{P}}(\mathcal{X}), id) \models \varphi$. ∎

As an immediate consequence of item 2. in Theorem 1, we obtain that CRWL deducibility is sound and complete w.r.t. semantic validity in all possible models under all possible *totally defined* valuations. In general, this result cannot be weakened to arbitrary valuations. For instance, $\mathcal{P} \vdash_{CRWL} X == X$ holds trivially for any program $\mathcal{P}$, but $(\mathfrak{A}, \eta) \models X == X$ requires η to be totally defined. Another interesting consequence of Theorem 1 is the following result, meaning that statements proved in CRWL have a universal reading:

Corollary 1. *Let* φ *be an approximation statement or a joinability statement with variables in* $\mathcal{X}$*, such that* $\mathcal{P} \vdash_{CRWL} \varphi$*. Then* $\mathcal{P} \vdash_{CRWL} \varphi\theta$ *holds for any* total *data substitution* $\theta \in Subst_{\Sigma}(\mathcal{X}, \mathcal{Y})$*.*

Proof. Assume $\mathcal{P} \vdash_{CRWL} \varphi$. Since θ is a totally defined valuation over $\mathfrak{M}_{\mathcal{P}}(\mathcal{Y})$, we can apply items 1. and 2.(b) of Theorem 1, and we get $(\mathfrak{M}_{\mathcal{P}}(\mathcal{Y}), \theta) \models \varphi$. Then, $\mathcal{P} \vdash_{CRWL} \varphi\theta$ follows from item 2. of Lemma 1. ∎

Note that Corollary 1 does not hold in general for partial data substitutions. For instance, $\mathcal{P} \vdash_{CRWL} X == X$, but not $\mathcal{P} \vdash_{CRWL} \bot == \bot$.

5.2.4 The Lazy Narrowing Calculus CLNC

Given a CRWL program $\mathcal{P}$, we are interested in *goal solving*, in the same vein as in logic programming. A goal G may be any compound joinability condition. When solving G w.r.t. $\mathcal{P}$ we expect as solutions one or more data substitutions θ such that $\mathcal{P} \vdash_{CRWL} G\theta$. Next, we show some concrete examples. In each case we indicate a program $\mathcal{P}$, a goal G and one or more possible solutions.

Example 9. Some goals and solutions.

1. $\mathcal{P}$ from Example 1; $G =$ `head(repeat1(z)) == X`.
   Solutions: $\theta = \{$`X` $\mapsto$ `z`$\}$ (and no more).
2. $\mathcal{P}$ from Example 4; $G =$ `double(coin) == X`.
   Solutions: $\theta_1 = \{$`X` $\mapsto$ `z`$\}$; $\theta_2 = \{$`X` $\mapsto$ `s(s(z))`$\}$ (and no more).
3. $\mathcal{P}$ from Example 4; $G =$ `X + Y == Z`.
   Solutions: $\theta = \{$`Y` $\mapsto$ `s(z)`, `Z` $\mapsto$ `s(X)`$\}$ (and others).
4. $\mathcal{P}$ from Example 1; $G =$ `head(repeat1(X)) == z`.
   Solutions: $\theta = \{$`X` $\mapsto$ `z`$\}$ (and no more).
5. $\mathcal{P}$ from Example 3; $G =$ `merge(Xs,Ys) == [A,B]`.
   Solutions: $\theta = \{$`Xs` $\mapsto$ `[B]`, `Ys` $\mapsto$ `[A]`$\}$ (and others).

Note that solving the goal in item 1. corresponds to computing the value of the expression `head(repeat1(z))` by lazy evaluation. Item 2 corresponds to the evaluation of a non-deterministic expression. Solving the goals in items

3–5. requires to combine unification [Apt90,Llo87] and rewriting. Such a combination is known as *narrowing*, and it was originally proposed as a theorem proving tool [Sla74, Lan75, Fay79, Hul80]. In its simplest form, an *ordinary narrowing step* $e \rightsquigarrow_\sigma e'$ proceeds by selecting a rewrite rule $l \to r$ and a subexpression $e_{|p}$ of e at some position p, which must be not a variable. If needed, $l \to r$ must be renamed to ensure that it has no variables in common with e. Then, assuming that $e_{|p}$ and l unify with most general unifier σ, e' may be taken as the result $e[r]_p\sigma$ of replacing the subexpression of $e\sigma$ at position p by $r\sigma$. This amounts to a rewriting step $e\sigma \to e'$ performed with the rewrite rule $l \to r$ after applying the unifier σ. Most investigations of narrowing as a goal solving mechanism have been made in the framework of equational logic. In this setting, goals are compound of equations $l \approx r$, and they are solved by iterating narrowing steps until the expresions at both sides become unifiable. The composition of all the unifiers used at the different steps gives a solution. For instance, if we interpret the rewrite rules for the operation $+$ in Example 4 as an equational program, the goal in Example 9, item 3, becomes the equation $\texttt{X + Y} \approx \texttt{Z}$, which can be solved by ordinary narrowing as follows:

$$\begin{array}{ll}
\underline{\texttt{X + Y}} \approx \texttt{Z} \rightsquigarrow_{\{Y \mapsto s(Y')\}} & \texttt{s(}\underline{\texttt{X + Y'}}\texttt{)} \approx \texttt{Z} \\
\rightsquigarrow_{\{Y' \mapsto z\}} & \underline{\texttt{s(X)} \approx \texttt{Z}} \\
\rightsquigarrow_{\{Z \mapsto s(X)\}} & \texttt{true.}
\end{array}$$

At each step, the narrowed subexpression has been underlined. The last narrowing step is a ficticious one, which merely computes a most general unifier between the left-hand and the right-hand sides. Composing the unifiers used at the different steps, we obtain the solution $\theta = \{\texttt{Y} \mapsto \texttt{s(z)}, \texttt{Z} \mapsto \texttt{s(X)}\}$ announced in Example 9, item 3. More generally, assuming any confluent TRS as program, and systems of equations $l \approx r$ as goals, it is known that ordinary narrowing is sound and complete for computing normalized solutions w.r.t. equational semantics; see e.g. [Hul80]. This result holds for general TRSs, not only for left-linear and constructor-based ones. Nevertheless, ordinary narrowing is not satisfactory in practice because of the huge size of the search space that must be explored. More refined narrowing strategies have been designed with the aim of pruning the search space without loss of completeness; see [DO90, Han94b] for a survey. Completeness results for narrowing strategies sometimes rely on subtle hypothesis, and some curious examples of incompleteness are known; see [MH94].

For our purposes, we need a variant of narrowing that is sound and complete w.r.t. CRWL, rather than equational logic. Unfortunately, ordinary narrowing and its classical refinements are unsound because they fail to respect call-time choice. Coming back to item 2. in Example 9, ordinary narrowing can compute the solution $\theta = \{\texttt{X} \mapsto \texttt{s(z)}\}$, that is incorrect w.r.t. CRWL:

$$\begin{array}{l}
\underline{\texttt{double(coin)}}\ \texttt{== X} \rightsquigarrow_{\{\}} \underline{\texttt{coin}}\ \texttt{+ coin == X} \rightsquigarrow_{\{\}} \\
\texttt{z + }\underline{\texttt{coin}}\ \texttt{== X} \rightsquigarrow_{\{\}} \underline{\texttt{z + s(z)}}\ \texttt{== X} \rightsquigarrow_{\{\}} \\
\texttt{s(}\underline{\texttt{z + z}}\texttt{) == X} \rightsquigarrow_{\{\}} \underline{\texttt{s(Z) == X}} \rightsquigarrow_{\{X \mapsto s(Z)\}} \texttt{true.}
\end{array}$$

Here, call-time choice has been violated by the first narrowing step (which in this particular case reduces to rewriting). In order to solve the problem, we must avoid that the non-deterministic expression `coin` occurring as argument in `double(coin)` is copied twice in the second goal. One way to do this is to transform the initial goal into the new goal `coin` $\rightarrow$ `Y, Y + Y == X`. More generally, each time we need to narrow a goal $f(\overline{e}_n) == e$ by means of a rewrite rule $f(\overline{t}_n) \rightarrow r \Leftarrow C \in \mathcal{P}$, we will transform the goal into a new one as follows:

$$f(\overline{e}_n) == e \;\Vdash_{ON}\; e_1 \rightarrow t_1, \ldots, e_n \rightarrow t_n,\; C,\; r == e$$

We can think of $e_i \rightarrow t_i$ as a *delayed unification*, corresponding to *parameter passing*. If needed, narrowing will be applied to the expressions e_i later on, and all those bindings that may be created for variables occurring in $e_i \rightarrow t_i$ will be shared all along the whole goal. In this way, non-strictness of functions and call-time choice non-determinism will be respected, as required by CRWL semantics.

A goal transformation step such as $\Vdash_{ON}$ above (where $\underline{\text{ON}}$ abbreviates $\underline{\text{O}}$uter $\underline{\text{N}}$arrowing) can be also understood by analogy with the inference rule (OR) from CRWL. This means that delayed unifications $e_i \rightarrow t_i$ do not correspond to solving equations, since the semantic meaning of $\rightarrow$ is not equality; remember Definition 5. Guided by the inference rules of CRWL as given in Definition 1, we will now define a $\underline{\text{C}}$onstructor-based $\underline{\text{L}}$azy $\underline{\text{N}}$arrowing $\underline{\text{C}}$alculus (CLNC, for short) consisting of a number of goal transformation rules. CLNC will solve goals by lazy narrowing in a sound and complete fashion w.r.t. CRWL semantics. Before presenting CLNC, we give a precise definition for the class of goals we are going to work with. In addition to joinability statements $l == r$, goals will also include approximation statements $e \rightarrow t$ for the reasons explained above, as well as bindings of the form $X \mapsto s$, intended to represent a partially computed solution. Those variables of a goal that have been introduced locally by previous CLNC steps will be marked as existential. Formally:

Definition 7. Admissible Goals have the form $G = \exists \overline{U}. S \,\square\, P \,\square\, E$, where

1. $evar(G) = \overline{U}$ are the *existential variables* of G.
2. $S = X_1 \mapsto s_1, \ldots, X_p \mapsto s_p$ is the *solved part* of G. For $1 \leq i \leq p$, X_i is a variable that occurs exactly once in G, and s_i is a total data term. S is intended to represent the substitution $\sigma_S = \{X_1 \mapsto s_1, \ldots, X_p \mapsto s_p\}$, a partially computed solution.
3. $P = e_1 \rightarrow t_1, \ldots, e_q \rightarrow t_q$ is the *delayed part* of G. For $1 \leq i \leq q$, e_i is a total expression and t_i is a total data term. The statement $e_i \rightarrow t_i$ represents a delayed unification between e_i and t_i, coming from parameter passing. The set of *produced variables* of G is defined as $pvar(P) = var(t_1) \cup \ldots \cup var(t_q)$. The relation $>>_P$ defined as $X >>_P Y$ iff there is some $1 \leq i \leq q$ s.t. $X \in var(e_i)$ and $Y \in var(t_i)$ is called the *production relation* of G.

4. $E = l_1 == r_1, \dots, l_m == r_m$ is the *unsolved part* of G. The set $dvar(E)$ of *demanded variables* of G is defined by the condition $X \in dvar(E)$ iff there is some $1 \leq i \leq m$ s.t. $X = l_i$ or $X = r_i$.
5. G must satisfy the following four properties, called *goal invariants*:
   LN The tuple $(t_1, \dots, t_p)$ is linear.
   EX $pvar(P) \subseteq evar(G)$.
   NC The transitive closure $>>_P^+$ of the production relation is irreflexive.
   SL $var(S) \cap pvar(P) = \emptyset$. □

The fact that $X \in dvar(E)$ will be used to trigger further narrowing of e for any delayed unification $e \to X \in P$. In fact, it is helpful to think of $e \to X$ as a *suspension* that can be resumed by narrowing e as soon as X happens to be demanded. The four goal invariant properties are also quite natural. Essentially, the invariants hold because approximation statements $e_i \to t_i$ will be introduced in goals as delayed unifications w.r.t. linear left-hand sides, using each time a renamed rewrite rule whose variables will be declared as existential and kept disjoint from the previous goal. In the sequel, we will use the notation $f(\bar{t}_n) \to r \Leftarrow C \in_{ren} \mathcal{P}$ to mean a renaming of some rewrite rule in $\mathcal{P}$.

We are now in a position to present the transformation rules of CLNC. The notation $G \Vdash G'$ will mean that G is transformed into G' in one step. The aim when using CLNC is to transform an *initial goal* of the form $G_{ini} = \exists \overline{U}. \,\square\, \square\, E$ into a *solved goal* of the form $G_{sol} = \exists \overline{U}.S \,\square\, \,\square\,$, representing a solution σ_S. By convention, in any admissible goal $G = \exists \overline{U}.S \,\square\, P \,\square\, E$, S must be understood as a *set*, while P and E must be understood as *multisets*. Therefore, CLNC assumes no particular *selection strategy* for choosing the statement within a given goal to be processed in the next step. In addition, for the purpose of applying CLNC transformation rules, statements $l == r$ are seen as symmetric. The notation $svar(e)$ used to state some of the rules, stands for the set of all variables X occurring in expression e at some position whose ancestor positions are all occupied by data constructors. For instance, $svar(c(X, f(Y))) = \{X\}$. The notation "[...]" means an optional part of a goal, present only under certain conditions. Finally, the notation $FAIL$, used in failure rules, represents an inconsistent goal without any solution. Failure rules are theoretically not needed, but any practical implementation should use them to detect unsolvable goals as soon as possible.

The CLNC Calculus

Rules for the Unsolved Part

DC Decomposition
$\exists \overline{U}.S \,\square\, P \,\square\, c(\bar{a}_n) == c(\bar{b}_n),\ E \Vdash \exists \overline{U}.S \,\square\, P \,\square\, \dots, a_i == b_i, \dots,\ E.$

ID Identity
$\exists \overline{U}.S \,\square\, P \,\square\, X == X,\ E \Vdash \exists \overline{U}.S \,\square\, P \,\square\, E.$

SB Solved Binding
$\exists\overline{U}.S \,\Box\, P \,\Box\, X == s,\ E \Vdash \exists\overline{U}.X \mapsto s,\ (S \,\Box\, P \,\Box\, E)\sigma$,
where s data term, $X \notin var(s)$, $X \notin pvar(P)$, $var(s) \cap pvar(P) = \emptyset$,
$\sigma = \{X \mapsto s\}$.

IM Imitation
$\exists\overline{U}.S \,\Box\, P \,\Box\, X == c(\overline{e}_n),\ E \Vdash$
$\exists\overline{X}_n, \overline{U}.X \mapsto c(\overline{X}_n),\ (S \,\Box\, P \,\Box\, \ldots, X_i == e_i, \ldots,\ E)\sigma$,
where **SB** not applicable, $X \notin svar(c(\overline{e}_n))$, $X \notin pvar(P))$, $\overline{X}_n$ fresh variables, $\sigma = \{X \mapsto c(\overline{X}_n)\}$.

ON Outer Narrowing
$\exists\overline{U}.S \,\Box\, P \,\Box\, f(\overline{e}_n) == e,\ E \Vdash \exists\overline{X}, \overline{U}.S \,\Box\, .., e_i \to t_i, ..,\ P \,\Box\, C,\ r == e,\ E$,
where $f(\overline{t}_n) \to r \Leftarrow C \in_{ren} \mathcal{P}$ with fresh variables $\overline{X}$.

CF Conflict Failure
$\exists\overline{U}.S \,\Box\, P \,\Box\, c(\overline{a}_n) == d(\overline{b}_m),\ E \Vdash FAIL$, where $c \neq d$.

OF Occurs Check Failure
$\exists\overline{U}.S \,\Box\, P \,\Box\, X == e,\ E \Vdash FAIL$, where $X \neq e$, $X \in svar(e)$.

Rules for the Delayed Part

DC Decomposition
$\exists\overline{U}.S \,\Box\, c(\overline{e}_n) \to c(\overline{t}_n),\ P \,\Box\, E \Vdash \exists\overline{U}.S \,\Box\, \ldots, e_i \to t_i, \ldots,\ P \,\Box\, E$.

OB Output Binding
$\exists\overline{U}.S \,\Box\, X \to t,\ P \,\Box\, E \Vdash \exists\overline{U}.[X \mapsto t,]\ (S \,\Box\, P \,\Box\, E)\sigma$,
where $[X \notin pvar(P),]$ t is not a variable, $\sigma = \{X \mapsto t\}$.

IB Input Binding
$\exists X, \overline{U}.S \,\Box\, t \to X,\ P \,\Box\, E \Vdash \exists\overline{U}.S \,\Box\, (P \,\Box\, E)\sigma$,
where t is a data term, $\sigma = \{X \mapsto t\}$.

II Input Imitation
$\exists X, \overline{U}.S \,\Box\, c(\overline{e}_n) \to X,\ P \,\Box\, E \Vdash \exists\overline{X}_n, \overline{U}.S \,\Box\, (\ldots, e_i \to X_i, \ldots,\ P \,\Box\, E)\sigma$,
where **IB** is not applicable, $X \in dvar(E)$, $\overline{X}_n$ fresh variables,
$\sigma = \{X \mapsto c(\overline{X}_n)\}$.

EL Elimination
$\exists X, \overline{U}.S \,\Box\, e \to X,\ P \,\Box\, E \Vdash \exists\overline{U}.S \,\Box\, P \,\Box\, E$, where $X \notin var(P \,\Box\, E)$.

ON Outer Narrowing
$\exists\overline{U}.S \,\Box\, f(\overline{e}_n) \to t,\ P \,\Box\, E \Vdash \exists\overline{X}, \overline{U}.S \,\Box\, .., e_i \to t_i, ..,\ r \to t,\ P \,\Box\, C,\ E$,
where t is not a variable or else $t \in dvar(E)$, $f(\overline{t}_n) \to r \Leftarrow C \in_{ren} \mathcal{P}$ with fresh variables $\overline{X}$.

CF Conflict Failure
$\exists\overline{U}.S \,\Box\, c(\overline{e}_n) \to d(\overline{t}_m),\ P \,\Box\, E \Vdash FAIL$, where $c \neq d$.

Some interesting features of CLNC are worth of attention. First, note that the failure rule (OF) is similar to occurs check in logic programming, but requiring $X \in svar(e)$ instead of $X \in var(e)$. The reason is that $X == e$ may have solutions when $X \in var(e) \setminus svar(e)$. For instance, $X == f(X)$ certainly has solutions if f is defined by the rewrite rule $f(X) \to X$. On the other hand, no occurs check is needed in the rules (OB), (IB) and (II), thanks to the goal invariant (NC). Rules (SB), (OB) and (IB) bind a variable and propagate the binding to the whole goal. This kind of transformation is called *variable elimination* in some other narrowing calculi. Its application sometimes helps to avoid the generation of redundant solutions. CLNC, however, performs variable eliminations only in some cases. In particular, propagation of bindings of the form $X \mapsto e$ (where e is not a data term) is generally unsound w.r.t. call-time semantics, and thus avoided. Note also that some CLNC transformations have side conditions to ensure that no produced variables go into the solved part. This is needed to maintain all the four goal invariants, which are used in the proofs of soundeness and completeness of CLNC. The rôle of statements $e \to X$ as suspensions has been already commented above. Rule (ON) for the delayed part can be used to awake suspensions when they are needed, while rule (EL) can discard a suspension that will be not needed any more. This reflects the possibly non-strict behaviour of functions.

Next, we present two concrete computations solving goals in *CLNC*. At each goal transformation step, we underline the relevant subexpression, and we omit superfluous existential quantifiers.

Example 10. First, we compute the solution from Example 9, item 4. This illustrates the use of suspensions $e \to X$ to achieve the effect of lazy evaluation.

$\Box \quad \Box\ \underline{\texttt{head(repeat1(X))}}\ \texttt{== z} \Vdash_{ON}$
$\exists\ \texttt{X', Xs'.}\quad \Box\ \underline{\texttt{repeat1(X)}} \to \texttt{[X'|Xs']}\quad \Box\ \texttt{X' == z} \Vdash_{ON}$
$\exists\ \texttt{X'',X',Xs'.}\ \Box\ \underline{\texttt{X}\to\texttt{X''}}\texttt{,[X''|repeat1(X'')]}\to\texttt{[X'|Xs']}\ \Box\ \texttt{X' == z} \Vdash_{IB}$
$\exists\ \texttt{X', Xs'.}\quad \Box\ \underline{\texttt{[X|repeat1(X)]} \to \texttt{[X'|Xs']}}\quad \Box\ \texttt{X' == z} \Vdash_{DC}$
$\exists\ \texttt{X', Xs'.}\quad \Box\ \underline{\texttt{X} \to \texttt{X'}}\texttt{, repeat1(X)} \to \texttt{Xs'}\quad \Box\ \texttt{X' == z} \Vdash_{IB}$
$\exists\ \texttt{Xs'.}\quad \Box\ \underline{\texttt{repeat1(X)} \to \texttt{Xs'}}\quad \Box\ \texttt{X == z} \Vdash_{EL}$
$\Box \quad \Box\ \underline{\texttt{X == z}} \Vdash_{SB}$
$\texttt{X} \mapsto \texttt{z} \quad \Box \quad \Box$

As a second case, we compute solution θ_1 from Example 9, item 2. This illustrates the use of the delayed part for ensuring call-time choice.

$\Box \quad \Box\ \underline{\texttt{double(coin)}}\ \texttt{== X} \Vdash_{ON}$
$\exists\ \texttt{Y.}\quad \Box\ \texttt{coin} \to \texttt{Y}\quad \Box\ \underline{\texttt{Y + Y}}\ \texttt{== X} \Vdash_{ON}$
$\exists\ \texttt{X',Y.}\quad \Box\ \texttt{Y} \to \texttt{X'},\quad \underline{\texttt{Y} \to \texttt{z}},\quad \texttt{coin} \to \texttt{Y}\quad \Box\ \texttt{X' == X} \Vdash_{OB}$
$\exists\ \texttt{X'.}\quad \Box\ \underline{\texttt{z} \to \texttt{X'}},\quad \texttt{coin} \to \texttt{z},\quad \Box\ \texttt{X' == X} \Vdash_{IB}$
$\Box\ \underline{\texttt{coin} \to \texttt{z}}\quad \Box\ \texttt{z == X} \Vdash_{OR}$
$\Box\ \underline{\texttt{z} \to \texttt{z}}\quad \Box\ \texttt{z == X} \Vdash_{DC}$
$\Box \quad \Box\ \texttt{z == X} \Vdash_{SB}$
$\texttt{X} \mapsto \texttt{z} \quad \Box \quad \Box$

In any practical implementation, call-time choice should be realized by means of sharing. In CLNC, the treatment of approximation statements $e \to t$ as delayed unifications amounts to a formalization of sharing which is semantically sound and very simple to deal with formally. Another known theoretical model of sharing is *term graph rewriting* [BvEG$^+$87], a variant of rewriting based on a representation of expresions as directed graphs. *Term graph narrowing* strategies have been also studied [HP96, HP99a], and some researchers [Hus92, Hus93, SE92] have used term graph narrowing for solving goals in non-deterministic languages with call-time choice semantics. In our setting, we obtain a natural correspondence between CLNC goal transformation rules and CRWL inferences. As far as we know, relating CRWL semantics to some lazy term graph narrowing strategy would be technically more involved.

We now turn to the main theoretical properties of CLNC, namely soundness and completeness w.r.t. CRWL semantics. Before stating them, we will give a precise definition of solution.

Definition 8. Solutions for Admissible Goals.
Let $G = \exists \overline{U}.\ S \ \Box\ P \ \Box\ E$ be an admissible goal for a CRWL program $\mathcal{P}$, and $\mathcal{X}$ some set of variables such that $var(G) \subseteq \mathcal{X}$. A partial substitution $\theta \in Subst_{\Sigma_\perp}(\mathcal{X}, \mathcal{Y})$ is called a *solution* of G (in symbols, $\theta \in Sol(G)$) iff the following conditions hold:
1. $\theta(X)$ is a *total* data term for all $X \in \mathcal{X}$, $X \notin pvar(P)$.
2. θ is a unifier of X and s for all $X \mapsto s \in S$.
3. $\mathcal{P} \vdash_{CRWL} (P \ \Box\ E)\theta$. □

The last item above is intended to mean that $\mathcal{P} \vdash_{CRWL} \varphi\theta$ holds for every $\varphi \in P \cup E$, where $P \cup E$ is viewed as a multiset. Any multiset $\mathcal{M}$ consisting of CRWL proofs, one for each $\varphi\theta \in (P \cup E)\theta$, will be called a *witness* for the fact that $\theta \in Sol(G)$. For instance, the CRWL proof from Example 5 is part of a witness for the solution shown in item 4. of Example 9. Note that solutions for initial goals must be *total* data substitutions $\theta \in Subst_\Sigma(\mathcal{X}, \mathcal{Y})$. Assuming admissible goals G, G', substitutions θ, θ' and a set of variables $\mathcal{X}$, we will use the following notations:
– $\theta = \theta'\ [\mathcal{X}] \iff_{def} \theta(X) = \theta'(X)$ for all $X \in \mathcal{X}$.
– $\theta = \theta'\ [\backslash \mathcal{X}] \iff_{def} \theta(X) = \theta'(X)$ for all $X \notin \mathcal{X}$.
– $\theta \leq \theta'\ [\mathcal{X}] \iff_{def} \theta' = \theta\delta\ [\mathcal{X}]$, for some substitution δ.
– $Sol(G') \subseteq_{ex} Sol(G) \iff_{def}$ for all $\theta' \in Sol(G')$ there is some $\theta \in Sol(G)$ such that $\theta = \theta'\ [\backslash(evar(G) \cup evar(G'))]$.
– $G \Vdash^* G' \iff_{def}$ there are goals $G_i, 0 \leq i < n$, such that $G = G_0 \Vdash G_1 \ldots \Vdash G_{n-1} = G'$.

Each CLNC transformation step is correct in the sense of the following result. A proof can be found in [GMHGLFRA99].

Lemma 2. *Soundness Lemma.*
Assume any admissible goal G for a given program $\mathcal{P}$. Then:
1. If $G \Vdash FAIL$, then $Sol(G) = \emptyset$.
2. Otherwise, if $G \Vdash G'$, then G' is admissible and $Sol(G') \subseteq_{ex} Sol(G)$. ■

Correctness of CLNC computed answers follows from Lemma 2, using induction on the length of CLNC computations. See [GMHGLFRA99] for details.

Theorem 2. *Soundness of CLNC.*
Let G be an initial goal for program $\mathcal{P}$. Assume that $G \Vdash^* \exists\overline{U}.S \,\square\, \square$.
Then $\sigma_S \in Sol(G)$. ■

Completeness of CLNC is proved with the help of a well-founded ordering for witnesses of solutions. Let $\prec$ be the well-founded multiset ordering for multisets of natural numbers [DM79]. Then:

Definition 9. Well-founded Ordering for Witnesses.
Let $\mathcal{M}$, $\mathcal{M}'$ be finite multisets of CRWL proofs. Let $\mathcal{SM}$, $\mathcal{SM}'$ be the corresponding multisets of natural numbers, obtained by replacing each CRWL proof by its *size*, understood as the number of CRWL inference steps. Then we define $\mathcal{M} \lhd \mathcal{M}'$ iff $\mathcal{SM} \prec \mathcal{SM}'$. □

The following result, proved in [GMHGLFRA99], guarantees that CLNC transformations can be chosen to make progress towards the computation of a given solution.

Lemma 3. *Progress Lemma.*
Assume an admissible goal G for some program $\mathcal{P}$, which is not in solved form, and a given solution $\theta \in Sol(G)$ with witness $\mathcal{M}$. Then:
1. There is some CLNC transformation applicable to G.
2. Whatever *applicable CLNC transformation is chosen, there is* some *transformation step $G \Vdash G'$ and some solution $\theta' \in Sol(G')$ with a witness $\mathcal{M}' \lhd \mathcal{M}$, such that $\theta = \theta'$ $[evar(G) \cup evar(G')]$.* ■

By reiterated application of Lemma 3, the following completeness result is easy to prove. See [GMHGLFRA99] for details.

Theorem 3. *Completeness of CLNC.*
Assume an initial goal G for program $\mathcal{P}$ and a given solution $\theta \in Sol(G)$. Then $G \Vdash^* \exists\overline{U}.S \,\square\, \square$ for some S such that $\sigma_S \leq \theta$ $[var(G)]$. ■

In CLNC and other similar calculi there are three independent sources of non-deteminism: (a) the choice of a subgoal in the current goal, (b) the choice of a goal transformation, and (c) the choice of the program rewrite rule to be aplied (in the case of certain goal transformations). As shown by Lemma 3, choices (a) and (b) are *dont'care* in CLNC. Only choice (c) remains *don't know*, which is unavoidable by the nature of narrowing. For this reason, we will say that CLNC is a *deterministic calculus*. This doesn't mean that choices (a) and (b) can be performed arbitrarily, since there are restrictions imposed by the side conditions of the CLNC rules. Any allowed choice, however, can be made *don't care* without loss of completeness. This leaves free room for different selection strategies at the implementation level.

Before closing this section we will compare CLNC with other variants of narrowing. We know already that ordinary narrowing is unsound w.r.t. CRWL semantics, because call-time choice is not respected. As we have mentioned before, *term graph narrowing* strategies [HP96, HP99a] can be used to formalize sharing, relying on a representation of expressions as graphs. Another way to protect call-time choice would be to adopt some *innermost narrowing* strategy [Fri85, Höl89]. Such strategies are complete for confluent and terminating constructor-based TRSs, under equational semantics. However, the termination requirement is too restrictive for CRWL. Therefore, we will limit ourselves to a short survey of some narrowing strategies which are closer to our own approach. We do not pretend to be exhaustive. The reader is referred to [DO90, Han94b, MH94] for more information on narrowing strategies.

Lazy narrowing as an alternative operational semantics for functional languages with a cpo-based (rather than equational) semantics was first proposed by Reddy in [Red85]. This paper presented an informal description of lazy narrowing, which amounts to an analogon of the CLNC rules (ON), (DC) and (CF), along with an eager propagation of bindings $X \mapsto e$ for arbitrary expressions, which can lead to unsoundness in our setting. The functional logic languages K-LEAF [GLMP91] and BABEL [MNRA92] can be seen as forerunners of CRWL. In these two languages we find the distinction between equality and strict equality, as well as a goal solving procedure that is sound and complete w.r.t a model-theoretic semantics based on algebraic cpos. In BABEL (where programs were presented as sets of rewrite rules) the goal solving procedure was lazy narrowing, understood as a restriction of ordinary narrowing that attempted to select outermost narrowing positions, moving to inner ones only in some cases. In K-LEAF (where programs were viewed as sets of Horn clauses) goals were solved by a combination of a flattening process (to cope with nested function calls) and SLD resolution, using an outermost strategy which in fact works analogously to lazy narrowing. The formal correspondence between SLD resolution and a refinement of *basic narrowing* [Hul80,NRS89,MH94] via flattening, has been investigated in [BGM88]. Flattened goals in K-LEAF were required to satisfy certain invariant properties, similar to our goal invariants in Definition 7. Both BABEL and K-LEAF allowed non-strict functions, and no termination hypothesis was needed for completeness. Non-determinism, however, was disallowed. Therefore, there was no need to worry about call-time choice. As a consequence, the goal solving procedures from [GLMP91, MNRA92] can compute CRWL-unsound solutions, unless sharing is used at the implementation level.

Lazy narrowing should be not confused with the *outermost narrowing* strategy which selects outermost narrowable positions and ignores narrowing steps at inner positions. Outermost narrowing is incomplete both in the sense of equational semantics and in the sense of CRWL semantics. This is shown by the following example, borrowed from [You89]:

Example 11. Consider the TRS $\mathcal{P}$ consisting of the rewrite rules $f(Y, a) \to \mathit{true}$, $f(c, b) \to \mathit{true}$ and $g(b) \to c$. Note that $\mathcal{P}$ is confluent, strongly terminating, and a CRWL program. The data (and thus normalized) substitution $\{X \mapsto b\}$ is a solution of the goal $f(g(X), X) \approx \mathit{true}$ w.r.t. equational semantics, and also a solution of $f(g(X), X) == \mathit{true}$ w.r.t. CRWL semantics. However, outermost narrowing computes only the incomparable solution $\{X \mapsto a\}$.

In the previous example the subexpression $f(g(X), X)$ selected by the outermost strategy has two different ground instances, namely $f(g(a), a)$ and $f(g(b), b)$, both obtained by means of a normalized substitution, of which the first is a redex while the second is not. In [Pad87, Ech90, Ech92] a narrowing strategy was called *uniform* in the case that every subexpression e selected by the strategy has the property that *all* the ground instances of e given by normalized substitutions are redices. The following results were proved in [Pad87, Ech90, Ech92]: assuming a confluent and strongly terminating TRS, any uniform narrowing strategy is complete for computing ground normalized solutions w.r.t. equational semantics. Moreover, the outermost narrowing strategy is uniform for those TRSs which satisfy certain (rather restrictive) conditions.

Being not the same as outermost narrowing, the term *lazy narrowing* is often used to mean any narrowing strategy that has the ability to delay unification problems or to avoid the eager selection of inner narrowing positions. From a more stringent viewpoint, lazy computations should consist of steps that are *needed* in some well-defined sense. For the case of rewriting, an adequate theory of *neededness* has been proposed by Huet and Lévy [HL79,HL91] for *orthogonal* TRSs. By definition, an orthogonal TRS is left-linear and such that no subterm of a left-hand side is unifiable with another left-hand side, except in the obvious trivial cases. Such systems are always confluent. The so-called *needed narrowing* strategy [AEM94] has been designed for *inductively sequential systems*, a proper subclass of the orthogonal constructor-based TRSs. Needed narrowing is sound and complete for this class of TRSs, and it enjoys interesting and useful optimality properties. Succesful computations consist of needed steps (in the sense of Huet and Levy's theory) and achieve minimal length under the assumption of sharing. Moreover, different computed solutions for the same goal are independent. A more recent paper [Ant97] has proposed an extension of needed narrowing called *inductively sequential narrowing.* This strategy is sound and complete for *overlapping inductively sequential systems*, a proper extension of the class of inductively sequential systems which includes non-confluent TRSs. Therefore, [Ant97] permits non-deterministic functions, but call-time choice semantics is not adopted. The optimality properties of needed narrowing are retained only in a rather weak sense by inductively sequential narrowing. In particular, different computed solutions are not always independent.

The soundness and completeness results for needed narrowing given in [AEM94] refer to goals of the form $l \doteq r \rightarrow^? \texttt{true}$, and the intended solutions are constructor substitutions θ such that $l\theta \doteq r\theta$ can be rewritten to the data constructor `true`. Soundness and completeness are proved w.r.t. rewriting, without any explicit concern on semantics. The special function symbol $\doteq$ is defined by rewrite rules to behave as strict equality in the following sense: $l\theta \doteq r\theta$ can be rewritten to `true` iff the two expressions $l\theta$ and $r\theta$ can be both reduced to the same *ground constructor term*. The view of strict equality as a function symbol defined by program rules was adopted also by K-LEAF [GLMP91] and BABEL [MNRA92]. In comparison to CRWL (where joinability is a logical primitive), this approach has the disadventage that goals are often solved by enumerating infinitely many ground solutions, instead of computing a single more general one. Coming back to Example 9 (item 3), CLNC computes the solution $\{\texttt{Y} \mapsto \texttt{s(z)},\ \texttt{Z} \mapsto \texttt{s(X)}\}$ for the goal `X + Y == Z`, while needed narrowing enumerates infinitely many solutions $\{\texttt{Y} \mapsto \texttt{s(z)},\ \texttt{X} \mapsto \texttt{s}^n\texttt{(z)},\ \texttt{Z} \mapsto \texttt{s}^{n+1}\texttt{(z)}\}$ (for all $n \geq 0$) for the corresponding goal $\texttt{X + Y} \doteq \texttt{Z} \rightarrow^? \texttt{true}$.

Inductively sequential narrowing [Ant97] obtains similar soundness and completeness results for goals of the form $e \rightarrow^? t$, where e is any expression, t is a data term, and intended solutions are data substitutions θ such that $e\theta$ can be rewritten to t. Since $\doteq$ is treated as a defined function symbol, this subsumes the soundness and completeness results form [AEM94] as a particular case. Nevertheless, from the proofs in [AEM94] it follows that needed narrowing is also sound and complete for the broader class of goals $e \rightarrow^? t$ considered in [Ant97]. The *outer narrowing* strategy of [You89] also considered goals of this form, which were viewed there as *matching problems* modulo the equational theory given by the underlying TRS. Outer narrowing is a lazy strategy that tries to apply narrowing steps at outermost positions and performs inner narrowing steps only if they contribute to some outer step later on; see [You89] for a formal definition. Outer narrowing is sound and complete for solving goals $e \rightarrow^? t$ where t is a data term. Moreover, the computed solutions are pairwise incomparable data substitutions θ such that $e\theta$ can be rewritten to t in the given TRS. These results were proved in [You89] for orthogonal *constructor-based* TRSs, which include inductively sequential systems as a particular case. However, [You89] gave no characterization of outer narrowing steps as *needed* steps.

We have presented our lazy narrowing calculus CLNC as a goal transformation system. This is is not a new idea. Many authors have investigated goal transformation systems for solving narrowing and unification problems; see e.g. [Höl89, Sny91]. In particular, [MOI96] investigates a lazy narrowing calculus LNC bearing some analogies to CLNC. Goals in LNC are systems of equations. A distinction is made between ordinary equations and parameter passing equations, somehow corresponding to the unsolved and delayed part of CLNC goals, respectively. However, [MOI96] works with inconditional

TRSs and considers completeness w.r.t. normalized solutions and equational semantics. A *selection function* is assumed for selecting the equation to be transformed in the current goal. *Strong completeness* (i.e., completeness for any choice of the selection function) holds in those cases where basic narrowing is complete. *Completeness* with the left-most selection strategy holds for all confluent TRSs. Moreover, for the case of orthogonal TRSs, [MOI96] proves the completeness of an *outside-in narrowing* strategy which extends the *outer narrowing* strategy from [You89]. Using this result, [MOI96] also proves the following: variable elimination for *solved equations* $X \approx t$ with $X \notin var(t)$ that are descendants of a parameter passing equation, can be performed eagerly without loss of completeness for left-most LNC, assuming an orthogonal TRS. This seems related to the CLNC rules (OB) and (IB), although CLNC works only with constructor-based rewrite rules.

In a later paper [MO95], more results on removing non-determinism from the LNC calculus have been obtained, assuming the left-most selection strategy. The previous result on eager variable elimination for left-most LNC has been extended to the class of left-linear confluent TRSs. Moreover, for the subclass of *constructor-based* left-linear and confluent TRSs, *dont' care* choice among all the applicable calculus transformations can be adopted without loss of completeness, provided that the semantics of goal equations is changed to strict equality. For this class of rewrite systems, LNC can be replaced by an equivalent calculus called LNC$_d$, which makes a syntactic distinction between ordinary equations and parameter passing equations. LNC$_d$ goal transformation rules are quite similar to those of CLNC. However, LNC$_d$ does not deal with condidional TRSs, and some of the LNC$_d$ rules are unsound w.r.t. CRWL semantics in the case of non-confluent CRWL programs. For instance, the analogon of the CLNC rule **IB** in LNC$_d$ allows an arbitrary expression in place of t, which leads to unsoundness w.r.t. CRWL semantics. Unlike LNC$_d$, CLNC is not complete w.r.t. the left-most selection strategy, because the selection of some subgoals must be postponed to ensure soundness w.r.t. CRWL semantics. Like *outer narrowing* [You89], LNC$_d$ enjoys the property of computing independent solutions in the case of orthogonal constructor-based TRSs. Such a result has not been proved for CLNC.

There are still some other lazy narrowing calculi bearing analogies to CLNC. The *left-most outside-in* narrowing calculus OINC from [IN97] is quite similar to LNC and it has a variant S-OINC tailored to strict equality, which is close to LNC$_d$; see [MO95] for a more detailed comparison. A conditional extension LCNC of LNC has been presented in [HM97]. This narrowing calculus is complete whenever basic narrowing is complete. The lazy narrowing calculus [Pre95] works with higher-order conditional rewrite rules whose left-hand sides are *patterns* in Miller's sense [Mil91]. Conditions and goals are composed of statements of the form $l \rightarrow^? r$. The intended solutions for such goals are normalized substitutions θ such that $r\theta$ is a normal form and $l\theta$ can be rewritten to $r\theta$. This amounts to consider rewriting itself as the intended semantics. The behaviour of CLN for left-linear rewrite systems and so-called *simple goals* is close to our calculus CLNC in some respects. Simple CLN goals enjoy invariant properties similar to those required in K-LEAF [GLMP91] and in our Definition 7. Statements of the form $e \to^? X$ within a simple goal can be delayed in CLN, thus achieving the effect of laziness and sharing. On the other hand, statements of the form $X \to^? t$ in a simple CLN goal give rise to elimination of the variable X by propagation of the binding $X \mapsto t$. More details on CLN and related research can be found in the monograph [Pre98].

Our lazy narrowing calculus CLNC, being based on the rewriting logic CRWL, highlights the relevance of semantics. In particular, call-time choice appears as a semantic issue. CRWL proofs taken as witnesses give rise to a methodology for proving goal solving methods complete. This approach leads quite naturally to the design of different extensions of CRWL and CLNC, as we will see in Sections 5.3 and 5.4. Moreover, CRWL semantics helps to avoid some disturbing incompleteness phenomena that have been reported in the literature, especially for conditional narrowing with equational semantics. Variables that occur in the right-hand side or in the conditions of a rewrite rule, but not in the left-hand side, are called *extra variables.* Conditional narrowing is known to be complete for confluent and terminating conditional TRSs *without extra variables* [Kap87, MH94], and it was believed for some time that the same result was true for terminating conditional TRSs with extra variables in the conditions. Giovannetti and Moiso [GM86], however, found the following counterexample:

Example 12. Consider the conditional rewrite system $\mathcal{P}$ consisting of the three rewrite rules $a \to b$, $a \to c$ and $b \to c \Leftarrow X \approx b,\ X \approx c$. It is easy to show that $\mathcal{P}$ is confluent and strongly terminating. Moreover, the equations $a \approx b$, $a \approx c$ and $b \approx c$ clearly follow from $\mathcal{P}$ in equational logic. Nevertheless, conditional narrowing fails to solve the goal $b \approx c$; infinite narrowing derivations result instead, see [GM86, MH94].

In order to solve this problem, Giovannetti and Moiso proposed a strengthened notion of confluence, called *level confluence.* They proved completeness of conditional narrowing for level confluent and strongly terminating conditional TRSs with extra variables in the conditions only. By means of another counterexample, they showed also that the strong termination hypothesis cannot be dropped; see [GM86, MH94]. From the CRWL viewpoint, in Example 12 one should think of a, b as defined nullary functions, while c could be either a nullary function or a data constructor. In any case, the conditions within the third rewrite rule would become $X == b,\ X == c$, the goal would become $b == c$, and it is easily checked that $\mathcal{P} \vdash_{CRWL} b \to t$ can be proved only for $t = \bot$. Therefore, $\mathcal{P} \vdash_{CRWL} b == c$ *does not hold* anymore, so that Example 12 does not contradict our completeness Theorem 3 for CLNC.

Similar considerations can be made for other incompleteness examples presented in [GM86, MH94]. The point is that many artificial problems arise just because of the discrepancy between equational logic and the behaviour of functions in lazy programming languages, which is better described by CRWL. The introduction of cpo-semantics and strict equality in the languages K-LEAF [GLMP91] and BABEL [MNRA92] was motivated by this kind of difficulties.

5.2.5 Programming in the $\mathcal{TOY}$ System

$\mathcal{TOY}$ [LFSH99] is an experimental language and system that supports all aspects of CRWL programming explained up to this point, as well as most of the extensions that will be presented in Sections 5.3 and 5.4 below. A $\mathcal{TOY}$ program consists mainly of definitions for *functions* by means of CRWL-like rewrite rules. The concrete syntax is mostly borrowed from Haskell [Pe99], with the remarkable exception that variables begin with upper-case letters, while data constructors and function symbols must begin with lower-case letters. Rewrite rules in $\mathcal{TOY}$ programs are written in the form `<lhs> = <rhs> <== <cond>`, where the intended meaning of "=" is "$\to$", as in CRWL. In the rest of this paper, we will continue using the rewrite notation "$\to$", except for the concrete presentation of programs in $\mathcal{TOY}$ syntax.

A $\mathcal{TOY}$ program can also contain definitions for *infix operators*, *datatypes* and *type alias*. The extensions of CRWL dealing with types will be presented in Section 5.3. As predefined primitives, $\mathcal{TOY}$ offers booleans, characters, integers, reals, tuples and lists, as well as the most usual arithmetic and boolean operations. Strings are treated as lists of characters, as in Haskell. For the sake of user's convenience, $\mathcal{TOY}$ allows `if _ then _ else _` conditional expressions; they are treated as calls to a predefined function `if-then-else`, which can be easily programmed by rewrite rules. Conditions of the form `b == true` can be abbreviated as `b`. Also, rewrite rules of the form `lhs = true <== C` can be written as clauses `lhs :- C`. Here, the condition `C` cannot be empty, but it may be simply `true`. As another facility, $\mathcal{TOY}$ allows rewrite rules with multiple occurrences of variables in the left-hand side. Such rules are preprocessed, replacing repeated occurrences of variable `X`, say, by fresh variables `Xi`, and adding new conditions `X == Xi`. From now on, we will use these conventions when writing $\mathcal{TOY}$ and/or CRWL programs. For instance, the predicate for merging lists from Example 2 can be defined as follows in $\mathcal{TOY}$:

Example 13. A $\mathcal{TOY}$ predicate for merging lists:
```
merge([ ],Ys,Ys)          :- true
merge([X|Xs],[ ],[X|Xs]) :- true
merge([X|Xs],[Y|Ys],[X|Zs]) :- merge(Xs,[Y|Ys],Zs)
merge([X|Xs],[Y|Ys],[Y|Zs]) :- merge([X|Xs],Ys,Zs)
```

All the programming examples that have been mentioned up to this point, in particular the goals from Example 9, can be written in $\mathcal{TOY}$ syntax and executed. It is interesting to compare the $\mathcal{TOY}$ implementation of the two versions of *permutation sort* from Examples 6 and 7. Running both algorithms for sorting pseudo-randomly generated lists of integers of different lengths shows considerable speedups of the lazy generate-and-test version w.r.t. the naïve generate-and-test version. In general, significant improvements of lazy generate-and-test w.r.t. naïve generate-and-test can be expected for problems where the tester has good opportunities for rejecting candidate solutions on the basis of partial information.

The $\mathcal{TOY}$ system itself has been implemented in SICStus Prolog [Int99]. At its outer level, $\mathcal{TOY}$ behaves as a small command interpreter which must be executed within a SICStus Prolog session. The internal core of the $\mathcal{TOY}$ system, however, is a *compilation* process that translates the source $\mathcal{TOY}$ program $\mathcal{P}$ to a Prolog program $PT(\mathcal{P})$. For each defined function `f` occurring in $\mathcal{P}$, the rewrite rules defining `f` are translated into Prolog clauses for a Prolog predicate `#f`, whose rôle is to implement the computation of *head normal forms* (briefly, *hnfs*). More precisely, goals of the form `#f(`$\bar{t}_n$`,H)` will succeed in $PT(\mathcal{P})$ by binding the Prolog variable `H` to suitable Prolog representations of hnfs for the expression `f(`$\bar{t}_n$`)`. Note that hnfs are computed by lazy narrowing, possibly leading to several solutions with different bindings for the variables in `f(`$\bar{t}_n$`)`. Let us illustrate this by an example.

Example 14. Computation of hnfs.
Assume the program from Example 3 written as a $\mathcal{TOY}$ program $\mathcal{P}$. Then, $PT(\mathcal{P})$ is expected to compute the following solutions for `#merge(As,Bs,Hs)` (where `As`, `Bs` and `Hs` are variables):

1. {`As` $\mapsto$ `[ ]`, `Hs` $\mapsto$ `Bs`}.
2. {`As` $\mapsto$ `[X|Xs]`, `Bs` $\mapsto$ `[ ]`, `Hs` $\mapsto$ `[X|Xs]`}.
3. {`As` $\mapsto$ `[X|Xs]`, `Bs` $\mapsto$ `[Y|Ys]`,
   `Hs`$\mapsto$ `[X|susp(merge(Xs,[Y|Ys]),R,T)]`}.
4. {`As` $\mapsto$ `[X|Xs]`, `Bs` $\mapsto$ `[Y|Ys]`,
   `Hs`$\mapsto$ `[Y|susp(merge([X|Xs],Ys),R,T)]`}.

The term `susp(merge(Xs,[Y|Ys]),R,T)` occurring in the third item above is a Prolog representation of a *suspension*. In general, the Prolog translation $PT(\mathcal{P})$ of any $\mathcal{TOY}$ program $\mathcal{P}$ represents suspensions as terms of the form `susp(e,R,T)`. In $PT(\mathcal{P})$ there are clauses defining a Prolog predicate `hnf` that cooperates with all the predicates `#f` for computing hnfs of arbitrary expressions, in particular suspensions. Computing some hnf for a suspension `susp(e,R,T)` for the first time binds the *result* variable `R` to some result `h`, and also binds the `tag` variable `T` to the special term `on`. Later on, any attempt to compute a hnf for `susp(e,h,on)` will take `h` as result. In this way, the $\mathcal{TOY}$ system implements sharing, ensuring that call-time choice semantics is respected.

Rewrite rules for a function `f` in $\mathcal{P}$ can be compiled into clauses for a predicate `#f` in $PT(\mathcal{P})$ in various ways. A *naïve compilation method* proceeds as follows: a Prolog call `#f(`$\overline{X}_n$`,H)` repeatedly invokes the predicate `hnf` for unifying the arguments $\overline{X}_n$ with the left-hand side of some rewrite rule for `f` in $\mathcal{P}$, and then invokes `hnf` again for converting the right-hand side of the rewrite rule to hnf. In this way, each rewrite rule for `f` can be translated into a different clause for `#f`. Applying this method to the function `merge` from Example 3 the following Prolog clauses are obtained:

Example 15. Naïve Prolog compilation of `merge`.

```
   #merge(As,Bs,Hs)      :- hnf(As,[ ]), hnf(Bs,Hs).
#merge(As,Bs,[X|Xs]) :- hnf(As,[X|Xs]), hnf(Bs,[ ]).
#merge(As,Bs,[X|susp(merge(Xs,[Y|Ys]),R,T)]) :- hnf(As,[X|Xs]),
                                                 hnf(Bs,[Y|Ys]).
#merge(As,Bs,[Y|susp(merge([X|Xs],Ys),R,T)]) :- hnf(As,[X|Xs]),
                                                 hnf(Bs,[Y|Ys]).
```

This compilation scheme looks fine at the first sight. In particular, the choice among these four clauses by Prolog is in one-to-one correspondence with the choice among the four `merge` rewrite rules by the CLNC transformation (ON), which seems a pleasant property. Unfortunately, the execution of $\mathcal{TOY}$ programs compiled in this way under Prolog's computation strategy tends to perform too many redundant computations, or even to diverge, in cases where such a behaviour can and should be avoided. As a concrete example, let `e` be the expression `merge(single(n),void(n))`, where the functions `void` and `single` are defined as in Example 1, and `n` is some *very big* positive number, written in Peano notation. Clearly, `e` is a deterministic expression, whose value is the hnf `[z]`. When trying to solve the goal `#merge(single(n),void(n),H)`, Prolog will behave as follows: to start with, the first clause for `#merge` will fail, after an expensive computation of `[z]` as hnf of `single(n)`. Next, the second clause will succeed after repeating the previous hnf computation and performing another expensive computation of `[ ]` as hnf of `void(n)`. Next, if the user asks for alternative solutions, Prolog will unsuccesfully try the third and fourth clauses for `#merge`, whereby the hnfs of `single(n)` and `void(n)` will be computed twice more. As a second example, let `e'` be the expression `merge(single(N),void(N))`, for which the hnf value `[z]` can be obtained by infinitely many narrowing computations, binding the variable `N` to different Peano numbers. Now, the Prolog goal `#merge(single(N),void(N),H)` will diverge, because the first argument expression `single(N)` can be narrowed to the hnf `[z]` in infinitely many ways, with different bindings for `N`. Therefore, Prolog will never try beyond the first clause for `#merge`.

In contrast to the unfortunate behaviour we have just described, we would expect a more reasonable one, namely: for expression e, a computation of its unique hnf value `[z]` using the second rewrite rule of `merge`, and evaluating the two argument expressions `single(n)` and `void(n)` only one time each; and for expression `e'`, successive computation of solutions with different bindings of the variable `N`, using the second rewrite rule of `merge`. In fact, this improved behaviour is achieved by the $\mathcal{TOY}$ system, because the clauses for predicates `#f` are generated according to a so-called *demand driven strategy*, rather than using the naïve compilation method. The rough idea is as follows: `#f` will try *first* to compute a hnf for some argument expression which is demanded by all the rewrite rules of `f`, *then* consider the rules that are not in conflict with the resulting hnf, and *finally* apply them (if possible), or reiterate the process, otherwise.

Considering again the function `merge`, the first argument is *demanded* or *needed* by all the rewrite rules, in the sense that `merge(`$\perp$`,`t`) = `$\langle\perp\rangle$. Semantically, this is a *strictness* property; syntactically, it is easy to detect because of the presence of an outermost data constructor in the position of the first argument, in the left-hand sides of all rewrite rules. Therefore, in order to compute a hnf for `merge(as,bs)`, one must start by computing a hnf `h` for `as`. If `h` is `[ ]`, the first rewrite rule of `merge` can be applied. If `h` is of the form `[a|as']`, then the first rewrite rule of `merge` can be excluded. The three remaining rules demand the second argument of `merge`. Therefore, one must reiterate the process, by computing a hnf for `bs` and choosing a suitable `merge` rewrite rule according to the result. More precisely, the Prolog clauses for `#merge` generated by the $\mathcal{TOY}$ compiler in accordance to the demand driven strategy, are as follows:

Example 16. Demand-driven Prolog compilation of `merge`.
```
#merge(As,Bs,Hs)        :- hnf(As,HAs), #merge_1(HAs,Bs,Hs).
#merge_1([ ],Bs,Hs)     :- hnf(Bs,Hs).
#merge_1([X|Xs],Bs,Hs) :- hnf(Bs,HBs), #merge_1_2([X|Xs],HBs,H).
#merge_1_2([X|Xs],[ ],[X|Xs]).
#merge_1_2([X|Xs],[Y|Ys],[X|susp(merge(Xs,[Y|Ys]),R,T)]).
#merge_1_2([X|Xs],[Y|Ys],[Y|susp(merge([X|Xs],Ys),R,T)]).
```

If a $\mathcal{TOY}$ program $\mathcal{P}$ includes conditional rewrite rules for some function `f`, the Prolog clauses for the predicate `#f` will include calls of the form `eq(l,r)`, which implement the resolution of conditions `l == r`. The special predicate `eq` is defined by clauses which are part of any compiled program $PT(\mathcal{P})$. The clauses for `eq` use the predicate `hnf` and take care of decomposition, imitation and binding propagation processes, in accordance to the transformation rules of CLNC.

Once a $\mathcal{TOY}$ program is compiled, the user can propose goals. These are as the initial CRWL goals from subsection 5.2.4. The $\mathcal{TOY}$ system solves them internally by means of the Prolog predicate `eq`, and presents computed solutions $\sigma_S = \{X_1 \mapsto s_1, \dots, X_p \mapsto s_p\}$ by displaying strict equalities $X_i == s_i$. This is in accordance with Theorem 2 and item 2.(b) of Theorem 1, which guarantee correctness only for totally defined values of the variables occurring in σ_S. Alternative solutions, if existing, are obtained by Prolog's backtracking. For a $\mathcal{TOY}$ program containing all the function definitions from examples 1 and 3, the goals

$$\texttt{merge(single(n),void(n)) == R} and \texttt{merge}(\texttt{single}(\texttt{N}), \texttt{void}(\texttt{N})) == \texttt{R}$$

can be solved, producing the expected solutions for `R` and `N`. All the goals from Example 9 can be also solved in the expected way.

As explained above, $\mathcal{TOY}$'s demand driven strategy does not correspond directly to the CLNC transformation (ON) for *Outer Narrowing*. Nevertheless, the difference between CLNC and the demand driven strategy pertains to *control* rather than to *logic*. Within each successful computation, the demand driven strategy ultimately chooses the program rewrite rules to be applied. Therefore, all solutions computed by $\mathcal{TOY}$ can be associated to some succesful CLNC computation and are correct w.r.t. CRWL semantics, due to Theorem 2. Of course, Prolog's computation strategy (leftmost selection rule + depth-first exploration of the search space) can give rise to diverging computations. Hence, completeness of CLNC as guaranteed by Theorem 3 is not realized by the $\mathcal{TOY}$ implementation. This could be mended by adopting some fair search strategy, instead of depth-first search. In practice, however, a Prolog-like strategy works well enough for many problems, and there are useful programming techniques that can help to avoid diverging search in more problematic cases.

To close the presentation of $\mathcal{TOY}$, we will refer to some related work on lazy functional logic languages and their implementation techniques. S. Narain [Nar86] proposed a technique for doing lazy evaluation in logic, that could be used for translating lazy functional programs into Prolog. This pioneering work included already a device for implementing sharing with the help of logic variables. S. Antoy [Ant91, Ant92] improved Narain's work for the case of left-linear, constructor-based TRSs. He introduced so-called *definitional trees* as a tool for achieving a reduction strategy which avoids unneeded reductions. P.H. Cheong and L. Fribourg [CF93] used techniques similar to those of Narain for the compilation of lazy narrowing into Prolog, aiming at the implementation of languages such as K-LEAF [GLMP91]. This work followed the naïve compilation method, whose disadventages have been discussed above. The *demand driven strategy* used by the $\mathcal{TOY}$ system was introduced in [LLFRA93], using Antoy's definitional trees for narrowing, instead of rewriting. The class of programs assumed in [LLFRA93] was equivalent to deterministic CRWL programs. Non-deterministic functions, however, pose no additional problem, provided that a sharing mechanism is implemented. The demand driven strategy is equivalent to *inductively sequential narrowing* [Ant97], an extension of *needed narrowing* [AEM94] already mentioned in Subsection 5.2.4 above. Both needed narrowing and inductively sequential narrowing were originally defined with the help of definitional trees. A Prolog implementation of needed narrowing for inductively sequential systems was presented by M. Hanus in [Han95], using definitional trees as in [LLFRA93] and modelling strict equality as a function $\doteq$ defined by rewrite rules. A novelty in this paper was the use of simplification rewrite rules to optimize deterministic computations, in the line of [Han94a].

Lazy narrowing is the main execution mechanism for non-strict functional logic languages, for which theoretical completeness results are available. Some functional logic languages, however, have chosen an alternative computation model, known as *residuation* [AKLN87, Han94b], which delays function calls including logic variables until they are ready for deterministic evaluation, because the variables have got bindings from the resolution of some other part of the goal. This mechanism is incomplete in general, but it tends to avoid search, and it supports a concurrent style of programming. The residuation idea, understood in a broad sense, plays an important rôle in concurrent-constraint languages such as Oz [Smo95]. A computation model which combines needed narrowing and residuation has been proposed by Hanus [Han97] and is used by the integrated functional and logic language Curry [H$^+$00]. From a user's viewpoint, Curry and $\mathcal{TOY}$ have many common features. A method for compiling Curry programs into Prolog has been described in [AH00]. This paper extends the compilation methods from [LLFRA93, Han95] with new mechanisms for performing concurrent evaluation steps, which are required to deal with residuation.

Another recent proposal for the integration of functional and logic programming is the language Escher [Llo95, Llo99], proposed by Lloyd as an attempt to combine the best ideas for functional languages such as Haskell [Pe99] and logic languages such as Gödel [HL94]. In Escher, the execution model is rewriting, rather than narrowing. Computation of alternative answers must be expressed explicitely by means of disjunctions. Escher's reduction strategy relies on predefined meta-rewrite rules, and it is intended to preserve the semantics of expressions in accordance to an extension of Church's simple theory of types [Chu40]. Each terminating computation stops with some final expression which is equivalent to the initial one, but no theoretical result ensures that this final expression always belongs to some well defined class of "values", as users of a programming language would expect.

5.3 Higher-Order Programming

Most functional programming languages are higher-order and have a type system, requiring programs to be well-typed at compilation time. In this section we will show that the CRWL approach to functional logic programming, as well as its realization in the system $\mathcal{TOY}$, can be naturally extended with such features. We will use the abbreviations "HO" and "FO" for "Higher-Order" and "First-Order", respectively.

5.3.1 Programming with Applicative Rewrite Systems

HO functional languages treat functions as first-class values that can be passed as arguments to other functions, and returned as results. This leads naturally to a modification in the syntax of expressions. Assuming a signature Σ and a set of variables $\mathcal{X}$ as in Subsection 5.2.1, *partial expressions* over $\mathcal{X}$ (notation: $e \in Exp_{\Sigma_\perp}(\mathcal{X})$) are now defined as follows:

$$e ::= X \quad (X \in \mathcal{X}) \mid \perp \mid h \quad (h \in DC \cup FS) \mid (e\ e_1)$$

These expressions are usually called *applicative*, because $(e\ e_1)$ stands for the *application* operation (represented as juxtaposition) which applies the function denoted by e to the argument denoted by e_1. Applicative expressions without occurrences of $\perp$ will be called *total*, as in the FO case. Following a usual convention, we will assume that application associates to the left, and omit unneeded brackets accordingly. For instance, we can abbreviate `((merge Xs) Ys)` as `(merge Xs Ys)`, or even as `merge Xs Ys`. This so-called *curried notation* can be understood as an alternative to the expression `merge(Xs,Ys)` in FO syntax. In the curried notation, `merge` is viewed as a function which is applied to `Xs`, returning another function which is then applied to `Ys`, while the FO notation views `merge` as a function which must be applied to a pair `(Xs,Ys)`. The idea to use curried notation instead of tuples stems from the logicians H.B. Curry and M. Schönfinkel. In the sequel, we will work with applicative expressions in curried notation. However, we will allow n-tuples $(e_1, \ldots, e_n)$, viewing them as a shorthand for $(tup_n\ e_1 \ldots\ e_n)$, where tup_n is a special data constructor. Thanks to tuples, the FO syntax from Section 5.2 is subssumed by the current HO syntax. We will also allow infix operators as syntactic sugar, viewing expressions of the form $(e_1 \oplus e_2)$ as a shorthand for $((\oplus)\ e_1\ e_2)$. Be careful not to confuse the syntax for tuples, as in (X, Y, Z), with that for function application, as in $(f\ X\ Y)$.

Most functional languages allow also expressions of the form $\lambda X.e$, called λ-*abstractions*. This syntax is borrowed from the λ-calculus [HS86] and stands for an anonymous function which, when given any actual parameter in place of the *formal parameter* X, will return the value resulting from the evaluation of the *body* expression e. Some approaches to HO functional logic programming [Pre94, Pre95, Pre98, HP99b, ALS94, SNI97, MIS99] use λ-abstractions (more precisely, *simply typed* λ-expressions). The HO logic programming language λ-Prolog [MN86, NM88] also uses simply typed λ-terms as a representation of data values, although the definition of evaluable functions is not supported. Working in the simply typed λ-calculus has the disadventage that unification is undecidable in general [SG89]. Therefore, the use of λ-terms in a practical programming language must be restricted somehow. In purely functional languages, the problem is easy to solve because λ-expressions are used only for evaluation, never for unification or matching. Some decidable subcases of HO unification are known [Mil91,Pre94,Pre98]. Syntactic restrictions on the form of programs and goals to guarantee decidable unification problems during goal solving are also known [ALS94]. However, applicative expressions without λ-abstractions are expressive enough for many purposes. In the sequel we will restrict ourselves to this case, following the approach from [GMHGRA97].

In a HO setting with applicative expressions, the syntax of *partial data terms* over $\mathcal{X}$ (notation: $t \in Term_{\Sigma_\perp}(\mathcal{X})$) becomes curried:

$$t ::= X \quad (X \in \mathcal{X}) \mid \perp \mid (c\ \overline{t}_n) \quad (c \in DC^n)$$

where $(c\ \overline{t}_n)$ abbreviates $(c\ t_1 \ldots\ t_n)$, which reduces to c in the case $n = 0$. For instance, using the data constructors `z` and `s` for Peano numbers as in Example 1, we must now represent number 2 as `(s (s z))`. In the case of lists, we will keep the notation `[X|Xs]`, understood as a shorthand for `(cons X Xs)`.

In what follows, we abbreviate $(e\ e_1 \ldots e_n)$ as $(e\ \overline{e}_n)$. Due to the curried notation, we can build *partial applications* $(f\ \overline{e}_m)$ with $f \in FS^n$, $m < n$. Partial applications $(c\ \overline{e}_m)$ with $c \in DC^n$ and $m < n$ are also possible. Combining data constructors and partial applications gives rise to *partial patterns* over any given set of variables $\mathcal{X}$ (notation: $t \in Pat_{\Sigma_\bot}(\mathcal{X})$), whose syntax is defined as follows:

$$\begin{aligned} t ::= X \quad (X \in \mathcal{X}) \mid \bot \mid (c\ \overline{t}_m) \quad (c \in DC^n,\ m \leq n) \\ \mid (f\ \overline{t}_m) \quad (f \in FS^n,\ m < n) \end{aligned}$$

Total patterns $t \in Pat_\Sigma(\mathcal{X})$ are defined similarly, omitting occurrences of $\bot$. Note that $Term_{\Sigma_\bot}(\mathcal{X}) \subset Pat_{\Sigma_\bot}(\mathcal{X}) \subset Exp_{\Sigma_\bot}(\mathcal{X})$. Therefore, patterns can be viewed as a generalization of data terms, and we will use them in place of data terms for most purposes. Patterns of the form $(f\ \overline{t}_m)$ $(f \in FS^n,\ m < n)$ are not data terms and can be used as an intensional representation of functions, as we will see later on. On the other hand, $(f\ \overline{t}_m)$ is not a pattern whenever $f \in FS^n$ and $m \geq n$.

Substitutions $\theta \in Subst_\Sigma(\mathcal{X}, \mathcal{Y})$ will be now mappings $\theta : \mathcal{X} \to Pat_\Sigma(\mathcal{Y})$, and analogously for partial substitutions. Moreover, we will define functions $f \in FS^n$ by left-linear, pattern-based applicative rewrite rules of the form $f\ \overline{t}_n \to r \Leftarrow C$, with $t_i \in Pat_\Sigma(\mathcal{X})$, and a condition C composed of joinability statements $l == r$ with $l,\ r \in Exp_\Sigma(\mathcal{X})$. This includes FO programs as a particular case, because tuples of expressions are expressions. Since the rewrite rules for $f \in FS^n$ expect n arguments, partial applications $(f\ \overline{e}_m)$ are irreducible at the outermost position. Hence, *head normal forms* (hnfs) in our HO setting are variables X and expressions with the outermost appearence of a pattern, i.e: $(c\ \overline{e}_m)$, where $c \in DC^n$, $m \leq n$; or $(f\ \overline{e}_m)$, where $f \in DC^m$, $m < n$.

Regarding types, we will adopt Milner's type system [Mil78,DM82], which is used by most current functional languages. We consider *typed signatures* given as triples $\Sigma = \langle TC, DC, FS \rangle$, where $DC = \bigcup_{n\in\mathbb{N}} DC^n$ resp. $FS = \bigcup_{n\in\mathbb{N}} FS^n$ are ranked sets of *data constructors* resp. *defined function symbols*, as before, and $TC = \bigcup_{n\in\mathbb{N}} TC^n$ is a ranked set of *type constructors*. For each $K \in TC^n$, $n = ar(K)$ is called the *arity* of K. Analogously, we can speak of arities $n = ar(c)$ resp. $n = ar(f)$ for $c \in DC^n$ resp. $f \in FS^n$. We will assume that countably many *type variables* (noted as A, B, C, etc.) are available. Given any set $\mathcal{A}$ of type variables, the syntax of *types* over $\mathcal{A}$ (notation: $\tau \in Type_\Sigma(\mathcal{A})$) is:

$$\tau ::= A \quad (A \in \mathcal{A}) \mid (K\ \tau_1 \ldots\ \tau_n) \quad (K \in TC^n) \mid (\tau_1 \to \tau)$$

We will abbreviate $(K\ \tau_1 \ldots\ \tau_n)$ as $(K\ \overline{\tau}_n)$. A type of this form, called a *constructed type*, reduces to K if $n = 0$. *Functional types* $(\tau_1 \to \tau)$ correspond to functions that expect arguments of type τ_1 and return results of type τ. We will follow the convention that "$\to$" associates to the right within types, and we will write $\overline{\tau}_n \to \tau$ as abbreviation for a type of the form $\tau_1 \to \cdots \to \tau_n \to \tau$. We will also assume special type constructors $prod_n \in TC^n$ and write $(\tau_1, \ldots, \tau_n)$ as a shorthand for $(prod_n\ \tau_1 \ldots\ \tau_n)$.

This will represent the *cartesian product* of the types τ_i, i.e., the type of tuples $(e_1, \ldots, e_n)$ where each e_i has type τ_i. Types without any occurrence of $\rightarrow$ will be called *datatypes*. The set of all type variables occurring in a type τ will be noted as $tvar(\tau)$. A type τ will be called *monomorphic* iff $tvar(\tau) = \emptyset$, and *polymorphic* otherwise. A global universal quantification of type variables is implicit in polymorphic types. This idea is related to the instantiation (or particularization) of types by the application of *type substitutions* $\kappa : \mathcal{A} \rightarrow Type_\Sigma(\mathcal{B})$. The set of all type substitutions of this kind will be noted as $TSubst_\Sigma(\mathcal{A}, \mathcal{B})$.

We will assume that each signature Σ includes type declarations for all the data constructors and function symbols. More precisely, we require

- Each $c \in DC^n$ comes with a type declaration $c :: \overline{\tau}_n \rightarrow \tau$, where τ_i, τ are datatypes, τ is not a variable, and $tvar(\overline{\tau}_n) \subseteq tvar(\tau)$. This last property, called *transparency*, ensures that data built by any data constructor bear full information about the types of the arguments.
- Each $f \in FS^n$ comes with a type declaration $f :: \overline{\tau}_n \rightarrow \tau$, where τ_i, τ are arbitrary types.

For signatures $\Sigma_\perp$ extended with $\perp$, we will asume the type declaration $\perp :: A$, meaning that $\perp$ belongs to every type. Given a type signature, we still need assumptions on the types of variables for being able to define well-typed expressions. Any set $\mathcal{V}$ of type assumptions $X :: \tau$ including exactly one occurrence of each variable $X \in \mathcal{X}$ will be called a *type environment* for the set of variables $\mathcal{X}$. A *type judgement* $\mathcal{V} \vdash_\Sigma e :: \tau$ is intended to mean that the expression $e \in Exp_{\Sigma_\perp}(\mathcal{X})$ has type τ under the type assumptions in $\mathcal{V}$. We will use the following inference rules for deriving type judgements:

Definition 10. Inference rules for type judgements.

VR Variable: $\mathcal{V} \vdash_\Sigma X :: \tau$ if $X :: \tau \in \mathcal{V}$.
ID Identifier: $\mathcal{V} \vdash_\Sigma h :: \tau\kappa$ if $h \in DC \cup FS$, $h :: \tau$ declared in Σ, κ type substitution.
AP Application: $\dfrac{\mathcal{V} \vdash_\Sigma e :: \tau_1 \rightarrow \tau \qquad \mathcal{V} \vdash_\Sigma e_1 :: \tau_1}{\mathcal{V} \vdash_\Sigma (e\ e_1) :: \tau}$ □

In the sequel, $Exp^\tau_\Sigma(\mathcal{X})$ will denote the set of all expressions $e \in Exp_\Sigma(\mathcal{X})$ such that $\mathcal{V} \vdash_\Sigma e :: \tau$. Similar notations will be used for partial expressions, patterns, etc. An expression e is called *well-typed* in the type environment $\mathcal{V}$ iff $\mathcal{V} \vdash_\Sigma e :: \tau$ for some type τ, and *ill-typed* otherwise. Due to the inference rule (ID), a well-typed expression can have more than one type. For every expression e which is well-typed in $\mathcal{V}$ there is a so-called *principal type* τ with the following properties:

- $\mathcal{V} \vdash_\Sigma e :: \tau$.
- For any other τ' such that $\mathcal{V} \vdash_\Sigma e :: \tau'$ there is some type substitution κ such that $\tau' = \tau\kappa$.

From now on we will be mainly interested in well-typed expressions. Sometimes we will use the notation $e :: \tau$ to indicate that e has type τ in some implicit environment. The existence of principal types was proved by Milner and Damas [Mil78, DM82]. They also gave a *type inference algorithm* which can decide wether e is well-typed in $\mathcal{V}$ or not, computing a principal type in the positive case.

We are now in a position to present some programming examples. We will use data constructors for Peano numbers and lists, assuming the following type declarations:

```
z :: nat              [ ]     :: [A]
s :: nat → nat        [· | ·] :: A → [A] → [A]
```

Of course, `nat` is assumed to be a nullary type constructor. Moreover, we have used the notation `[A]` as a convenient shorthand for `(list A)`, where `list` is a unary type constructor. The next example shows some typical HO functions. Their definitions includes a *type declaration* in addition to the rewrite rules.

Example 17. Typical HO functions.

```
(·) :: (B → C) → (A → B) → A → C
(·) F G X → F (G X)

twice :: (A → A) → A → A
twice F X → F (F X)

filter :: (A → bool) → [A] → [A]
filter P [ ] → [ ]
filter P [X|Xs] → if P X then [X|filter P Xs] else filter P Xs

map :: (A → B) → [A] → [B]
map F [ ] → [ ]
map F [X|Xs] → [F X|map F Xs]
```

We will write `((·) F G)` also as `(F · G)`; it behaves as the composition of the functions `F` and `G`. Function `twice` is such that `(twice F)` has the same behaviour as `(F · F)` Functions `map` and `filter` are related to list processing. The effect of `(map F Xs)` is to apply the function `F` to each element of the list `Xs`, returning a new list as result, while `(filter P Xs)` returns the list formed by those elements of `Xs` which satisfy the predicate `P`.

Recall the function `double` from Example 4, and observe that the expression `(twice twice double (s z))` (whose value is the number 16 in Peano notation), makes sense in spite of the fact that the arity of `twice` is less than 3. More generally, expressions of the form $(f\ \overline{e}_m)$ can make sense for $m > ar(f)$. Note also that syntactically different patterns can sometimes represent the same function, as in the case of `twice double` and `(double . double)`. The HO extension of CRWL to be presented in Subsection 5.3.3 will be unable to deduce $t == t'$ whenever t and t' are syntactically different patterns. For this reason, we will say that patterns provide an *intensional* representation of functions in our setting. The usefulness of this device is illustrated in the next example, where `circuit` is used as an alias for the type `(bool,bool,bool)` $\to$ `bool`. Functions of this type are intended to represent simple circuits which receive three boolean inputs and return a boolean output. We assume that the boolean functions `not`, `/\` and `\/` have been defined elsewhere.

Example 18. Simple circuits.

```
x1, x2, x3 :: circuit
x1(X1,X2,X3) → X1     x2(X1,X2,X3) → X2     x3(X1,X2,X3) → X3
   notGate :: circuit → circuit
notGate C (X1,X2,X3) → not(C(X1,X2,X3))

andGate :: circuit → circuit → circuit
andGate C1 C2 (X1,X2,X3) → C1(X1,X2,X3) /\ C2(X1,X2,X3)

orGate :: circuit → circuit → circuit
orGate C1 C2 (X1,X2,X3) → C1(X1,X2,X3) \/ C2(X1,X2,X3)
```

Functions `x1`, `x2` and `x3` represent the basic circuits which just copy one of the inputs to the output. More interestingly, the HO functions `notGate`, `andGate` and `orGate` take circuits as parameters and build new circuits, corresponding to the logical gates `NOT`, `AND` and `OR`. Imagine that we are interested in a circuit whose output coincides with the majority of its three inputs. Using λ-notation, a circuit with this desired behaviour can be written as follows:

$$\lambda X1.\ \lambda X2.\ \lambda X3.(((X1\ /\backslash\ X3)\ \backslash/\ X2)\ /\backslash\ (X1\ \backslash/\ X3))$$

In our λ-free setting, an intensionally different circuit with the same behaviour can be represented as a pattern:

```
(andGate (orGate (andGate x1 x3) x2) (orGate x1 x3))
```

More generally, many patterns in the signature of this example are useful representations of circuits. We will further ellaborate the example in Section 5.3.4

Programs in Examples 17 and 18 are well-typed (even *strongly* well-typed) in the following sense:

Definition 11. Well-typed Programs.
Assume a program $\mathcal{P}$ with typed signature Σ.

1. Let C be a compound joinability condition with variables in $\mathcal{X}$, and let $\mathcal{V}$ be a type environment for $\mathcal{X}$. By convention, we will write $\mathcal{V} \vdash_\Sigma C$ iff for each $l == r \in C$ there is some type τ such that $\mathcal{V} \vdash_\Sigma l :: \tau$ and $\mathcal{V} \vdash_\Sigma r :: \tau$.
2. Let $f :: \overline{\tau}_n \to \tau$ be a type declaration in Σ. A rewrite rule $f\ \overline{t}_n \to r \Leftarrow C$ using variables in $\mathcal{X}$ is called well-typed iff there is some type environment $\mathcal{V}$ for $\mathcal{X}$ such that $\mathcal{V} \vdash_\Sigma t_i :: \tau_i$ for all $1 \le i \le n$, $\mathcal{V} \vdash_\Sigma r :: \tau$ and $\mathcal{V} \vdash_\Sigma C$.
3. $\mathcal{P}$ is called well-typed iff all the rewrite rules belonging to $\mathcal{P}$ are well-typed.
4. $\mathcal{P}$ is called *strongly well-typed* iff $\mathcal{P}$ is well-typed and $var(r) \subseteq var(l)$ holds for every rewrite rule $l \to r \Leftarrow C\ \in \mathcal{P}$. □

In the rest of this paper we will be mainly interested in well-typed rewrite rules and programs. Given the type declarations for data constructors and the rewrite rules of a program, the type inference algorithm from [Mil78, DM82] can be adapted to decide whether the program is well-typed, and to compute principal types for all the defined functions in the positive case. Therefore, the type declarations for defined functions do not have to be declared by the programmer. We have included them as part of typed signatures just for technical convenience.

5.3.2 Algebraic Datatypes

Datatypes in current functional languages are *free*, in the sense that syntactically different data terms are viewed as descriptions of different data values. In some cases, however, this may be not convenient. Consider, for instance, the following type declarations, intended to define datatypes for polymorphic sets and multisets:

$\{\ \}\ :: \{\mathtt{A}\}$ $\quad\quad$ $\{\!\!\{\ \}\!\!\}\ :: \{\!\!\{\mathtt{A}\}\!\!\}$

$\{\cdot \mid \cdot\}\ :: \mathtt{A} \to \{\mathtt{A}\} \to \{\mathtt{A}\}$ $\quad\quad$ $\{\!\!\{\cdot \mid \cdot\}\!\!\}\ :: \mathtt{A} \to \{\!\!\{\mathtt{A}\}\!\!\} \to \{\!\!\{\mathtt{A}\}\!\!\}$

Here we are proceeding by analogy with the case of lists: $\{A\}$ represents the type of all sets with elements of type A; $\{\ \}$ represents the empty set; $\{X \mid Xs\}$ represents the set obtained by adding X as a new element to the set Xs; and similarly for multisets. In contrast to lists, some non-trivial equivalences between data terms are expected to hold for sets and multisets. In what follows, we use $\{X,\ Y \mid Zs\}$ as an abbreviation for $\{X \mid \{Y \mid Zs\}\}$, and analogously for the multiset case. We will use the brackets $''\{''$ and $''\}''$ both in the object language and in the meta language, hoping that no serious ambiguities will arise.

Example 19. Axioms for Sets and Multisets.

1. Sets are expected to satisfy the two axioms:
   $\{X,\ Y \mid Zs\} \simeq \{Y,\ X \mid Zs\}$ $\quad\quad$ $\{X \mid Zs\} \simeq \{X,\ X \mid Zs\}$
2. Multisets are expected to satisfy the axiom:
   $\{\!\!\{X,\ Y \mid Zs\}\!\!\} \simeq \{\!\!\{Y,\ X \mid Zs\}\!\!\}$

Following [ASRA97b], we will allow axioms for data constructors in our programs. Formally, this leads to a new enrichment of typed signatures. They become now quadruples $\Sigma = \langle TC, DC, \mathcal{C}, FS\rangle$, where TC, DC, FS are as before, and $\mathcal{C}$ is a set of so-called *data axioms*, of the form $s \simeq t$, where s, t are data terms built from variables and data constructors from DC. The following properties of data axioms will be important for us:

Definition 12. Properties of data axioms.
A data axiom $t \simeq s$ given within a typed signature Σ is called

1. *Regular* iff $var(t) = var(s)$.
2. *Non-collapsing* iff neither t nor s is a variable.
3. *Strongly regular* iff it is regular and non-collapsing.

4. *Well-typed* iff it is strongly regular, of the form $(c\ \overline{t}_n) \simeq (d\ \overline{s}_m)$, and there are variants $c :: \overline{\tau}_n \rightarrow \tau$, $d :: \overline{\tau'}_m \rightarrow \tau$ of the type declarations for c, d in Σ and some type environment $\mathcal{V}$ such that $\mathcal{V} \vdash_\Sigma (c\ \overline{t}_n) :: \tau$ and $\mathcal{V} \vdash_\Sigma (d\ \overline{s}_m) :: \tau$. □

When speaking of a data axiom $t \simeq s$ we will mean, by an abuse of language, either $t \simeq s$ or $s \simeq t$. With respect to a given signature Σ, we will say that a data constructor c is *algebraic* iff Σ includes some data axiom of the form $(c\ \overline{t}_n) \simeq s$. Otherwise, we will say that c is *free*. A datatype τ will be called *algebraic* iff there is some algebraic data constructor returning values of type τ, and free otherwise. For instance, $\{A\}$ and $\{\!\!\{A\}\!\!\}$ are algebraic, polymorphic datatypes. The corresponding data axioms, shown in Example 19, are strongly regular and well-typed. In the rest of this paper we will always assume strongly regular and well-typed data axioms. These assumptioms are needed for our current results on semantics [ASRA97b] and they are sufficient to define a wide class of interesting algebraic datatypes. Note that the assumptions are not imposed to the *rewrite rules* used to define functions in a program. For instance, assuming `pred` $\in FS^1$ and `suc` $\in DC^1$, the rewrite rule `pred (suc X)` $\rightarrow$ `X` can appear in a program.

The intended meaning of a data axiom $t \simeq s$ is not exactly equality. Our aim is to use the data axioms $\mathcal{C}$ in our current signature for refining the approximation ordering $\sqsubseteq$, yielding a finer monotonic ordering $\sqsubseteq_\mathcal{C}$ over partial patterns.

Definition 13. Approximation ordering modulo $\mathcal{C}$.
Assume a typed signature Σ and a set of variables $\mathcal{X}$.

1. We say that a substitution $\theta \in Subst_{\Sigma_\perp}(\mathcal{Y}, \mathcal{X})$ is *safe* for a data term t iff $\theta(Y)$ is total for every variable Y which has more than one occurrence in t.
2. We define $\sqsubseteq_\mathcal{C}$ as the least preorder (i.e., reflexive and transitive relation) over $Pat_{\Sigma_\perp}(\mathcal{X})$ which satisfies:
   (a) $\perp \sqsubseteq_\mathcal{C} t$ for all $t \in Pat_{\Sigma_\perp}(\mathcal{X})$.
   (b) $t \sqsubseteq_\mathcal{C} t'$, $t_1 \sqsubseteq_\mathcal{C} t'_1 \Longrightarrow (t\ t_1) \sqsubseteq_\mathcal{C} (t'\ t'_1)$.
   (c) $s\theta \sqsubseteq_\mathcal{C} t\theta$ for every data axiom $s \simeq t \in \mathcal{C}$ with variables in $\mathcal{Y}$, and for every $\theta \in Subst_{\Sigma_\perp}(\mathcal{Y}, \mathcal{X})$ which is safe for t.
3. For all t, $t' \in Pat_{\Sigma_\perp}(\mathcal{Y})$, we define: $t \cong_\mathcal{C} t'$ iff $t \sqsubseteq_\mathcal{C} t'$ and $t' \sqsubseteq_\mathcal{C} t$. □

For a given data axiom $s \simeq t \in \mathcal{C}$, we accept $s\theta \sqsubseteq_\mathcal{C} t\theta$ only if θ is safe for t. For instance, from the data axiom $\{X \mid Zs\} \simeq \{X,\ X \mid Zs\}$ for sets, we will accept $\{z\} \sqsubseteq_\mathcal{C} \{z,\ z\}$, which comes from the safe substitution $\{X \mapsto z,\ Zs \mapsto \{\ \}\}$, but we will not accept $\{\perp\} \sqsubseteq_\mathcal{C} \{\perp,\ \perp\}$, coming from the unsafe substitution $\{X \mapsto \perp,\ Zs \mapsto \{\ \}\}$. In fact, accepting $\{\perp\} \sqsubseteq_\mathcal{C} \{\perp,\ \perp\}$ is counterintuitive, since the information "singleton set" is present in $\{\perp\}$ but missing in $\{\perp,\ \perp\}$. The following properties of $\sqsubseteq_\mathcal{C}$ and $\cong_\mathcal{C}$ are easy to check:

Proposition 4. *For every typed signature Σ and set of variables $\mathcal{X}$, we have:*

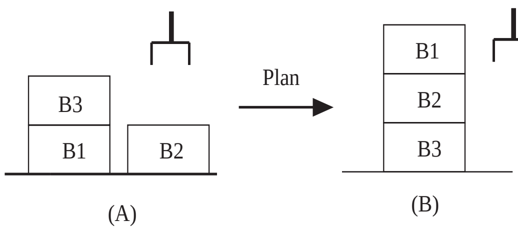
1. *$\sqsubseteq_{\mathcal{C}}$ is preorder over $Pat_{\Sigma_\perp}(\mathcal{X})$ compatible with application, and $\cong_{\mathcal{C}}$ is the least congruence (i.e., equivalence relation and compatible with application) which extends $\sqsubseteq_{\mathcal{C}}$. The equivalence class of a pattern t w.r.t. $\cong_{\mathcal{C}}$ will be noted as $[t]_{\mathcal{C}}$.*
2. *Assume that all the data axioms in $\mathcal{C}$ are regular. Then, if $s \sqsubseteq_{\mathcal{C}} t$ and s is total, we can conclude that t is also total and such that $s \cong_{\mathcal{C}} t$.* ■

The regularity hypothesis in item 2. above cannot be omitted. For instance, given the irregular data axiom $(c\, X) \simeq (d\, Y)$, and assuming a nullary constructor k, we get $(c\, k) \sqsubseteq_{\mathcal{C}} (d\, \perp)$.

As we will see in the next subsection, delayed unification modulo $\mathcal{C}$ is used for the execution of CRWL programs. For this reason, algebraic datatypes allow to write more clear and concise programs in many cases. In particular, the datatype multiset is well-suited for many applications. One significant case is the General Abstract Model for Multiset Manipulation (GAMMA, for short), proposed by Banâtre and Le Métayer [BM90, BM93] as a high level model for programming, taking multisets as the basic data structure. In [ASRAar] we have shown that GAMMA programs can be emulated in our setting.

Another interesting area where multisets play a useful rôle is that of *action and change problems*, where one is interested in finding actions to transform a given initial situation into a final situation which satisfies some desired property. When attempting to solve action and change problems in classical predicate logic, one meets the so-called *frame problem*, roughly meaning that all the properties of a situation that are not affected by the application of an action, must be explicitly asserted within the logical formula which formalizes the effect of the action. This gives rise to a combinatorial explosion when trying to use automated deduction techniques such as resolution. Among other approaches [MM99], the representation of situations and multisets of facts, and actions as multiset transformations, helps to deal with action and change while avoiding the frame problem. Following this idea, the authors of [HS90] have developed an approach to *planning problems* (a particular case of action and change problems) based on *equational logic programs*. In this setting, programs consist of Horn clauses using algebraic data constructors, in addition to free data constructors. SLD resolution uses unification modulo the equational theory of the algebraic constructors present in the program. More precisely, the approach to planning problems in [HS90] proposes to use a binary associative-commutative constructor $\circ$ (written in infix notation) to represent situations as multisets of facts $\mathsf{Fact}_1 \circ \ldots \circ \mathsf{Fact}_n$, and a ternary predicate `execPlan` to model the transformation of an initial situation into a final situation by the execution of a plan.

In our setting, we can choose to represent situations as multisets of facts, plans as lists of actions, and we can use a function for modelling the transformation of a situation according to a given plan. More precisely, we can consider the types `situation` and `plan` as alias for $\{\!\!\{$ `fact` $\}\!\!\}$ and `[action]`, respectively, where the datatypes `fact` and `action` must be properly defined for each particular planning problem. As a concrete illustration, we will consider a blocksworld where several actions are available for a robot hand to move blocks. We are interesting in finding a plan for transforming the initial situation (A) into the final situation (B):



In this particular problem, we need datatypes `block`, `fact` and `action`, with the following data constructors:

```
b1 :: block   b2 :: block   b3 :: block

empty :: fact
clear, table, hand :: block → fact
over :: block → block → fact

pickup, putdown :: block → action
stack,  unstack :: block → block → action
```

The intended meaning of facts and actions should be clear. The facts assert that the hand is empty, that a block has no other block over it, or is on the table, or is held by the hand, or is placed over another block. Actions `pickup` and `putdown` refer to a block placed on the table, while `stack` and `unstack` refer to a block placed over another block. The initial and final situations, as well as the functions needed to model plan execution can be defined as follows. Beware that multiple occurrences of variables in left-hand sides must be understood as explained in Subsection 5.2.5.

Example 20. Planning in a Block's World.

```
ini, fin :: situation

ini → {[clear b2, clear b3, over b3 b1, table b2, table b1,
        empty]}
fin → {[clear b1, over b1 b2, over b2 b3, table b3, empty]}
execPlan         :: plan → situation → situation

execPlan [ ]    S → S
execPlan [A|As] S → execPlan As (execAction A S)

execAction :: action → situation → situation
execAction (pickup B) {[table B, clear B, empty | S]}      →
           {[hand B | S]}
execAction (putdown B) {[hand B | S]}                      →
           {[table B, clear B, empty | S]}
execAction (stack B C) {[hand B, clear C | S]}             →
           {[over B C, clear B, empty | S]}
execAction (unstack B C) {[over B C, clear B, empty | S]} →
           {[hand B, clear C | S]}
```

Now, the plan we were looking for can be computed as a solution of the goal `execPlan Plan ini == fin`. There is a solution of length 6 for `Plan`, namely:

```
[unstack b3 b1, putdown b3, pickup b2, stack b2 b3,
             pickup b1, stack b1 b2]
```

as well as solutions of greater length.

5.3.3 Extending CRWL and CLNC with HO Functions and Types

Semantics and goal solving for CRWL programs in the untyped FO case have been discussed in Subsections 5.2.2, 5.2.3 and 5.2.4. Now we will explain more briefly how a similar development can be achieved for the typed HO applicative programs studied in the last subsection. Our main motivation is to stay close to FO semantics, in order to avoid undecidability problems, especially undecidability of HO unification [SG89]. Therefore, we will start with a presentation of some known techniques, intended to "reduce HO to FO" in some sense.

Following a suggestion of Warren [War82], some people have asked themselves wether "HO extensions are needed". In fact, [War82] gave a method for translating HO programs into FO programs. Roughly, the idea consists in introducing an explicit application operation `apply` to be used in FO notation, replacing λ-abstractions (or similar constructs, such as partial applications) by means of new data constructors, and providing rewrite rules (Warren thought of Horn clauses) to define the proper behaviour of the application operation when meeting terms where these new data constructors appear. For instance, the partial application `(map F)` would be translated as `map1(F)` by means of a new data constructor `map1`, and the following rewrite rule (among others) would be provided for the application operation: `apply(map1(F),Xs)` $\to$ `map(F,Xs)`. Of course, the rewrite rules for `map` would be also translated into FO syntax. The second one (see Example 17) would become `map(F,[X|Xs])` $\to$ `[apply(F,X)|map(F,Xs)]`.

Originally, Warren's motivation was to demonstrate that HO programming techniques could be used in Prolog. However, "HO to FO" translations in Warren's spirit can be used also to define the semantics of HO programming languages. For instance, [BG86] presented IDEAL, a HO extension of the language K-LEAF [GLMP91], whose semantics was defined by a translation of IDEAL programs to K-LEAF programs. Unfortunately, the translation approach does not solve all problems. Application plays a very peculiar rôle in HO languages, and treating it just as an ordinary operation leads to technical difficulties in relation to types. More precisely, the rewrite rules for `apply` obtained by means of Warren's translation are not well-typed in Milner's system [Mil78, DM82]. This is because their left-hand sides do not correspond to the principal type of the application operation, but rather to different particular instances. This was pointed out in [Han89, Han90], where a more general polymorphic type system was proposed for HO functional and logic languages. In contrast to Milner's, this type system required type information at run time, so that static type discipline was lost. Moreover, non-stric functions were not supported, because the intended semantics was equational rather than cpo-based.

Instead of performing a translation to the FO level, some languages have chosen to follow a "HO syntax with FO semantics" approach. Usually, one thinks of functions as mappings over a domain of elements, so that the *principle of extensionality* holds: two functions which return the same result for all possible arguments are equal. Alternatively, one can take an *intensional* view, thinking of functions simply as abstract objects (technically called *function intensions*) that can be applied to arguments by means of the application operation, and not requiring extensionality any more. It turns out that this idea can be made precise, so that undecidability problems can be avoided and nice soundness and completeness results can be proved. In the sequel, we will use the term *intensional semantics* for this kind of approach. An example of a HO programming language with intensional semantics is HiLog [CKW93], which supports a purely logic programming style based on the definition of predicates by means of clauses.

An intensional semantics for untyped HO applicative programs has been defined in [GMHGRA97]. Given a program $\mathcal{P}$ and a set of variables $\mathcal{X}$, one gets a least Herbrand model $\mathfrak{M}_{\mathcal{P}}(\mathcal{X})$ whose domain is the set $Pat_{\Sigma_\perp}(\mathcal{X})$ of all partial patterns over $\mathcal{X}$. In this model, patterns act as function intensions. Function symbols f with $ar(f) > 0$ are interpreted as intensions. Additionally, $\mathfrak{M}_{\mathcal{P}}(\mathcal{X})$ provides an application operation, whose behaviour depends on the rewrite rules of $\mathcal{P}$ in a natural way. Models over other poset domains are also defined following similar ideas. This intensional semantics behaves as a natural extension of the FO case, as described in Subsection 5.2.3 above. In fact, the intensional HO semantics can be reduced to the FO semantics by means of Warren's translation method, as shown in [GM94].

For the case of FO typed programs with algebraic datatypes, a natural extension of the semantics from Subsection 5.2.3 is also known [ASRA97b, ASRAar]. In this setting, models have two domains, one for data values and another one for *type intensions*, i.e., objects intended to represent types. A membership relation between data values and type intensions is also provided within each model. In particular, least Herbrand models $\mathfrak{M}_{\mathcal{P}}(\mathcal{X}, \mathcal{A})$ are guaranteed to exist, whose data domain is the quotient $Term_{\Sigma_\perp}(\mathcal{X})/\cong_{\mathcal{C}}$

of all partial data terms with variables in $\mathcal{X}$ modulo $\cong_{\mathcal{C}}$, and whose type domain is the set $Type_{\Sigma}(\mathcal{A})$ of all syntactic types with type variables in $\mathcal{A}$. Both [GMHGRA97] and [ASRA97b] rely on suitable extensions of the rewriting logic CRWL. A natural combination of the two extensions leads to a single extension of FO CRWL, which is presented below.

Definition 14. HO extension of CRWL, with algebraic datatypes.
BT Bottom: $e \to [\bot]$ for any partial expression e.
RR Restricted Reflexivity: $X \to [X]$ for any variable X.
DC Decomposition: $\dfrac{e_1 \to [t_1] \cdots e_m \to [t_m]}{h\ \overline{e}_m \to [h\ \overline{t}_m]}$ for any partial pattern $h\ \overline{t}_m$.
OR Outer Reduction: $\dfrac{e_1 \to [t_1\theta] \cdots e_n \to [t_n\theta] \qquad C\theta \qquad r\theta\ \overline{a}_m \to [t]}{f\ \overline{e}_n \overline{a}_m \to [t]}$

for $t \in Pat_{\Sigma_\bot}(\mathcal{X})$, $t \neq \bot$, $f\ \overline{t}_n \to r \Leftarrow C \ \in\ \mathcal{P}$, $\theta \in Subst_{\Sigma_\bot}(\mathcal{Y}, \mathcal{X})$.
JN Joinability: $\dfrac{l \to [t] \qquad r \to [t]}{l == r}$ for *total* $t \in Pat_{\Sigma}(\mathcal{X})$. □

The additional arguments $\overline{a}_m$ in rule (OR) are due to the fact that an n-ary function can sometimes be applied to more than n arguments, as we have seen in Subsection 5.3.1. In contrast to Definition 1, approximation statements have now the form $e \to [t]$, where $[t]$ is an abbreviated notation for the equivalence class $[t]_{\mathcal{C}}$ of a *partial pattern* t. Of course, $\mathcal{C}$ stands for the data axioms of the current signature. The equivalence class notation indicates that the new CRWL calculus is intended to work modulo the congruence $\cong_{\mathcal{C}}$. More precisely, at any point where we are trying to deduce a statement $e \to [t]$, we are implicitely allowed to replace t by any other pattern t' such that $t \sqsubseteq_{\mathcal{C}} t'$. This is more general than $t \cong_{\mathcal{C}} t'$, but also sound, because the intended semantics for "$\to$" is the inverse of the approximation preorder $\sqsubseteq_{\mathcal{C}}$. A concrete example of CRWL proof, involving multisets, is shown below.

Example 21. The magic bag.
Assume a datatype `coin` with data constructors `m1` and `m100`, among others. We define a magic bag including infinitely many coins of kinds `m1` and `m100`, as well as a predicate which checks wether a bag includes at least one occurrence of `m100`.

```
bag :: {[coin]}                rich             :: {[coin]} → bool

bag → {[m1,m100|bag]}          rich {[m100|Xs]} → true
```

The statement `rich bag → [true]` has the following CRWL proof:

1. `rich bag → [true]` by (OR), 2, 3.
2. `bag → [{[m100|⊥]}]` by (OR), 4.
3. `true → [true]` by (DC).
4. `{[m1,m100|bag]} → [{[m100|⊥]}]` by (DC), 5, 6.
5. `m1 → [⊥]` by (BT).
6. `{[m100|bag]} → [{[m100|⊥]}]` by (DC), 7, 8.

7. `m100` $\to$ `[m100]` by (DC).
8. `bag` $\to$ `[`$\bot$`]` by (BT).

Reasoning modulo $\cong_{\mathcal{C}}$ has taken place at step 4. in the previous proof. The statement `{[m1,m100|bag]}` $\to$ `[{[`$\bot$`, m100|`$\bot$`]}]` implicitely replaces the statement `{[m1,m100|bag]}` $\to$ `[{[m100|`$\bot$`]}]`, since

$$\{\!\![\mathtt{m100|}\bot]\!\!\} \sqsubseteq_{\mathcal{C}} \{\!\![\bot,\ \mathtt{m100|}\bot]\!\!\}$$

Goals for HO CRWL programs are similar to the FO case. Extensions of the lazy narrowing calculus CLNC dealing with algebaic data constructors and HO applicative expressions have been presented in [ASRA97a, ASRAar] and [GMHGRA97], respectively. These calculi are sound and complete w.r.t. the corresponding semantics.

The narrowing calculus in [ASRA97a, ASRAar] is intended for strongly well-typed FO programs. In comparison to untyped CLNC, the main novelty here is that delayed unifications and joinability statements have to be solved modulo $\cong_{\mathcal{C}}$. Unfortunately, this leads to a more indeterministic calculus, tending to compute too many redundant solutions. Improvements should be certainly possible, at least for some interesting special cases such as sets and multisets. Assuming a well-typed initial goal, resolution is guaranteed to preserve well-typepedness, and computed solutions are always well-typed.

On the other hand, the narrowing calculus HOCLNC from [GMHGRA97] has been designed for untyped HO applicative programs. In comparison to FO CLNC, all the goal transformation rules are adapted to work with patterns instead of data terms. Also, the outer narrowing transformation (ON) foresees the possibility of additional arguments $\overline{a}_m$, as in the HO CRWL rule (OR). Maybe the more significant difference is related to the treatment of *HO variables* F, occurring in expressions of the form $F\ \overline{e}_m$. HOCLNC includes special transformations to guess appropriate bindings for the HO variable F in such cases. For example, one of the outer narrowing rules in HOCLNC looks as follows:

GN Guess + Outer Narrowing
$\exists\overline{U}.S \,\square\, P \,\square\, F\ \overline{e}_n\overline{a}_k == e,\ E \Vdash$
$\exists\overline{X},\overline{U}.F \mapsto f\ \overline{t}_p,\ (S \,\square\, \dots, e_j \to s_j, \dots,\ P \,\square\, C,\ r\ \overline{a}_k == e,\ E)\sigma$,
where $f\ \overline{t}_p\overline{s}_q \to r \Leftarrow C \in_{ren} \mathcal{P}$ with fresh variables $\overline{X}$, $\sigma = \{F \mapsto f\ \overline{t}_p\}$.

A complete HOCLNC computation is shown below. We assume a program including data constructors for Peano numbers and lists, as well as the applicative rewrite rules from Example 17.

Example 22. Goal solving with HO variables.
The goal $\square$ $\square$ `F X == s (s z)` admits the solution {F $\mapsto$ twice s, X $\mapsto$ z}, which can be computed as follows:

$\square$ $\square$ $\underline{\mathtt{F\ X}}$ `== s (s z)` $\Vdash_{GN}$
$\exists$ `F', X'. F` $\mapsto$ `twice F'` $\square$ $\underline{\mathtt{X \to X'}}$ $\square$ `F' (F' X') == s (s z)` $\Vdash_{IB}$

$\exists$ `F'. F` $\mapsto$ `twice F'` $\Box\ \Box$ $\underline{\texttt{F' (F' X) == s (s z)}}$ $\Vdash_{BD}$
$\exists$ `F'. F'` $\mapsto$ `s, F` $\mapsto$ `twice s` $\Box\ \Box$ $\underline{\texttt{s X == s z}}$ $\Vdash_{DC}$
$\exists$ `F'. F'` $\mapsto$ `s, F` $\mapsto$ `twice s` $\Box\ \Box$ $\underline{\texttt{X == z}}$ $\Vdash_{BD}$
$\exists$ `F'. X` $\mapsto$ `z, F'` $\mapsto$ `s, F` $\mapsto$ `twice s` $\Box\ \Box$

The transformation (BD) used at the third step above is also related to HO variables. It combines generation of a new binding for the solved part with decomposition. The CLNC rule (SB) becomes a particular case of (BD), as seen in the last step. Note that the binding `F'` $\mapsto$ `s` refers to an intermediate variable, and is not needed for the computed answer. A longer HOCLNC derivation can compute the solution {F $\mapsto$ twice s, X $\mapsto$ z, Y $\mapsto$ s(s(s z))} for the more complex goal $\Box\ \Box$ `map F [X, s z] == [s(s z), Y]`.

Unfortunately, HOCLNC has some drawbacks from the viewpoint of type discipline. One important property of Milner's type system for functional languages is that a statically well-typed program is guaranteed to produce no type errors during execution, because rewriting does always preserve well-typed expressions. As functional programmers like to say, "well-typed programs don't go wrong". In our setting, however, well-typed goals for strongly well-typed programs can go wrong sometimes. There are two different reasons for this. Firstly, goals including HO variables can become ill-typed if the binding guessed for a HO variable `F` at some step has the wrong type. Coming back to Example 22, assume that the datatypes and function definitions from Example 20 are also included in the current program, which will be strongly well-typed. Consider again the well-typed goal $\Box\ \Box$ `map F [X, s z] == [s(s z), Y]`. Then HOCLNC computes the ill-typed solution {F $\mapsto$ execPlan [ ], X $\mapsto$ s(s z), Y $\to$ s z}. The point here is that the function `execPlan [ ]` behaves as the identity, although its type forbids to take a list of Peano numbers as argument. Type errors of this kind are related to the fact that HOCLNC works w.r.t. intensional semantics, which (being equivalent to a "HO to FO" translation in Warren style) can give rise to an ill-typed behaviour of the implicit `apply` operation. To avoid these problems, one has to give up static type discipline and perform some amount of dynamic type checking at run time, see [GMHGRA99]. For programs and goals without HO variables, dynamic type checking can be avoided.

A second source of type errors in HOCLNC is the decomposition transformation (DC) when applied to some HO goals. Consider the well-typed function `snd :: A` $\to$ `B` defined by the rewrite rule `snd X Y` $\to$ `Y`, and the well-typed goal $\Box\ \Box$ `snd true == snd z`. One (DC) step produces the ill-typed and unsolvable goal $\Box\ \Box$ `true == z`. In this case, the type error arises because `snd` is *not transparent*, in the sense that the type of its result does not reveal the type of its first argument. Also solvable goals can become ill-typed after one non-transparent (DC) step. For instance, this happens in the case of the well-typed goal $\Box\ \Box$ `snd (map s [ ]) == snd (map not [ ])`. This kind of type errors seem difficult to avoid, unless one restritcs oneself to programs which make no use of HO patterns. In any case, the notion of well-typed program given in Definition 11 must be further restricted by the requirement that all the patterns occurring in the left-hand sides of rewrite rules are transparent; see [GMHGRA99] for a precise formulation.

This closes our discussion of semantics and goal solving techniques for typed HO applicative programs. In addition to the CRWL oriented works [GMHGRA97, ASRA97b, ASRA97a, ASRAar, GMHGRA99] which have served as the basis of our presentation, we have mentioned other approaches to HO functional logic programming all along this section. At this point we will still mention two other related papers.

In [GMHGRA93] applicative rewrite systems were proposed as a basis for a functional logic language with a cpo-based semantics, for which a sound and complete narrowing calculus was defined. This paper also introduced HO patterns as a generalization of FO data terms. Types and non-deterministic functions were not considered. In fact, the narrowing calculus did not support delayed evaluation, neither sharing.

Working on the basis of equational semantics, the authors of [NMI95] presented a narrowing calculus NCA, which is sound and complete for orthogonal applicative TRSs w.r.t. to *right normal* goals, i.e., goals consisting of equations whose right-hand sides are ground normal forms. The completeness proof was achieved by a reduction to the previously known narrowing calculus OINC [IN97], which is complete for orthogonal TRSs w.r.t. right normal goals. Regarding application as an explicit binary operation, it turns out that any applicative orthogonal TRS is also an orthogonal TRS in the ordinary sense. For this reason, OINC itself is already complete for orthogonal applicative TRSs. The calculus NCA, however, is more efficient and natural for the applicative case. It is presented by means of transformation rules bearing some similarity to those of HOCLNC, but without the semantic distinction between approximation and joinability statements. NCA views strict equality as a function to be defined by rewrite rules, and some special transformation rules are proposed in order to solve strict equations more efficiently. Their behaviour, however, is not equivalent to the transformation rules for joinability statements in HOCLNC. In particular, strict equalities between HO patterns cannot be solved in NCA.

5.3.4 Higher Order Programming in $\mathcal{TOY}$

The $\mathcal{TOY}$ system supports the programming features described in the previous subsections, with the only exception of algebraic datatypes. Types in $\mathcal{TOY}$ correspond exactly to the abstract syntax given in Subsection 5.3.1 above. The concrete syntax is such that identifiers for type constructors must begin with lower-case letters, while type variables begin with upper-case letters. The predefined types `bool`, `int`, `real` and `char` count as nullary type constructors. A type constructor for lists is also predefined, with syntax `[A]`. The predefined type `string` is an alias for `[char]`. More generally, type alias can be introduced by means of declarations `type <alias> = <type expression>`, as in Haskell [Pe99]. Data constructors are defined in $\mathcal{TOY}$ by means of `data` declarations, also borrowed from Haskell. A `data` declaration introduces a type constructor along with its associated data constructors and their principal types. For instance, the datatype `action` used in Example 20 above, should be declared as follows in $\mathcal{TOY}$:

```
data action = pickup block | putdown block |
              stack block block | unstack block block
```

Similar declartions can be written for the other datatypes from Example 20. Rewrite rules in HO $\mathcal{TOY}$ programs follow the same conventions as in the FO case, except that left-hand sides can now use curried notation and patterns, as explained in Subsection 5.3.1. Type declarations for defined functions are optional. The $\mathcal{TOY}$ system infers principal types from the program, according to Milner's polymorphic type system [Mil78, DM82]. In case that an inferred type is not unifiable with the type declared by the user, a type error is reported and compilation is aborted. If no syntactic errors neither type errors are found, compilation proceeds by combining a translation from HO to FO described in [GM94] with the techniques already explained in Subsection 5.2.5, and executable SICStus Prolog code is generated. Some work is already done [ASLFRA98] towards a Prolog compilation scheme to deal with the multiset datatype {[A]}. This is not yet part of the working system. One of the difficulties regarding the implementation of multisets is the absence of a clear notion of demanded argument. For instance, matching the simple pattern {[z|Xs]} modulo the data axiom for multisets, demands an unpredictable amount of evaluation for the actual parameter expression. Of course, similar problems are to be expected for other algebraic datatypes.

In the rest of this subsection we will illustrate the use of HO programming techniques in $\mathcal{TOY}$. In addition to the adventages of the functional style, $\mathcal{TOY}$ can exploit logic programming features, especially non-deterministic search. Moreover, HO patterns go beyond typical functional languages in their ability to manipulate functions as data values. This feature has been already used in Example 18, where simple circuits built from `NOT`, `AND` and `OR` gates were represented as HO patterns, and the problem of finding a realization of the majority circuit was considered. This problem can be solved by means of the *lazy generate and test* method, already explained in Example 7. To this purpose, we need a *generator* and a *tester*. As generator, we will use the non-deterministic function `genCircuit`, such that `genCircuit N` returns all possible patterns representing circuits of size `N`. The size is understood as the number of gates.

```
infixr 20 //                          X // Y = X
(//)        :: A -> A -> A            X // Y = Y

type ngates = int

genCircuit  :: ngates -> circuit

genCircuit N = if N == 0
               then x1 // x2 // x3
               else notGate (genCircuit (N-1)) //
                    tryBinaryGate N
tryBinaryGate   :: ngates -> circuit
tryBinaryGate N = andGate (genCircuit K) (genCircuit L) //
                  orGate (genCircuit K) (genCircuit L)
                  <== N > 0, K == upto (N-1), L == N-1-K
upto   :: int -> int
upto N = if N == 0 then 0 else N // upto (N-1)
```

Note that we have used the *choice* function (//) as a convenient device for the definition of other non-deterministic functions. The type of circuit testers, and a particular tester for the behaviour of the majority circuit, are defined as follows:

```
type test = circuit -> bool
majorityTest   :: test
majorityTest C :- map C allInputs == map majority allInputs
allInputs :: [(bool,bool,bool)]
allInputs = [(false,false,false), (false,false,true),
             (false,true,false), (false,true,true),
             (true,false,false), (true,false,true),
             (true,true,false),  (true,true,true)]
majority             :: (bool,bool,bool) -> bool
majority(false,X2,X3) = X2 /\ X3
majority (true,X2,X3) = X2 \/ X3
```

Although `majority` has type `circuit`, it does not serve as a solution by itself, because its definition does not describe a combination of logical gates. As in Example 7, we still need an auxiliary function to connect the tester and the generator.

```
checkSatisfies     : : test -> circuit -> circuit
checkSatisfies T C  =  C <== T C
```

Now we can write the main function which looks for a circuit with some desired behaviour and size. We define also a non-deterministic nullary function which generates all non-negative integers.

```
anyInt :: int
anyInt = 0 // 1+anyInt
findCircuit     : : test -> ngates -> circuit
findCircuit T N  =  checkSatisfies T (genCircuit N)
```

Finally, to find a realization of the majoity circuit, we propose the goal:

```
L == anyInt, N == L+3, findCircuit majorityTest N == C
```

Due to the left-to-right execution strategy, this causes a search by levels L, looking for circuits with increasing sizes N $\geq$ 3. The first solution given by the $\mathcal{TOY}$ system is what we have mentioned in Example 18:

```
L == 1, N == 4,
C == (andGate (orGate (andGate x1 x3) x2) (orGate x1 x3))
```

Here and in Example 7 we have seen two different application of the lazy generate and test technique. In fact, all the applications of this method follow the same scheme, and it is possible to program a single generic solution, by means of a HO function `findSol`. This function expects three parameters: a generator `G`, a tester `T` and an input `I`, and it returns a solution `S`. The generator is applied to the input in order to produce candidate solutions, and any candidate which passes the tester can be returned as solution. As we already know, lazy evaluation will allow that partially computed candidates are rejected by the tester. The $\mathcal{TOY}$ definitions realizing this algorithm are as follows.

```
findSol :: (Input -> Sol) -> (Sol -> bool) -> Input -> Sol

findSol G T I = check T (G I)

check     :: (Sol -> bool) -> Sol -> Sol
check T S = S <== T S
```

An alternative definition of the functions `findCircuit` and `permSort` as particular instances of the generic function `findSol` is now possible:

```
findCircuit     : : test -> ngates -> circuit
findCircuit T N  =  findSol genCircuit T N
```

Or, more simply:

```
findCircuit : : test -> ngates -> circuit
findCircuit  =  findSol genCircuit
```

As a simple exercise, the reader can write an alternative definition for the function `permSort` from Example 7, using `findSol`. To conclude this subsection, we will cite two papers including more interesting examples of $\mathcal{TOY}$ programming techniques. In [CRLF99a] the authors present so-called *extensions*, a functional-logic programming technique which serves as a convenient alternative to *monads* [Pe99] in several situations. In [CRLF99b] the reader will find a systematic methodology for programming *parsers* as non-deterministic functions, taking adventage of logic variables and HO patterns.

5.4 Constraint Programming

The Constraint Logic Programming (CLP) scheme was introduced by Jaffar and Lassez [JL87] as a formal framework, based on constraints, for an extended class of logic programs. In a similar spirit, some extensions based on constraints can be developed for the CRWL approach to functional logic programming, as we will see in this final section.

5.4.1 Programming with Constrained Rewrite Systems

The key insight of CLP languages is to extend logic programming by allowing constraints with a predefined interpretation. CLP is intended as a general scheme that supplies a language CLP($\mathcal{C}$) for each particular choice of a *constraint system* $\mathcal{C}$. A constraint system consists of four components: a constraint language $\mathcal{L}_{\mathcal{C}}$, a constraint structure $\mathcal{D}_{\mathcal{C}}$, a constraint theory $\mathcal{T}_{\mathcal{C}}$ and a constraint solver $solv_{\mathcal{C}}$, all of them belonging to $\mathcal{C}$'s signature $\Sigma_{\mathcal{C}}$. For most practical purposes, it is reasonable to assume that $\mathcal{L}_{\mathcal{C}}$ is a fragment of the language of first-order logic with signature $\Sigma_{\mathcal{C}}$, including the atomic formulas *true* and *false* and all equations $t \approx s$ between terms of signature $\Sigma_{\mathcal{C}}$, and closed under variable renaming, existential quantification and conjunction. Formulas belonging to $\mathcal{L}_{\mathcal{C}}$ are called *constraints.* $\mathcal{D}_{\mathcal{C}}$ must be some $\Sigma_{\mathcal{C}}$-structure, which represents the computation domain and provides the intended interpretation of constraints. The theory $\mathcal{T}_{\mathcal{C}}$ must be designed to axiomatize some relevant properties of $\mathcal{D}_{\mathcal{C}}$. Finally, the solver $solv_{\mathcal{C}}$ maps each constraint $\varphi \in \mathcal{L}_{\mathcal{C}}$ to one of the three results *true*, *false* or *unknown*, indicating if φ is satisfiable in $\mathcal{D}_{\mathcal{C}}$, unsatisfiable in $\mathcal{D}_{\mathcal{C}}$, or it cannot tell. Two significant examples of constraint system are: $\mathcal{R}$, corresponding to the computation domain of real numbers, which supports arithmetic constraints; and $\mathcal{H}$, corresponding to the Herbrand domain that underlies ordinary logic programming. See [JM94, JMMS96] for more details.

CLP($\mathcal{C}$) programs are sets of definite Horn clauses $a \Leftarrow b_1, \ldots , b_n$, where a must be an atom $p(t_1, \ldots , t_n)$ corresponding to some user-defined predicate p, and each b_i can be either an atom of this form, or a *constraint.* Horn logic, as in Subsection 5.2.1, together with the constraint theory $\mathcal{T}_{\mathcal{C}}$ provides a logical semantics for CLP($\mathcal{C}$) programs. For each given CLP($\mathcal{C}$) program $\mathcal{P}$ there is a least interpretation of the user-defined predicates which, taken as a conservative extension of the constraint structure $\mathcal{D}_{\mathcal{C}}$ gives a model of $\mathcal{P}$. Goals for CLP($\mathcal{C}$) programs have the same form as bodies of clauses. They are solved by a combination of SLD-resolution [Apt90, Llo87] and constraint solving, called *constrained SLD-resolution.* Unification between the head of a clause and the atom selected from the current goal is replaced by equality constraints which are placed in the new goal, and the solver $solv_{\mathcal{C}}$ can be applied at each step, causing failure as soon as the conjunction of all the constraints in the current goal becomes unsatisfiable. More precisely, consider a goal $p(\overline{t}_n),\ G,\ \varphi$ and a program clause $p(\overline{s}_n) \Leftarrow H,\ \psi$, where we assume that φ, ψ are conjunctions of constraints, while G, H are conjunctions of user-defined atoms. Then, constrained SLD resolution causes the following goal transformation:

$p(\overline{t}_n),\ G,\ \varphi \Vdash G,\ H,\ t_1 \approx s_1, \ldots ,\ t_n \approx s_n,\ \varphi,\ \psi$

provided that $solv_{\mathcal{C}}(t_1 \approx s_1 \wedge \ldots \wedge t_n \approx s_n \wedge \varphi \wedge \psi) \neq false$. Goals which contain only a satisfiable conjunction of constraints are shown as solutions to the user, usually after simplifying them to some $\mathcal{C}$-equivalent *solved form.* Constrained SLD-resolution is sound and complete w.r.t. the logical semantics of CLP programs. Detailed definitions and proofs can be found in [JMMS96]. The next example shows a simple CLP($\mathcal{R}$) program to compute Fibonacci numbers:

Example 23. Fibonacci numbers in CLP($\mathcal{R}$).

```
nthFib(0,1).
nthFib(1,1).
nthFib(N,X)  ⇐ N ≥ 2, nthFib(N-1,A), nthFib(N-2,B), X ≈ A+B
```

The constraint solver for CLP($\mathcal{R}$) [JMSY92] uses the simplex algorithm and Gauss-Jordan elimination to handle linear constraints, and delays non-linear constraints until they become linear. Due to these constraint solving abilities, the Fibonacci program admits multiple modes of use:

Goal: `nthFib(10,X)` **Goal:** `nthFib(N,89)`
Solution: `X ≈ 89` **Solution:** `N ≈ 10`

By analogy with the CLP idea, we can also think of Constraint Functional Logic Programming (CFLP), with programs consisting of *constrained* applicative rewrite rules $f\ \overline{t}_n \Leftarrow C$ similar to those from Subsection 5.3.1, extended with the possibility to include constraints in the condition C. More precisely, C will be now understood as a finite conjunction whose members are either joinability statements between expressions, or constraints φ. CFLP can be also understood as a general scheme which supplies one particular language CFLP($\mathcal{C}$) for each given constraint system $\mathcal{C}$. The next example shows a small CFLP($\mathcal{R}$) program which also computes Fibonacci numbers.

Example 24. Fibonacci numbers in CFLP($\mathcal{R}$).

```
fib N       → (fibSeq 1 1) !! N
fibSeq A B  → [A|fibSeq B (A+B)]
[X|Xs] !! N → if N == 0 then X else Xs !! (N-1)
```

In this program, `fib N` is computed as the `N`th member of the *infinite sequence* of all Fibonacci numbers. Multiple modes of use are also possible:

Goal: `fib 10 == X` **Goal:** `fib N == 89`
Solution: `X == 89` **Solution:** `N == 10`

Moreover, this program has linear time and space complexity, while the program in Example 23 has exponential complexity. Of course, a more efficient CLP($\mathcal{R}$) program can be also written, but in any case Example 24 illustrates an interesting combination of lazy evaluation and constraint solving.

A CFLP scheme was developed in [LF92, LF94], preceding the first proposal of the CRWL framework in [GMHGRA96]. CFLP programs were built as sets of constrained rewrite rules, as we have explained. The setting was FO and untyped, and non-deterministic functions were disallowed. Similarly to the CLP case, CFLP($\mathcal{C}$) programs have a least model which is a conservative extension of the constraint structure $\mathcal{D}_{\mathcal{C}}$, and *constrained lazy narrowing* can be used as a sound and complete goal solving mechanism. The presence of defined functions introduces several significant complications w.r.t. the CLP case. Firstly, constraints in goals and conditions of programs rules do not always belong to the primitive constraint language $\mathcal{L}_{\mathcal{C}}$, but can have occurrences of defined function symbols. As a consequence, the constraint solver and the lazy narrowing mechanism become mutually dependent, and lazy computation steps cannot be detected effectively in general. Secondly, delayed unifications in CFLP (needed for parameter passing in any lazy functional logic language) cannot be treated as equality constraints. In fact, constraint structures in CFLP must be adapted to a cpo-based semantics, where constraints must behave as monotonic and continuous predicates. Equations $t \approx s$ with the semantics of equational logic must be replaced by joinability statements, whose meaning does not correspond to delayed unifications. Approximation statements $e \to t$ can express delayed unification, as we know. However, they cannot be seen as constraints, because they lack monotonicity. In [LF92, LF94] this problem was solved by imposing left-hand sides of the form $f(\overline{X}_n)$ with n *different* variables X_i, and using special constraints in the conditions to emulate the effect of lazy unification with data patterns.

CFLP($\mathcal{H}$) was proposed in [LF94] as a particular instance of CFLP, where the constraint system $\mathcal{H}$ corresponded to a cpo-version of the Herbrand domain, equivalent to the ideal completion of a Herbrand poset $Term_{\Sigma_\perp}(\mathcal{X})$, but including also several primitive constraints. These were *strict equality* `==`, *disequality* `/=`, and special *unification constraints*, used only to emulate delayed unifications. Thanks to the unification constraints, CFLP($\mathcal{H}$) allows data terms as syntactic sugar in the left-hand sides of rewrite rules. Strict equality has the same meaning as joinability in CRWL, restricted to the deterministic case. The semantics of disequality can be explained in CRWL terms as follows: for given expressions a and b (maybe including defined function symbols), a `/=` b holds if there are partial data terms s and t such that $a \to s$, $b \to t$ and s, t have conflicting data constructors at some common position. Computed solutions in CFLP($\mathcal{H}$) are constraints in solved form, including strict equalities X `==` t which represent a substitution, and also disequalities of the form X `/=` t. Disequality constraints are quite useful for several problems; we will consider some concrete examples in the next subsection.

More recently, some work has been done to combine CFLP ideas with the typed CRWL approach presented in Subsection 5.3.2. The paper [ASLFRA99] proposes a CFLP language, called SETA, whose constraint system is based on a combination of primitive and user-defined domains. SETA's primitive domain is given by the constraint structure $\mathcal{R}$, as in CLP($\mathcal{R}$). In addition, the user can define his own domains as datatypes, using a polymorphic type constructor for multisets and arbitrary *free* type constructors. Currently, SETA is defined as a FO language, and no functional types are considered, but an extension to the HO case will probably pose no significant difficulties. The primitive type `real` counts in SETA's type system as a nullary type constructor. The constraint language in SETA includes the primitive *arithmetic constraints* given by $\mathcal{R}$, as well as the *symbolic constraints* `==`, `/=`, $\in$ and $\notin$. The constraints `==` and `/=` can be used over any type, and are similar to their analoga in CFLP($\mathcal{H}$), except that they work modulo the data axioms for multisets. Constraints of the forms $e \in es$ and $e \notin es$

make sense for any expressions $e :: \tau$ and $es :: \{\![\,\tau\,]\!\}$. The rewriting calculus and the lazy narrowing calculus from [ASRA97b, ASRA97a] have been extended in [ASLFRA99] to obtain a semantics and a goal solving procedure for SETA. Delayed unifications $e \to t$ occur during goal solving. They are solved modulo the data axiom for multisets, and not regarded as constraints. See [ASLFRA99, AS98] for more details.

One of the motivations for the design of SETA was the work by Marriott and others on so-called *constraint multiset grammars* as a formalism for the specification of visual languages [HM91, Mar94, MMW98]. Visual languages, in contrast to textual languages, display information in the form of pictures, figures and diagrams, where certain spacial relationships between components are relevant, but no particular sequential ordering of components is intended. Therefore, in the field of visual language theory, multisets of components turn out to be a convenient representation of pictures, and constraints can be used to express relationships between components. Arithmetic constraints over the real numbers are particularly useful in many cases. As a simple example, inspired by [HM91], we will consider the recognition of quadrilaterals by means of a SETA program. For convenience, we will write rewrite rules of the form $l \to true \Leftarrow C$ as clauses, as explained in Subsection 5.2.5. We assume two type constructors `pt` and `qd` for building points of the plane and quadrilaterals, respectively:

```
pt :: (real,real) → point

qd :: (real,real,real,real) → figure
```

Our aim is that `pt(X,Y)` represents a point with coordinates `(X,Y)`, while `qd(P1,P2,P3,P4)` represents a quadrilateral (a particular case of figure) with vertices `Pi`. A well-built quadrilateral can be specified by the condition that the four midpoints of its sides form a parallelogram. Using this characterization, we solve the problem of building quadrilaterals from a given multiset of points, in such a way that the resulting figures have no common vertices.

Example 25. Recognizing quadrilaterals.

```
quadrilateral :: (point,point,point,point) → bool
quadrilatral(P1,P2,P3,P4) :- parallelogram(midPoint(P1,P2)
                                           midPoint(P2,P3),
                                           midPoint(P3,P4),
                                           midPoint(P4,P1))
parallelogram :: (point,point,point,point) → bool
parallelogram(M1,M2,M3,M4) :- M1-M4 == M2-M3

build :: {[point]} → {[quadrilateral]}
build({[ ]})              → {[ ]}
build({[P1,P2,P3,P4|R]}) → {[qd(P1,P2,P3,P4)|build(R)]}
                          ⇐ P1 ∉ R, P2 ∉ R, P3 ∉ R, P4 ∉ R
```

```
ms :: {[point]}
ms → {[pt(-3,0), pt(12,3),pt(4,3),pt(11,-6),
      pt(3,0),pt(14,-2), pt(8,0),pt(5,-4)]}
```

The easy definitions of the functions `midPoint` and `(-)` have been omitted. With this program, SETA could solve the goal `build(ms) == M` giving as solution

```
M == {[qd(pt(-3,0),pt(4,3),pt(3,0),pt(5,-4)),
      qd(pt(8,0),pt(12,3),pt(14,-2),pt(11,-6))]}
```

Formally, SETA is not an instance of the CFLP scheme from [LF92], where types were not considered. As interesting future work, we plan to investigate a revised version of CFLP, borrowing ideas from typed CRWL. The revised general scheme would have SETA as one of its instances. In addition to constraints over predefined types (corresponding to the constraint structures of CLP and CFLP), the new scheme should consider constraints over user-defined datatypes as well.

5.4.2 Constraint Programming in $\mathcal{TOY}$

A first proposal to implement SETA's multiset constraints by means of a Prolog translation was given in [ASLFRA98], but this is not yet part of the working $\mathcal{TOY}$ system. On the contrary, disequality constraints over arbitrary free datatypes and arithmetic constraints over the real numbers are already available in the $\mathcal{TOY}$ system. Implementing disequality constraints requires to maintain a constraint store with constraints in solved form X `/=` t, which must be awaken whenever X becomes bound. A first implementation technique for a subset of CFLP($\mathcal{H}$), consisting of so-called *uniform programs*, was presented in [KLFMNRA92]. Later on, an implementation of full CFLP($\mathcal{H}$), based on a translation into Prolog with demand driven strategy, was poposed in [ASGLLF94]. The current $\mathcal{TOY}$ system relies on this technique to solve disequality constraints, and it invokes the CLP($\mathcal{R}$) solver of Sictus Prolog [Int99] to solve real arithmetic constraints. Of course, user-defined functions occurring within arithmetic constraints have to be evaluated before invoking the solver. A formalization of the interplay between the arithmetic solver and lazy narrowing has been given in [ASLFRA99, AS98]. The $\mathcal{TOY}$ system has two modes of use: with or without the arithmetic solver. The user can switch between these two modes by means of special commands. Activating the arithmetic solver causes some overhead in those computations that actually do not need it.

We will now show some examples of constraint programming in $\mathcal{TOY}$. First we illustrate a very simple application of disequality constraints to define a function with the behaviour of strict equality, but with the ability to return two boolean results.

```
streq :: A -> A -> bool
streq X Y = true  ⇐ X == Y
streq X Y = false ⇐ X /= Y
```

An optimized version of this definition is available in $\mathcal{TOY}$ as a primitive. Moreover, the notation `l == r` is understood by $\mathcal{TOY}$ either as a joinability constraint or a call to the function `streq`, according to the context. This facility is used in the next $\mathcal{TOY}$ definitions.

```
del          :: A -> [A] -> [A]
del Y [X|Xs] = if Y == X then Xs else [X|del Y Xs]

infixl 50 --&
(--&)         :: [A] -> [A] -> [A]
Xs --& [ ]    = Xs
Xs --& [Y|Ys] = (del' Y Xs) --& Ys
```

Note that `(del Y Xs)` returns the result of deleting the first occurrence of the element `Y` (if there are any) from the list `Xs`, which is expected to include at least one occurrence of `Y`. If this is not the case, `(del Y Xs)` is undefined. The call `Xs --& Ys` computes a new list by deleting from `Xs` the first occurrence of each element from `Ys`. Goals using these functions can give rise to disequality constraints as part of the solution. For instance, $\mathcal{TOY}$ computes two solutions for the goal `del Y [X1, X2] == Xs`, namely:

`Y == X1, Xs == [X2];`   and   `Y /= X1, Y == X2, Xs == [X1]`

We will now reconsider Example 20, dealing with a planning problem in a blocks world. This cannot be directly executed in $\mathcal{TOY}$ because multisets are not yet supported by the current implementation. Unification modulo multisets is needed for the left-hand sides of the rewrite rules defining the function `execAction`. In this particular case, each rewrite rule describes an action which requires certain facts to be present (as indicated by the left-hand side) and prescribes how to replace them by other facts (as indicated by the right-hand side). This is simple enough to be programmable in $\mathcal{TOY}$, taking lists as a representation of multisets, and using the list difference function `(--&)`. More precisely, we can define:

```
type mset A = [A]  type situation = mset fact
type plan = [action]

ini, fin :: situation
 ini = [clear b2,clear b3,over b3 b1,table b2,table b1,empty]
 fin = [clear b1,over b1 b2,over b2 b3,table b3,empty]

execPlan          :: plan -> situation -> situation
execPlan [ ]    S = S
execPlan [A|As] S = execPlan As (execAction A S)
```

```
execAction                  :: action -> situation -> situation
execAction (pickup B)    S = [hand B] ++
                             (S --& [table B,clear B,empty])
execAction (putdown B)   S = [table B,clear B,empty] ++
                             (S --& [hand B]])
execAction (stack B C)   S = [over B C,clear B,empty] ++
                             (S --& [hand B,clear C])
execAction (unstack B C) S = [hand B,clear C] ++
                             (S --& [over B C,clear B,empty])

infixl 40 ==&
(==&)     :: situation -> situation -> bool
Xs ==& Ys = (Xs --& Ys == [ ]) /\ (Ys --& Xs == [ ])
```

Note that we need a special function `(==&)` to test equality of situations, because `==` would not work as desired for multisets. Now, it would seem that solving the goal `execPlan Plan ini ==& fin` will give us the desired plan. Unfortunately, $\mathcal{TOY}$'s search strategy (inherited from Prolog) does not behave well in this case. The problem can be mended by defining an auxiliary predicate:

```
isList        :: [A] -> bool
isList [ ]    :- true
isList [X|Xs] :- isList Xs
```

Now, solving the goal `isList Plan, execPlan Plan ini ==& fin` will generate lists of increasing sizes, initially consisting of different logic variables, and then further constrained to become a working plan for our problem. The first solution computed by $\mathcal{TOY}$ is the same we had announced in Example 20:

```
[unstack b3 b1, putdown b3, pickup b2, stack b2 b3, pickup b1,
                           stack b1 b2]
```

Up to this point we have illustrated the use of disequality constraints. The language CLP($\mathcal{R}$) [JMSY92] has shown the usefulness of arithmetic constraints for several applications, that can be easily written as $\mathcal{TOY}$ programs. As a simple example, we show a slight variant of a well-known CLP($\mathcal{R}$) solution to the problem of computing a multiple period loan (mortgage) that lasts for `T` time periods. We follow the presentation in [MS98], but defining a function instead of a predicate.

```
type principal, time, interest, repayment, balance = real

mortgage :: (principal,time,interest,repayment) -> balance
mortgage(P,T,I,R) = P ⇐ T == 0
mortgage(P,T,I,R) = mortgage(P+P*I-R,T-1,I,R) ⇐ T >= 1
```

The possibility of diverse modes of use is very clearly illustrated by this function. We can compute the balance `B` corresponding to borrowing 1000 Euros for 10 years at an interest rate of 10%, repaying 150 Euros per year:

Goal: `mortgage(1000,10,10/100,150) == B`

Solution: `B == 203.129`

We can also ask how much can be borrowed in a 10 year loan at 10%, with annual repayments of 150 Euros:

Goal: `mortgage(P,10,10/100,150) == 0`

Solution: `P == 921.685`

More complex queries are also possible, as e.g. asking for the relationship between the initial principal, the repayment and the balance in a 10 year loan at 10%. The solution is a linear constraint relating the values of three variables:

Goal: `mortgage(P,10,10/100,R) == B`

Solution: `B == 2.594*P-15.937*R`

Of course, lazy evaluation and other features of $\mathcal{TOY}$ can be combined with arithmetic constraints whenever convenient. A simple illustration is given by Example 24, that can be executed in $\mathcal{TOY}$. A more involved example, where arithmetic constraints are combined with several features of functional logic programming, deals with the computation of regions in the plane. It can be found in $\mathcal{TOY}$'s reference manual [LFSH99].

5.5 Conclusions

In this paper we have looked to extensions of *logic programming* with more powerful features coming from the *functional and constraint programming* fields. One of our main motivations was to achieve a *logical characterization* of program semantics in such a combined setting. To this purpose, we have presented the special rewriting logic CRWL, developed by the Declarative Programming Group at Universidad Complutense de Madrid, where the notion of lazy and possibly non-deterministic function plays a central rôle. In contrast to more traditional frameworks such as *equational logic* and *Horn logic*, CRWL has the ability to characterize the intended computational behaviour in a correct way. We have presented goal solving procedures based on *lazy narrowing* which are sound and complete w.r.t. CRWL semantics. In contrast to classical results on the completeness of narrowing strategies, neither termination nor confluence hypothesis are required. However, *sharing* is needed for soundness, in order to respect the semantics of non-deterministic functions. A formalization of sharing is built-in in our presentation of lazy narrowing, and it is very simple to deal with.

The main part of this paper has been devoted to a quite detailed description of the basic CRWL approach, which deals with the combination of logic programming and lazy functional programming in an untyped, first-order setting. Various natural extensions of basic CRWL allow to incorporate a polymorphic type system, higher-order programming features, and constraints. We have discussed such extensions in a less detailed way. Moreover, we have presented the $\mathcal{TOY}$ language and system, which implements most of the

CRWL features described in the paper, and we have shown a number of simple but illustrative programs written in $\mathcal{TOY}$.

In our view, the CRWL approach gives an attractive and mathematically well-founded basis for multiparadigm declarative programming. Nevertheless, the $\mathcal{TOY}$ system is an experimental tool with may limitations. Currently, our group is working in some extensions of the system. They will include primitives for input-output in monadic style, as well as a programming environment with a graphical user interface, offering tools to support program edition, compilation, execution and debugging. In particular, the debugging tools will rely on the CRWL semantics of programs, following the *declarative debugging* approach [Nai73]. Extensions of CRWL to accomodate a generic scheme for constraint programming are another interesting topic for future research.

Acknowledgements

The author is thankful to an anonymous referee for his/her constructive criticisms. He is also very much indebted to Puri Arenas, Rafa Caballero, Juan Carlos González, Paco López, Teresa Hortalá, Jaime Sánchez, Mercedes Abengózar and other members of the Declarative Programming Group at Universidad Complutense de Madrid, for several years of cooperation in different works related to the CRWL approach to multiparadigm declarative programming. Rafa, Paco and Jaime have helped a lot to select and test programming examples, and Mercedes has done fine work in implementing the current version of the $\mathcal{TOY}$ system. Moreover, Rafa has read a preliminary version of the paper very carefully, pointing out typos and mistakes. Teresa merits special thanks for her help in preparing the final version of the paper.

References

[AEM94] S. Antoy, R. Echahed, and M.Hanus. A needed narrowing strategy. In *Proc. ACM Symp. on Principles of Programming Languages (POPL'94)*, ACM Press, pages 268–279, 1994.

[AH00] S. Antoy and M. Hanus. Compiling multiparadigm declarative programming into Prolog. In *Proc. Int. Workshop on Frontiers of Combining Systems (FROCOS'2000)*, volume 1794 of *Springer LNCS*, pages 171–185, 2000.

[AKLN87] H. Aït-Kaci, P. Lincoln, and R. Nasr. Le Fun: Logic, equations and functions. In *Proc. IEEE Int. Symp. on Logic Programming*, IEEE Comp. Soc. Press, pages 17–23, 1987.

[ALS94] J. Avenhaus and C. Loría-Sáez. Higer order conditional rewriting and narrowing. In *Proc. 1st Int. Conference on Constraints in Computational Logics (CCL'94)*, volume 845 of *Springer LNCS*, pages 269–284, 1994.

[Ant91] S. Antoy. Lazy evaluation in logic. In *Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'91)*, volume 528 of *Springer LNCS*, pages 371–382, 1991.

[Ant92] S. Antoy. Definitional trees. In *Proc. Int. Conf. on Algebraic and Logic Programming (ALP'92)*, volume 632 of *Springer LNCS*, pages 143–157, 1992.

[Ant97] S. Antoy. Optimal non-deterministic functional logic computations. In *Proc. Int. Conf. on Algebraic and Logic Programming (ALP'97)*, volume 1298 of *Springer LNCS*, pages 16–30, 1997.

[Apt90] K.R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 10, pages 493–574. Elsevier and The MIT Press, 1990.

[AS98] P. Arenas-Sánchez. *Programación Declarativa con Restricciones sobre Tipos de Datos Algebraicos (in spanish).* PhD thesis, Universidad Complutense de Madrid, 1998.

[ASGLLF94] P. Arenas-Sánchez, A. Gil-Luezas, and F.J. López-Fraguas. Combining lazy narrowing with disequality constraints. In *Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'94)*, volume 844 of *Springer LNCS*, pages 385–399, 1994.

[ASLFRA98] P. Arenas-Sánchez, F.J. López-Fraguas, and M. Rodríguez-Artalejo. Embedding multiset constraints into a lazy functional logic language. In *Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'98), held jointly with the 6th Int. Conf. on Algebraic and Logic Programming (ALP'98)*, volume 1490 of *Springer LNCS*, pages 429–444, 1998.

[ASLFRA99] P. Arenas-Sánchez, F.J. López-Fraguas, and M. Rodríguez-Artalejo. Functional plus logic programming with built-in and symbolic constraints. In *Proc. Int. Conf. on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Springer LNCS*, pages 152–169, 1999.

[ASRA97a] P. Arenas-Sánchez and M. Rodríguez-Artalejo. A lazy narrowing calculus for functional logic programming with algebraic polymorphic types. In *Proc. Int. Symp. on Logic Programming (ILPS'97)*, The MIT Press, pages 53–68, 1997.

[ASRA97b] P. Arenas-Sánchez and M. Rodríguez-Artalejo. A semantic framework for functional logic programming with algebraic polymorphic types. In *Proc. Int. Joint Conference on Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *Springer LNCS*, pages 453–464, 1997.

[ASRAar] P. Arenas-Sánchez and M. Rodríguez-Artalejo. A general framework for lazy functional logic programming with algebraic polymorphic types. *Theory and Practice of Logic Programming*, To appear.

[BG86] P.G. Bosco and E. Giovannetti. IDEAL: an ideal deductive applicative language. In *Proc. IEEE Int. Symp. on Logic Programming*, IEEE Comp. Soc. Press, pages 89–95, 1986.

[BGLM94] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19&20:3–23, 1994.

[BGM88] P.G. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theoretical Computer Science*, 59:3–23, 1988.

[BKK+96] P. Borovansky, C. Kirchner, H. Kirchner, P. Moreau, and M. Vittek. Elan, a logical framework based on computational systems. In *Proc. of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 35–50. Elsevier Science, 1996. Electronic publication.

[BM90] J.P. Banâtre and D. Le Métayer. The Gamma model and its discipline of programming. *Science of Computer Programming*, 15:55–77, 1990.

[BM93] J.P. Banâtre and D. Le Métayer. Programming by multiset transformation. *Communications of the ACM*, 36:98–111, 1993.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[BvEG+87] H.P. Barendregt, M.C.J.D. van Eeckelen, J.R.W. Glauert, J.R. Kennaway, M.J. Plasmeijer, and M.R.Sleep. Term graph rewriting. In *Proc. PARLE'87*, volume 259 of *Springer LNCS*, pages 141–158, 1987.

[CDE+99] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: Specification and programming in rewriting logic. Technical report, Computer Science Laboratory, SRI International, 1999.

[CF93] P.H. Cheong and L. Fribourg. Implementation of narrowing. the Prolog-based approach. In de Bakker Apt and Rutten, editors, *Logic Programming Languages: Constraints, Functions and Objects*, pages 1–20. The MIT Press, 1993.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CKW93] W. Chen, M. Kifer, and D.S. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15:187–230, 1993.

[CRLF99a] R. Caballero-Roldán and F.J. López-Fraguas. Extensions: A technique for structuring functional logic programs. In *Proc. Third Int. Conf. on Perspectives of System Informatics*, volume 1755 of *Springer LNCS*, pages 297–310, 1999.

[CRLF99b] R. Caballero-Roldán and F.J. López-Fraguas. A functional-logic perspective of parsing. In *Proc. 4th Fuji Int. Symp. on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *Springer LNCS*, pages 85–99, 1999.

[De86] D. DeGroot and G. Lindstrom (eds.). *Logic Programming: Functions, Relations and Equations.* Prentice-Hall, Englewood Cliffs, 1986.

[DFI+96] R. Diaconescu, K. Futatsugi, M. Ishisone, A.T. Nagakawa, and T. Sawada. An overview of CafeOBJ. In *Proc. of the First International Workshop on Rewriting Logic and its Applications*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 75–88. Elsevier Science, 1996. Electronic publication.

[DJ90] N. Dershowitz and J.P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier and The MIT Press, 1990.

[DM79] N. Dershowitz and Z. Manna. Proving termination with multiset orderings. *Communications of the ACM*, 22(8):465–476, 1979.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Proc. ACM Symp. on Principles of Programming Languages (POPL'82)*, ACM Press, pages 207–212, 1982.

[DO90] N. Dershowitz and M. Okada. A rationale for conditional equational programming. *Theoretical Computer Science*, 75:111–138, 1990.

[Ech90] R. Echahed. On completeness of narrowing strategies. *Theoretical Computer Science*, 72:133–146, 1990.

[Ech92] R. Echahed. Uniform narrowing strategies. In *Proc. 3rd Int. Conf on Algebraic and Logic Programming (ALP'92)*, volume 632 of *Springer LNCS*, pages 259–275, 1992.

[Fay79] M.J. Fay. First-order unification in an equational theory. In *Proc. Workshop on Automated Deduction (CADE'79)*, Academic Press, pages 161–177, 1979.

[FLMP93] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Information and Computation*, 102(1):86–113, 1993.

[Fri85] L. Fribourg. SLOG: A logic programming interpreter based on clausal superposition and rewriting. In *Proc. IEEE Int. Symp. on Logic Programming*, IEEE Comp. Soc. Press, pages 172–184, 1985.

[GLMP91] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A logic plus functional language. *Journal of Computer and System Science*, 42(2):139–185, 1991.

[GM86] E. Giovannetti and C. Moiso. A completeness result for e-unification algorithms based on conditional narrowing. In *Proc. of the Workshop on Foundations of Logic and Functional Programming, Trento*, volume 306 of *Springer LNCS*, pages 157–167, 1986.

[GM87] J.A. Goguen and J. Meseguer. Models and equality for logical programming. In *Proc. TAPSOFT'87*, volume 250 of *Springer LNCS*, pages 1–22, 1987.

[GM94] J.C. González-Moreno. *Programación Lógica de Orden Superior con Combinadores (in spanish)*. PhD thesis, Universidad Complutense de Madrid, 1994.

[GMHGLFRA99] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.

[GMHGRA92] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. Denotational versus declarative semantics for functional programming. In *Proc. Int. Conf. on Computer Science Logic (CSL'91)*, volume 626 of *Springer LNCS*, pages 134–148, 1992.

[GMHGRA93] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. On the completeness of narrowing as the operational semantics of functional logic programming. In

Proc. Int. Conf. on Computer Science Logic (CSL'92), volume 702 of *Springer LNCS*, pages 216–230, 1993.

[GMHGRA96] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symp. on Programming (ESOP'96)*, volume 1058 of *Springer LNCS*, pages 156–172, 1996.

[GMHGRA97] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. A higher order rewriting logic for functional logic programming. In *Proc. Int. Conf. on Logic Programming (ICLP'97)*, The MIT Press, pages 153–167, 1997.

[GMHGRA99] J.C. González-Moreno, M.T. Hortalá-González, and M. Rodríguez-Artalejo. Semantics and types in functional logic programming. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *Springer LNCS*, pages 1–20, 1999.

[GS90] C.A. Gunter and D. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 633–674. Elsevier and The MIT Press, 1990.

[GTWJ77] J.A. Goguen, J.W. Thatcher, E.G. Wagner, and J.B.Wright. Initial algebra semantics and continuous algebras. *Journal of the ACM*, 24(1):68–95, 1977.

[H^+00] M. Hanus et al. Curry: an integrated functional logic language, version 0.7. Technical report, Universität Kiel, February 2000. Available at `http://www.informatik.uni-kiel.de/ curry/`.

[Han89] M. Hanus. Polymorphic higher-order programming in Prolog. In *Proc. Int. Conf. on Logic Programming (ICLP'89)*, The MIT Press, pages 382–397, 1989.

[Han90] M. Hanus. A functional and logic language with polymorphic types. In *Proc. Int. Symp. on Design and implementation of Symbolic Computation Systems*, volume 429 of *Springer LNCS*, pages 215–224, 1990.

[Han94a] M. Hanus. Combining lzy narrowing and simplification. In *Proc. Int. Symp. on Programming Language Impplementation and Logic Programming (PLILP'94)*, volume 884 of *Springer LNCS*, pages 370–384, 1994.

[Han94b] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 9&20:583–628, 1994.

[Han95] M. Hanus. Efficient translation of lazy functional logic programs into Prolog. In *Proc. Int. Workshop on Logic Programming Synthesis and Transformation (LOPSTR'95)*, volume 1048 of *Springer LNCS*, pages 252–266, 1995.

[Han97] M. Hanus. A unified computation model for functional and logic programming. In *Proc. ACM Symp. on Principles of Programming Languages (POPL'97)*, ACM Press, pages 80–93, 1997.

[HL79] G. Huet and J.J. Lévy. Call by need computations in non-ambiguous linear term rewriting systems. Technical Report 359, IRIA, 1979.

[HL91] G. Huet and J.J. Lévy. Computations in orthogonal term rewriting systems i, ii. In J.L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honour of J. Alan Robinson*, pages 395–414 and 415–443. The MIT Press, 1991.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. Logic Programming Series. The MIT Press, 1994.

[HM91] R. Helm and K. Marriott. Declarative specification and semantics for visual languages. *Journal of Visual Languages and Computing*, 2:311–331, 1991.

[HM97] M. Hamada and A. Middeldorp. Strong completeness of a lazy conditional narrowing calculus. In *Proc 2nd Fuji Int. Workshop on Functional and Logic Programming*, World Scientific, pages 14–32, 1997.

[Höl89] S. Hölldobler. *Foundations of Equational Logic Programming*. Lecture Notes in Computer Science. Springer Verlag, 1989.

[HP96] A. Habel and D. Plump. Term graph narrowing. *Mathematical Structures in Computer Science*, 6(6):649–676, 1996.

[HP99a] A. Habel and D. Plump. Complete strategies for term graph narrowing. In *Recent Trends in Algebraic Development Techniques*, volume 1589 of *Springer LNCS*, pages 152–167, 1999.

[HP99b] M. Hanus and C. Prehofer. Higher-order narrowing with definitional trees. *Journal of Functional Programming*, 9(1):33–75, 1999. Preliminary version appeared in Proc. Int. Conf. on Rewriting Techniques and Applications (RTA'96), Springer LNCS 1103, pp. 138–152, 1996.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-Calculus*. Number 1 in London Mathematical Society Student Texts. Cambridge University Press, 1986. (reprinted 1988,1990).

[HS90] S. Hölldobler and J. Schneeberger. A new deductive approach to planning. *New Generation Computing*, 8:225–244, 1990.

[Hul80] J.M. Hullot. Canonical forms and unification. In *Proc. Conf. on Automated Deduction (CADE'80)*, volume 87 of *Springer LNCS*, pages 318–334, 1980.

[Hus92] H. Hussmann. Nondeterministic algebraic specifications and nonconfluent term rewriting. *Journal of Logic Programming*, 12:237–255, 1992.

[Hus93] H. Hussmann. *Non-determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[IN97] T. Ida and K. Nakahara. Leftmost outside-in narrowing calculi. *Journal of Functional Programming*, 7(2):129–161, 1997.

[Int99] Intelligent Systems Laboratory, Swedish Institute of Computer Science. *SICStus Prolog User's Manual, Release 3.8*, October 1999.

[JL87] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proc. ACM Symp. on Principles of Programming Languages (POPL'97)*, ACM Press, pages 111–119, 1987.

[JLM84] J. Jaffar, J.L. Lassez, and M.J. Maher. A theory of complete logic programs with equality. *Journal of Logic Programming*, 1:211–223, 1984.

[JM94] J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19&20:503–581, 1994.

[JMMS96] J. Jaffar, M.J. Maher, K. Marriott, and P.J. Stuckey. The semantics of constraint logic programs. Technical Report 96/39, University of Melbourne, 1996.

[JMSY92] J. Jaffar, S. Michaylov, P.J. Stuckey, and R.H.C. Yap. The CLP(R) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[Kap87] S. Kaplan. Symplifying conditional term rewriting systems: Unification, termination and confluence. *Journal of Symbolic Computation*, 4(3):295–334, 1987.

[KLFMNRA92] H. Kuchen, F.J. López-Fraguas, J.J. Moreno-Navarro, and M. Rodríguez-Artalejo. Implementing a lazy functional logic language with disequality constraints. In *Proc. Joint Int. Conf. and Symposium on Logic Programming (JICSLP'92)*, The MIT Press, pages 207–221, 1992.

[Klo92] J.W. Klop. Term rewriting systems. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 2–116. Oxford University Press, 1992.

[Lan75] D.S. Lankford. Canonical inference. Technical Report ATP-32, Department of Mathematics and Computer Science, University of Texas at Austin, 1975.

[LF92] F.J. López-Fraguas. A general scheme for constraint functional logic programming. In *Proc. Int. Conf. on Algebraic and Logic Programming (ALP'92)*, volume 632 of *Springer LNCS*, pages 213–227, 1992.

[LF94] F.J. López-Fraguas. *Programación Funcional y Lógica con Restricciones (in spanish).* PhD thesis, Universidad Complutense de Madrid, 1994.

[LFSH99] F.J. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. 10th Int. Conf. on Rewriting Techniques and Applications (RTA'99)*, volume 1631 of *Springer LNCS*, pages 244–247, 1999. Available at `http://titan.sip.ucm.es/toy`.

[LLFRA93] R. Loogen, F.J. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'93)*, volume 714 of *Springer LNCS*, pages 184–200, 1993.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, 1987. 2nd. ed.

[Llo95] J.W. Lloyd. Declarative programming in escher. Technical Report CSTR-95-013, Univ. of Bristol, Dept. of Comp. Sci., 1995.

[Llo99] J.W. Lloyd. Programming in an integrated functional and logic language. *Electronic Journal of Functional and Logic Programming*, 1999.

[Mar94] K. Marriott. Constraint multiset grammars. In *Proc. IEEE Symp. on Visual Languages*, IEEE Comp. Soc. Press, pages 118–125, 1994.

[MBP97] J.M. Molina-Bravo and E. Pimentel. Modularity in functional-logic programming. In *Proc. Int. Conf. on Logic Programming (ICLP'97)*, The MIT Press, pages 183–197, 1997. Extended version to appear in the Journal of Logic Programming.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[MH94] A. Middeldorp and E. Hamoen. Completeness results for basic narrowing. *Applicable Algebra in Engineering, Communications and Computing*, 5:213–253, 1994.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, 17:348–375, 1978.

[Mil91] D. Miller. A logic programming language with lambda-abstraction, function variables and simple unification. *Journal of Logic and Computation*, 1:497–536, 1991.

[MIS99] M. Marin, T. Ida, and T. Suzuki. On reducing the search space of higher-order lazy narrowing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, volume 1722 of *Springer LNCS*, pages 319–334, 1999.

[MM99] N. Martí and J. Meseguer. Action and change in rewriting logic. In R. Pareschi and B. Fronhöfer, editors, *Dynamic Worlds: From the Frame Problem to Knowledge Management*, volume 12 of *Applied Logic Series*, pages 1–53. Kluwer Academic Publishers, 1999.

[MMW98] K. Marriott, B. Meyer, and K.B. Wittenburg. A survey of visual language specification and recognition. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, chapter 2, pages 5–85. Springer Verlag, 1998.

[MN86] D. Miller and G. Nadathur. Higher-order logic programming. In *Proc. Int. Conf. on Logic Programming (ICLP'86)*, volume 225 of *Springer LNCS*, pages 448–462, 1986.

[MNRA92] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language babel. *Journal of Logic Programming*, 12:191–223, 1992.

[MO95] A. Middeldorp and S. Okui. A deterministic lazy narrowing calculus. *Journal of Symbolic Computation*, 25(6):104–118, 1995.

[MOI96] A. Middeldorp, S. Okui, and T. Ida. Lazy narrowing: Strong completeness and eager variable elimination. *Theoretical Computer Science*, 167:95–130, 1996. preliminary version appeared in Proc. CAAP'95, Springer LNCS 915, pp. 394–408, 1995.

[Möl85] B. Möller. On the algebraic specification of infinite objects - ordered and continuous models of algebraic types. *Acta Informatica*, 22:537–578, 1985.

[Mos90] P. Mosses. Denotational semantics. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 11, pages 575–631. Elsevier and The MIT Press, 1990.

[MS98] K. Marriott and P.J. Stuckey. *Programming with Constraints, an Introduction.* The MIT Press, 1998.

[Nai87] L. Naish. *Negation and Control in Prolog.* Lecture Notes in Computer Science. Springer Verlag, 1987.

[Nai73] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3). 27 pages.

[Nar86] S. Narain. A technique for doing lazy evaluation in logic. *Journal of Logic Programming*, 3:259–276, 1986.

[NM88] G. Nadathur and D. Miller. An overview of λ-Prolog. In *Proc. Int. Conf. on Logic Programming (ICLP'88)*, The MIT Press, pages 810–827, 1988.

[NMI95] K. Nakahara, A. Middeldorp, and T. Ida. A complete narrowing calculus for higher order functional logic programming. In *Proc. Int. Symp. on Programming Language Implementation and Logic Programming (PLILP'95)*, volume 982 of *Springer LNCS*, pages 97–114, 1995.

[NRS89] W. Nutt, P. Réty, and G. Smolka. Basic narrowing revisited. *Journal of Symbolic Computation*, 7:295–317, 1989.

[Pad87] P. Padawitz. Strategy-controlled reduction and narrowing. In *Proc. Int. Conf. on Rewriting Techniques and Applications (RTA'87)*, volume 256 of *Springer LNCS*, pages 242–255, 1987.

[Pe99] J. Peterson and K. Hammond (eds.). Report on the programming language Haskell 98, a non-strict, purely functional language. Technical report, February 1999.

[Pre94] C. Prehofer. Higher-order narrowing. In *Proc. IEEE Symp. on Logic in Computer Science (LICS'94)*, IEEE Comp. Soc. Press, pages 507–516, 1994.

[Pre95] C. Prehofer. A call-by-need strategy for higher-order functional logic programming. In *Proc. Int. Logic Programming Symp. (ILPS'95)*, The MIT Press, pages 147–161, 1995.

[Pre98] C. Prehofer. *Solving Higher Order Equations: From Logic to Programming.* Birkhäuser Verlag, 1998.

[Red85] U. Reddy. Narrowing as the operational semantics of functional languages. In *Proc. Int. Symp. on Logic Programming*, IEEE Comp. Soc. Press, pages 138–151, 1985.

[RS82] J.A. Robinson and E.E. Sibert. LOGLISP: Motivation, design and implementation. In K.L. Clark and S.A. Tärnlund, editors, *Logic Programming*, pages 299–313. Academic Press, 1982.

[SE92] A. Sarmiento-Escalona. *Una aproximación a la Programación Lógica con Funciones Indeterministas (in spanish).* PhD thesis, Universidad A Coruña, 1992.

[SG89] W. Snyder and J. Gallier. Higher-order unification revisited: Complete sets of transformations. *Journal of Symbolic Computation*, 8:101–140, 1989.

[Sla74] J.R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *Journal of the ACM*, 21(4):622–642, 1974.

[Smo95] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000 in Lecture Notes in Computer Science, pages 324–343. Springer Verlag, 1995.

[SNI97] T. Suzuki, K. Nakagawa, and T. Ida. Higher-order lazy narrowing calculus: A computation model for a higher-order functional logic language. In *Proc. Int. Conf. on Algebraic and Logic Programming (ALP'97)*, volume 1298 of *Springer LNCS*, pages 99–113, 1997.

[Sny91] W. Snyder. *A Proof Theory for General Unification.* Birkhäuser Verlag, 1991.

[Wad85] P. Wadler. How to replace failure by a list of successes. In *Proc. IFIP Int. Conf. on Funct. Prog. Lang. and Computer Architectures*, volume 201 of *Springer LNCS*, pages 113–128, 1985.

[War82] D.H.D. Warren. Higher-order extensions to Prolog: are they needed? In D. Michie J.E. Hayes and Y.H. Yao, editors, *Machine Intelligence*, volume 10, pages 441–454. Edinburg Univ. Press, 1982.

[You89] J.H. You. Enumerating outer narrowing derivations for constructor-based term rewriting systems. *Journal of Symbolic Computation*, 7:319–341, 1989.

6 Building Industrial Applications with Constraint Programming

Helmut Simonis[⋆]

COSYTEC SA, Orsay, France

Current Address: Parc Technologies Ltd, London, United Kingdom

6.1 Introduction

In this chapter[1] we will give an overview of real-life applications developed with constraint logic programming. Constraint logic programming (CLP) combines declarative logic based programming with specialised constraint solving methods from artificial intelligence, Operations Research (OR) and mathematics. It allows the clear and concise expression of a wide class of combinatorial problems together with their efficient solution. In parallel with ongoing research in this field, CLP is now increasingly used to tackle real world decision making problems. In a first part of the chapter, we will briefly present the methods and tools used for CLP and describe typical application areas. We introduce the concepts of global constraints, meta-heuristics and constraint visualisation, which are central to large scale constraint solving.

A second part of the presentation will give a classification of problem domains suitable for the constraint programming approach and discusses various large scale applications which have been developed using the CHIP CLP system. These applications were co-developed by COSYTEC with specialists in the respective application domain.

- The ATLAS application is used at the Monsanto plant in Antwerp for the scheduling of a herbicide production and bottling plant. It is a typical example of a semi-process industry scheduling problem with constraints due to limited storage capacity at all levels.
- The FORWARD-C system is used in 5 oil refineries world-wide for short- and medium term planning of the production sequence. A graphical user interface is used to describe the refinery and generate the constraint model. The constraint program is coupled with a non-linear simulation tool written in FORTRAN.
- The TACT system is an integrated transport planning and scheduling system developed for a large food-industry company in the UK. The system plans the delivery of raw materials to different processing factories with a fleet of lorries and a variety of other resources.

⋆ Supported by the ESPRIT working group CCL-II, ref. WG # 22457.

1 This paper is an extended version of reference [Sim95a].

The last part of the presentation will concentrate on lessons learned from this application development, showing interesting new research fields and practical problems in developing state-of-the-art decision support systems.

6.2 Constraint Programming

In this section we give some background on constraint programming, the type of problems that are handled and the techniques that are used.

6.2.1 Characteristics of Problems

We now briefly introduce constraint satisfaction problems and discuss the motivation for the use of constraint logic programming (CLP). We explain which type of problem is best suited for the CLP approach and where these problems occur in practice. They all share a set of characteristics, which make them very hard to tackle with conventional problem solving methods. A more general introduction to constraint logic programming can be found in [JM93] [FHK92] [MS98].

Combinatorial problems occur in many different application domains. We encounter them in Operations Research (for example scheduling and assignment), in hardware design (verification and testing, placement and layout), financial decision making (option trading or portfolio management) or even biology (DNA sequencing). In all these problems we have to choose among many possible alternatives to find solutions respecting a large number of constraints.

We may be asked to find an admissible, feasible solution to a problem or to find a good or even optimal solution according to some evaluation criteria. From a computational point of view, we know that most of these problems are difficult, since they belong to the class of NP-hard problems. This means that no efficient and general algorithms are known to solve them.

At the same time, the problems themselves are rapidly evolving. For example, in a factory, new machines with unforeseen characteristics may be added, which can completely change the form of the production scheduling problem. New products with new requirements will be introduced on a regular basis. If a scheduling system can not be adapted to these changes, it will rapidly become outdated and useless.

Another aspect is that the complexity of the problems is steadily increasing. This may be due to increased size or complexity of the problem, or may be caused by higher standards on quality or cost effectiveness. Many problems which have previously been handled manually now exceed the capacity of a human problem solver.

At the same time, humans are typically very good at solving these complex decision making problems. They have a good understanding of the underlying problem and often know effective heuristics and strategies to solve them. In many situation they also know which constraints can be relaxed or ignored when a complete solution is not easily found. For this reason, it is necessary to build systems which co-operate with the user via friendly graphical interfaces and which can incorporate different rules and strategies as part of the problem solving mechanism.

Developing such applications with conventional tools has a number of important drawbacks:

- The development time will be quite long. This will increase the cost, making bespoke application development very expensive.
- The programs are hard to maintain. Due to their size and complexity, a change in one place may well require other changes in different parts of the program.
- The programs are difficult to adapt and to extend. As new requirements arise during the life cycle of the program, changes become more and more difficult.

The use of constraint logic programming should lead to a number of improvements:

- The development time is decreased, as the constraint solving methods are reused.
- A declarative problem statement makes it easier to change and modify programs.
- As constraints can be added incrementally to a program, new functionality can be added without changing the overall architecture of the system.
- Strategies and heuristics can be easily added or modified. This allows to include problem specific information inside the problem solver.

6.2.2 Main Concepts

Constraint programming is based on the idea that many interesting and difficult problems can be expressed declaratively in terms of *variables* and *constraints*. The variables range over a (finite) set of values and typically denote alternative decisions to be taken. The constraints are expressed as relations over subsets over variables and restrict feasible value combinations for the variables. Constraints can be given explicitly, by listing all possible tuples or implicitly, by describing a relation in some (say mathematical) form. A *solution* is an assignment of variables to values which satisfies all constraints.

Constraint propagation is the mechanism which controls the interaction of the constraints. Each constraint can deduce necessary conditions on the variable domains of its variables. The methods used for this constraint reasoning depend on the constraints, in the finite domain case they range from general, rather syntactic inference rules to complex combinations of algorithms used in global constraints. Whenever a constraint updates a variable, the constraint propagation will wake all relevant constraints to detect further consequences. This process is repeated until a fixpoint is reached, i.e. no further inference is possible.

For many interesting constraint systems, the constraint reasoning and constraint propagation is incomplete, i.e. it can not detect inconsistency at all times. Propagation must be combined with *search*, which is used to assign values to variables. After each assignment step constraint propagation restricts the possible values for the remaining variables, removing inconsistent values or detecting failure. If a failure is detected, the search returns to a previous decision and chooses an alternative. Search in constraint programming is usually under user control, heuristics and strategies can be easily programmed as search routines.

Constraint programming can be expressed over many different domains like linear terms over rational numbers, Boolean algebra, finite/infinite sets or intervals over floating point numbers. Very interesting development is possible for most of these domains or even more general, domain independent constraint solvers. But in this paper we only discuss finite domain constraint programming, currently the most developed constraint domain. At the moment, probably more than 95% of all industrial constraint applications use finite domains. Most examples in this paper are using the CHIP [DVS88] [VH89] system and its finite domain solver.

6.2.3 A Combination of Ideas

We will now review some of the background techniques which are used inside CLP. Figure 6.1 shows the basic influences on constraint logic programming. In the next paragraphs, we will discuss these different influences.

CLP can be seen as a natural extension of logic programming. Problems are expressed declaratively in Horn clause form, and the built-in search mechanism provides a simple way to generate solutions. Most of the theoretical results about correctness and completeness can be extended to CLP [JL87] [JM93] [MS98].

Constraint propagation and consistency checking are techniques originating in artificial intelligence. These methods provide the underlying framework for the constraint solvers in CLP tools.

Many of the actual methods to solve particular constraints come from the areas of finite mathematics and of Operations Research. Often classical algorithms can be changed to work within the constraint framework. They usually must be made *incremental*, i.e. avoiding repeated work when more information becomes available during the search and *backtrackable*, i.e. remembering the state of the propagation so that we can return to this state later during our search.

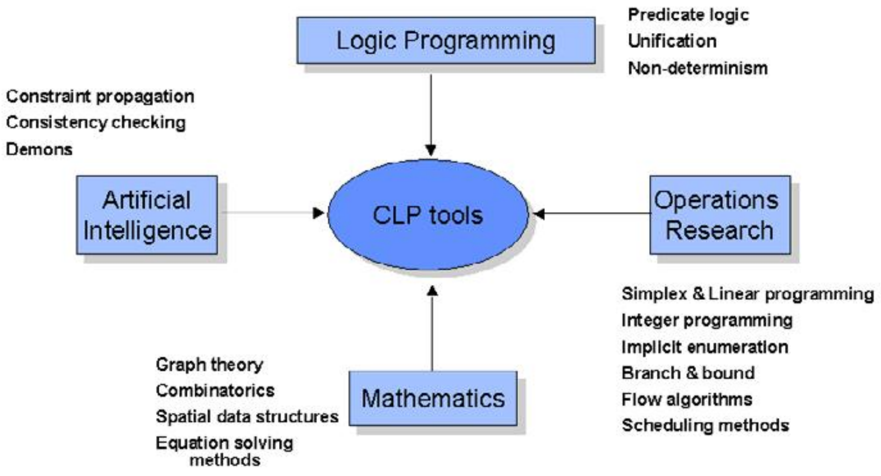


Fig. 6.1. Techniques behind constraints

6.2.4 The Emergence of Global Constraints

Much work on finite domain constraints in recent years was centred on the introduction of *global constraints* [AB93] [BC94] and *partial search techniques* [BBC97] [HG95]. We will briefly describe these topics before giving some examples of their use in a number of constraint applications.

The first versions of finite domain constraint systems were using rather simple propagation techniques. Consider the example shown in figure 6.2. Four variables are constrained by pair-wise disequality constraints. The constraint propagation for disequality is using *forward-checking* [VH89], which will solve the constraint as soon as one variable is assigned. With the given domains, no propagation is possible. Starting assignment with variable X, the values 1, 2, 3 are tested and lead to a failure, before the consistent value 4 is found for the variable X. We can avoid this useless search by applying better reasoning. The four variables must be pair-wise different, they must be alldifferent. Considering this one constraint, we can reformulate the problem as a *bi-partite matching problem* between variables and values and apply classical graph algorithms to solve it. We can deduce that the value 4 must be used by the variable X and the value 1 by the variable Z. The values 2 and 3 can be used for the variables Y and U.

This reformulation is the key to the understanding of global constraints. Instead of expressing a problem by many primitive, binary constraints we describe it with a few global constraint which work on sets of variables. Using this higher abstraction has two main advantages:

- Problems can be expressed more easily and with fewer constraints.

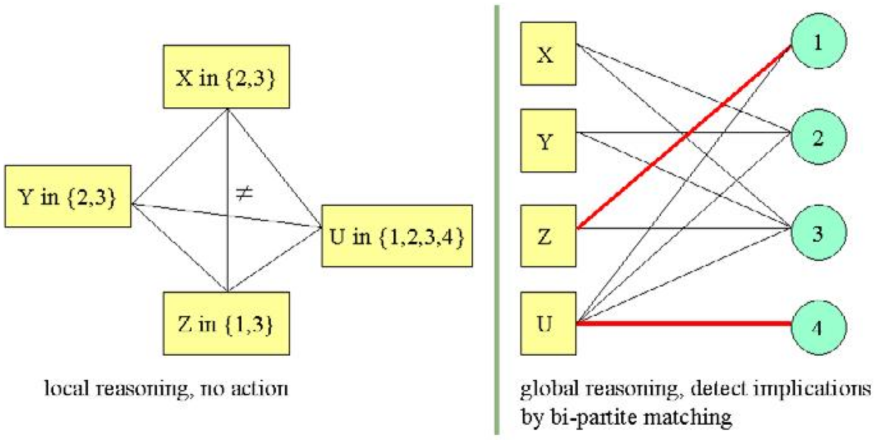

X in {2,3}
Y in {2,3}
≠
U in {1,2,3,4}
Z in {1,3}
local reasoning, no action
X
Y
Z
U
1
2
3
4
global reasoning, detect implications by bi-partite matching

Fig. 6.2. The need for global reasoning

- The abstraction allows us to use more complex algorithms inside the constraints.

Obviously, it is not advisable to create a new global constraint for each application that we solve. They should satisfy the following criteria:

- The constraint can be used for many different applications.
- The abstraction is as general as possible.
- It can be used together with other constraints.
- Efficient methods for constraint reasoning are known or can be developed for this constraint.
- The algorithmic complexity of the methods is acceptable.

In CHIP, the development of global constraints is based on the constraint morphology shown in figure 6.3. There are five basic concepts (shown at the bottom) for constraints. The *different* concept states that items are different from each other, the *order* concept states that items are (partially) ordered. The *resource* concept expresses that items use limited resources, The *tour* concept states that items in different locations must be visited in some sequence and the *dependency* concept states that some item depends on some other items.

In the first versions of finite domain solvers, primitive constraints for each of these concepts were introduced (shown in the second line from the bottom). A binary *inequality* constraint could be used to express a partial order between variables, for example. The propagation methods for all these constraints are quite simple and can be described with only a few syntactic inference rules [VH89].

A number of constraints can be introduced to solve particular problems with global reasoning (shown in the center), but which are not sufficiently

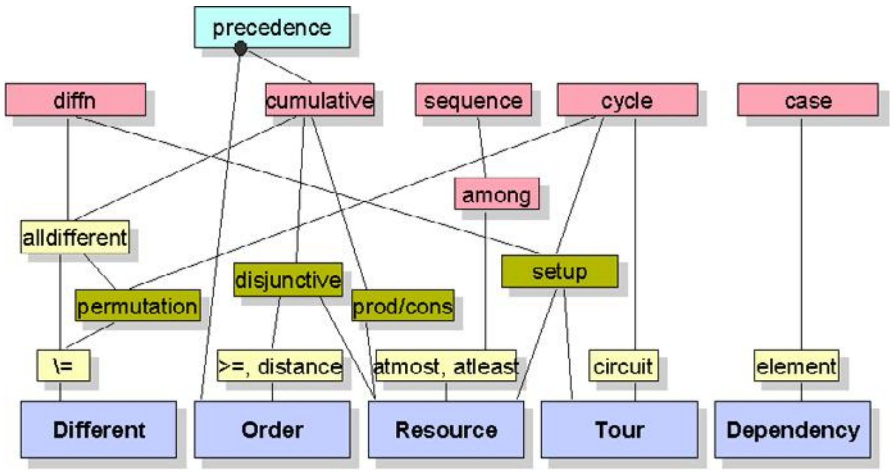


Fig. 6.3. Global Constraint Morphology

general to be considered as global constraints. A *disjunctive* constraint for example can be used to express resource constraints where each resource has capacity one and each task uses one resource.

But we can generalise the constraint to the case where the resource capacity can be arbitrary as well as the resource requirements for each task. This leads us to the *cumulative* [AB93] constraint, which is one of the CHIP global constraints (shown at the top). Connections between constraints show that the one below can be expressed by the one above.

The *precedence* [BBR96] constraint (at the top) is a further abstraction. It combines several cumulative constraints with a precedence graph between tasks. Considering these constraints together allows us to deduce even more information.

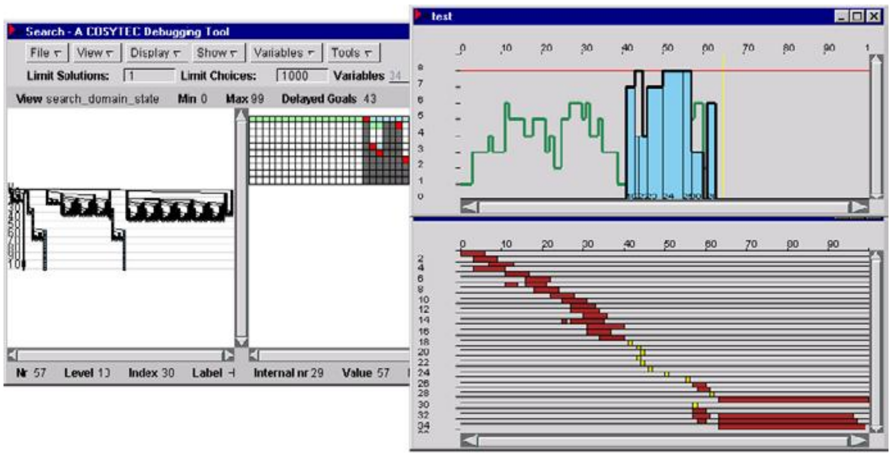
A number of other constraint systems include variants of global constraints. Variants of the cumulative constraint can be now be found for example in Sicstus Prolog [COC97], Eclipse [WNS97], IF Prolog and Oz [Smo95]. They differ not only in the parameters and side constraints that can be expressed, but also in methods and the mix of these methods inside the constraint reasoning.

6.2.5 Understanding Search

An important advantage of constraint programming over for example integer programming is the ease with which the programmer can control the search process. Most constraint systems have pre-defined search primitives, which can be easily extended for a particular program. The use of logic programming with its embedded backtracking search provides a particularly simple framework. Two complementary developments have significantly improved search methods in the last years.

One is the development of partial search techniques as meta-heuristics. Chronological depth-first search is often limited in exploring only a small part of the search tree. Decisions which are made early in the tree are not reconsidered since too many alternatives for variables below must be explored. Partial search methods allow to overcome this problem by limiting the number of alternatives explored in different nodes. Schemes like *limited discrepancy search* (LDS) [HG95] or *credit-based search* [BBC97] can consider many different paths in the search tree, without excessive implementation overhead. These meta-heuristics can be combined with the usual heuristics and strategies based on variable and value selection, and so provide problem independent tools to improve search.

The visualisation of the search tree with tools like the Oz Explorer [Sch97] or the CHIP search tree tool allow a much deeper understanding of the problem solving process. Developed within the DiSCiPl Esprit project (22532), the CHIP search tree tool [SA00] (shown in figure 6.4) adds functionality to analyse propagation and the state of the domain variables in each node. This can be combined with the visualisation of global constraints [SAB00], which show the reasoning inside the constraint for each propagation step.



Limit Solutions: 1 Limit Choices: 1000 Variables 34

View search_domain_state Min 0 Max 99 Delayed Goals 43

Nr 57 Level 13 Index 30 Label -I Internal nr 29 Value 57

Fig. 6.4. Search tree tool and constraint visualisation

6.3 The CHIP System

While constraints are the centrepiece of the CHIP application environment, a number of other components play an important role in developing applications. We now give an overview of this environment.

6.3.1 System Components

The CHIP system consists of a number of different components which together greatly simplify the development of applications. The tool extends the functionality of a host language with constraints and other sub systems. Two host languages are supported at the moment.

- The Prolog based version of CHIP uses intrinsic language features like unification and backtracking search to achieve a deep integration of the constraints with the host language. Constraints and search procedures can be easily defined and extended in the base language.
- The C/C++ version of the CHIP system takes the form of a constraint library which can be used inside application programs. Since search and variable binding are not part of the host language, the integration is not as seamless as in the Prolog version.

Other modules of the CHIP environment include XGIP, a graphical library based on Xwindows, QUIC, an SQL interface to relational databases and foreign language interfaces CLIC and EMC. The different modules can be used together with an object modelling system based on the object layer CHIP++ to build large scale end-user systems [KS95].

6.3.2 History

CHIP [DVS88] was one of the first CLP systems together with PROLOG III [Col90] and CLP(R) [JL87]. It was developed at the European Computer-Industry Research Centre (ECRC) in Munich during the period 1985-1990 [DVS88] [DVS87] [DSV90]. Different constraint solvers were developed and integrated in a Prolog environment. At the same time the technology was tested on a number of example applications. The CHIP system is now being marketed and developed by COSYTEC. The early versions of CHIP were also the basis for a number of derivatives by ECRC shareholder companies. BULL developed CHARME, a constraint based system with a C like syntax, ICL produced DecisionPower based on the ECRC CHIP/SEPIA system and Siemens is using constraints in SNI-Prolog (later IF-Prolog). ECRC has also continued research on constraints, which led to the Eclipse system, now maintained by IC-PARC. COSYTEC's current CHIP version V5.2 differs from the earlier constraint system by the inclusion of powerful global constraints [AB93] [BC94], partial search methods [BBC97] and visualisation tools [SA00] [SAB00].

6.3.3 Behind the Scene

The CHIP system consists of around 220000 lines of C code. Table 6.1 gives an overview of the size of the different modules. The Prolog kernel accounts for less than 10%, the different interfaces and the graphical system use 20%.

The constraint kernel uses another 10% of the code. The remaining 60% of the system are used in the code of the global constraints, which combine multiple (up to 40) algorithms and constraint methods for each primitive.

Table 6.1. CHIP system lines of code

Module	Size (lines of code)
CHIP kernel	175000
Prolog	15000
Core Constraints	20000
Global Constraints	140000
Interfaces (CLIC,EMC,QUIC)	5000
XGIP graphics and libraries	37000

This distribution shows the importance of both the graphical interface XGIP and in particular the global constraint handling methods inside CHIP.

6.4 Application Studies

From the beginning, the development of CHIP was linked to application requirements. A large number of application studies showed the practical interest of constraint technology. Many of these studies were benchmark comparisons with either conventional, special purpose programs or Operations Research (OR) tools. Three initial areas of interest were covered, combinatorial problems from OR [DVS87] [DVS88] [VC88] [DSV90] [VH89] [DSV92] [DSV88] [DS91], circuit design applications [SND88] [SD93] [GVP89] and financial decision support [Ber90].

An overview of early application studies can be found in [DVS88a] [VH89]. Other application studies with CHIP (outside COSYTEC) are for example [BDP94] [BLP95]. Results on benchmark problems can also be found in [VSD92] [AB93] [BC94] [BBR96]. A general classification of problems suitable for the constraint approach is given in [Sim96], while different types of constraint applications are listed in [Wal96].

6.5 Industrial Applications

In this section we present some large scale, industrial systems developed with CHIP. These examples show that constraint logic programming based on Prolog can be used efficiently to develop end-user applications. The constraint solver is a crucial part of these applications. Other key aspects are graphical user interfaces and interfaces to data bases or other programs. Some background on how such applications can be developed with logic programming is given in [KS95].

We have grouped these applications in different categories (following the classification from [Sim96]) according to the types of constraints which are included.

6.5.1 Assignment Problems

Assignment problems were the first type of industrial application that were solved with the emerging constraint tools. A typical example is the stand allocation for airports, were aircraft must be parked on the available stands during their stay at the airport. Different from scheduling problems, the (provisional) arrival and departure times are fixed in this type of problem. Additional constraints further restrict which resources can be used for which activity.

HIT (ICL). Probably the first industrial CLP application was developed for the HIT container harbour in Hong Kong [Per91]. The program was written in an early version of ECRC's CHIP, later adapted to DecisionPower from ICL. The objective was to allocate berths to container ship in the harbour, in such a way that resources and stacking space for the containers are available.

APACHE (Air France). The APACHE system [DS91] was a demonstrator for stand allocation at Roissy airport in Paris. Here the objective was to replan the allocation whenever a change of arrival/departure times occurred. A detailed model of this program is described in [Sim97]. An operational version of such an application is running in Korea.

Refinery Berth Allocation (ISAB). Another instance of this application type is refinery berth allocation. Arriving crude oil tankers and barges for the transport of finished products must be loaded/unloaded at different jetties. Due to the high operating cost of crude oil tankers, ships must be processed immediately after arrival.

6.5.2 Network Problems

Applications in this group deal with generalised versions of the warehouse assignment problem already studied as a constraint problem in [VC88]. Limited capacity of warehouses or multiple product lines add new types of constraints to this otherwise well-understood problem from OR.

Locarim (France Telecom, Telesystemes, COSYTEC). This application lays out the computer/phone network in large buildings. Telephone connections in different rooms must be linked together via concentrators, which have limited capacity. Cables must be run according to the building layout, which is scanned in from architectural plans. The system is used by France Telecom to propose a cabling and to estimate cabling costs for new buildings. This application was developed by the software house Telesystemes together with COSYTEC.

Distribution Planning (EBI). The Turkish company EBI has build a distribution planning system which determines the placement of different products in selected warehouses. Customers for multiple products are served from these warehouses. The objective is to minimise storage cost while achieving customer satisfaction with small delivery delays.

PlaNets (Enher, UPC/CSIC). This application controls the reconfiguration of an electrical power network in order to perform repairs and maintenance on some segments of the network. The system re-routes energy in order to minimise end-user problems respecting multiple electrical and other constraints. The program was developed at the university of Catalonia in Barcelona for the power company Enher [CGR95].

Banking Network (ICON). ICON, an Italian software house, has developed a system for facilities and load distribution in the Italian inter-banking network [CF95]. The program distributes applications over a network of mainframes in order to provide services to customers while balancing and minimising network traffic.

CLOCWiSe. The Esprit project CLOCWiSe (IN 10316I) is using CHIP for the management and operational control of water systems. A first implementation is under way at Bordeaux in France. The constraint model is being developed by the university of Catalonia in Barcelona.

6.5.3 Production Planning/Scheduling

Production planning and detailed scheduling are major application areas of the CHIP constraint system. Typical constraints are temporal sequences and limits, resource capacity limits and resource allocation problems. For a more detailed view on scheduling with constraints see for example [Sim95] [Sim97] [Sim98].

Plane (Dassault Aviation). Dassault Aviation has developed several applications with CHIP. PLANE [BCP92] is a medium-long term scheduling system for aircraft assembly line scheduling. The objective of the program is to decide on changes of the speed of production for the assembly line, called the cadencing of production. By changing the speed often, production can follow varying demand more closely, decreasing inventory costs. On the other hand, each change in speed incurs a large cost for assigning/reassigning personnel and material.

Made (Dassault Aviation). Another application developed by Dassault, MADE [CDF94] is a detailed scheduling system for a complex work-cell in the factory. The program controls a cutting press and related machines in a work-shop. As part of the scheduling, the system has to find a 2D placement of different products on sheets of metal, corresponding to different orders which are processed at the same time on the press.

Coca (Dassault Aviation). The COCA system of Dassault is used for production step planning. The program decides which operations on one module of the aircraft can be performed together, depending on the rotational orientation of the aircraft and the place on the module where the operations are performed. Precedence constraints and safety rules must also be taken into account.

Production Schedule (Amylum/OM Partners). This program schedules production lines for a glucose producing factory in Belgium. Main constraints are the set-up restrictions, which exclude certain product sequences, and machine choices, which change production rates. The objective is to find a compromise between achieving due-dates for orders and using long production runs to minimise operating cost.

SAVEPLAN (ATOS). The French software house ATOS has redeveloped an existing, FORTRAN based scheduling and inventory control system with the help of the CHIP constraint solver inside an object based development environment.

MOSES (COSYTEC). The MOSES application has been developed by COSYTEC for an animal feed producer in the UK. It schedules the production of compound feed for different animal species, eliminating contamination risks and satisfying customer demand with minimal cost. This system was developed in 1997 and is now operational in 5 factories, parts of its constraint model are described in [Sim98].

6.5.4 Transport

The transport area contains many different types of interesting constraint problems. A major concept is tour planning, finding for example the optimal sequence of deliveries for a set of lorries which transport goods from factories to customers. The CHIP *cycle* constraint [BC94] was especially developed to model this type of problem.

EVA (EDF/GIST). EVA is used by EDF (the French electricity company) to plan and schedule the transport of nuclear waste between reactors and the reprocessing plant in La Hague. This problem is highly constrained by the limited availability of transport containers and the required level of availability for the reactors. The program was co-developed by EDF and the software house GIST.

Transport planning (EBI). As a second stage of the project described in section 6.5.2, EBI has developed a tour planning system for multiple warehouses/customers with capacity constraints on the lorry fleet. Each lorry can transport a limited amount of goods from some warehouses to customers. The sequence of deliveries is controlled by the urgency of the order and the objective to minimise travel costs.

DYNALOG. The DYNALOG Esprit project (26387) studies the use of constraints and high-performance computing for general logistics and distribution problems. Research there is focused on large-scale problems and the use of parallelism to speed up constraint reasoning.

COBRA (NWT/PA Consulting/COSYTEC). Another type of transport problem is solved with COBRA [SC98]. This program generates diagrams (work plans) for train drivers and conductors of North Western Trains in the UK. For each week, around 25000 activities must be scheduled in nearly 3000 diagrams, taking a complex route network into account. On benchmark problems, improvements of 2-3 % were obtained compared to the previous solution.

LCCR (Eurocontrol/COSYTEC). Yet another transport planning problem is handled in the LCCR system. Eurocontrol is responsible for allocation air-traffic control sectors to all flights in the European airspace. Due to severe capacity limitations, flights must be allocated start and landing times such that the capacity of the sectors is not exceeded. This demonstrator is still under development, using the C++ version of CHIP.

6.5.5 Personnel Allocation

Another important application group are personnel assignment problems. Applications in this domain are characterised by the variety of rules and restrictions which must be taken into account. These constraints arise from government regulations, union contracts or technical limits, and can become very complex.

EPPER (Servair). The EPPER [DD96] system developed for the catering company SERVAIR by the software house GSI performs personnel planning and assignment for the French TGV train bar/restaurant personnel. It creates a four week planning, assigning agents of the correct qualification to different catering jobs in the trains. The assignment respects travel times, rest periods and other hard constraints and tries to balance the overall workload for all agents.

TAP-AI (SAS Data/COSYTEC). TAP-AI [BKC94] is a planning system for crew assignment for the airline SAS. It schedules and reassigns pilots and cabin crews to flights and aircraft respecting physical constraints, government regulations and union agreements. The program is intended for day-to-day management of operations with rapidly changing constraints.

DAYSY (Lufthansa, SEMA, COSYTEC, U. Patras). Another Esprit project, DAYSY (8402), aims at developing a personnel re-assignment system for daily operations of an airline. It reacts to delays, cancellation or addition of flights or changes in the availability of personnel by changing the existing crew assignment to satisfy the new constraints. The project was undertaken by Lufthansa, Sema Group, COSYTEC and the university of Patras, Greece.

OPTISERVICE (RFO/GIST/COSYTEC). The system [Col96] generates a personnel assignment for each of the TV stations of RFO in the French overseas departments. It allocates qualified personnel to each of the tasks in the station, from reporting events to providing 24h service for technicians. The constraint system is combined with a rule-based module to identify working-time limits based on the different types of working contracts.

Gymnaste (UJF/PRAXIM/COSYTEC). The Gymnaste [CHW98] system produces rosters for nurses in hospitals. An important aspect in this application is the possibility to modify rules and regulations interactively, as well as taking multiple social factors into account. The program is operational in multiple hospitals in France.

MOSAR (Ministère de la JUSTICE). The MOSAR application generates the roster of prison guards in France in 10 regional sites for 200 prisons. It generates provisional plans for a one year period and helps to determine personnel needs. It was developed by Cisi and COSYTEC. The objective of the system is a reduction of overtime cost and an increase of personnel satisfaction by a more balanced, impartial rostering process.

6.5.6 Others

There are quite a number of CHIP applications for other application areas, ranging from database query optimisation to military command and control systems. One of the first large scale CHIP systems in daily use was the SEVE application described in the next paragraph.

SEVE (CDC). SEVE is a portfolio management system for CDC, a major bank in Paris, developed in-house. It uses non-linear constraints over rationals and finite domain constraints to choose among different investment options. The system can be used for "what-if" and "goal-seeking" scenarios, based on different assumptions on the development of the economy.

DSP Design. Another area where finite domain constraint can be used to advantage is the design of digital signal processing (DSP) hardware. The problem is to implement some filtering algorithm with the right mix of hardware modules. The trade-off between speed/size/cost leads to multiple scheduling and resource allocation problems which have to be solved in order to find a good design compromise. Very impressive results with CHIP are described in [SGK99].

6.6 Case Studies

In this section we present several case studies in more detail. The first application is a production scheduling system in a chemical factory in Belgium, the second a scheduling and simulation tool for oil refineries, the third an integrated transport planning and scheduling system for a food company in the UK.

6.6.1 ATLAS

The ATLAS system is a scheduling application for one part of the MONSANTO plant in Antwerp. The program schedules the production of different types of herbicides. The production process is shown in figure 6.5. Production basically consists of two steps, a batch formulation part and a continuous packaging (canning) operation. In the formulation part, batches of several ten thousand litres of chemical products are produced. Each formulation batch takes several hours. Several processors with different characteristics are available. After the formulation, the product is kept in buffer tanks with a very limited capacity. Exceptionally, it can be stored in movable containers for a limited time period. From the buffer tanks, products are fed to the different canning lines where the products are filled in bottles and cans of different sizes. The canning lines differ in speed, manpower requirements and the type of products that they can handle. The bottles and cans are packed in cartons and palettes and stored in a holding area before delivery. Both the storage area for packaging material and the holding area are limited in size, which imposes constraints on the production process.

Besides many constraints associated with machine choice, set-up times and cumulative manpower constraints, producer/consumer constraints [SC95] play a major role in the problem solver. Intermediate products must be produced in the right quantity before they can be used on the packaging lines, and packaging material must be available in the warehouse before a task can be started.

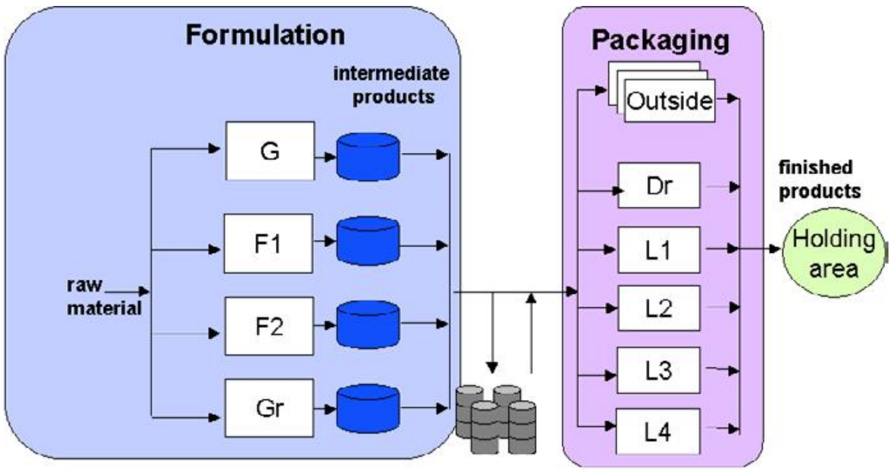


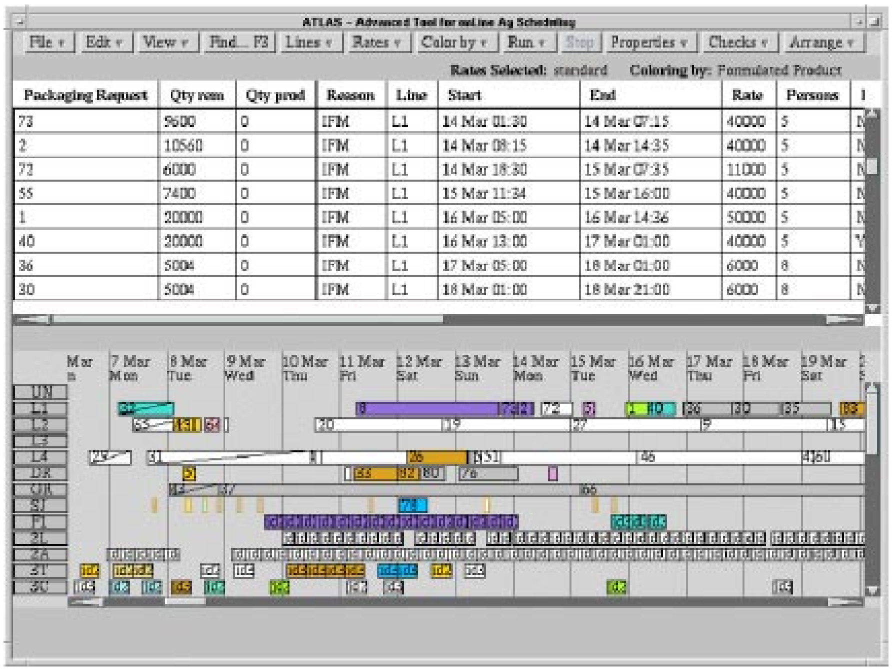
Fig. 6.5. ATLAS production process

The system works with data sets of several dozen intermediate products and more than one thousand packaging materials. A typical schedule may contain several hundred batches and canning tasks split into many sub-tasks to handle the stock level constraints with an eight hour resolution. In addition, several thousand constraints are needed to express other parts of the scheduling problem.

There is a production planning problem for the batch formulation part, where the system determines the correct number of batches for intermediate products and their optimal sizes. Since the storage tanks for intermediate products are limited in number and size, the correct amount of each product must be formulated to avoid left-overs which would block the storage tanks.

In figure 6.6, we show an example screen dump of the ATLAS application. The schedule is shown both in the form of a Gantt chart and as a textual list. The user can interact with the program via direct manipulation of items on the screen. A number of different views is provided to manipulate schedules, resource availability or stock levels.

The system is used by several users with different responsibilities. The scheduler for the factory works together with people responsible for scheduling the formulation and the inside and outside packaging lines. Other persons control the inventory and are responsible for ordering packaging materials. These users continuously exchange information via the system. An overview of the dynamic scheduling environment is given in figure 6.7. Orders and sales forecasts are added to the data base from external sources. Information about stocks of packaging material and deliveries are also available, shop floor data is entered to update the current situation on the manufacturing side. The current production schedule is updated on demand and is then used to determine new requirements of raw materials and packaging materials. The program can be used on a "what if" basis in order to see the impact of accepting new orders or changing delivery due dates.



Rates Selected: standard **Coloring by:** Formulated Product

Packaging Request	Qty rem	Qty prod	Reason	Line	Start	End	Rate	Persons	I
73	9600	0	IFM	L1	14 Mar 01:30	14 Mar 07:15	40000	5	N
2	10560	0	IFM	L1	14 Mar 08:15	14 Mar 14:35	40000	5	N
72	6000	0	IFM	L1	14 Mar 18:30	15 Mar 07:35	11000	5	N
55	7400	0	IFM	L1	15 Mar 11:34	15 Mar 16:00	40000	5	N
1	20000	0	IFM	L1	16 Mar 05:00	16 Mar 14:36	50000	5	N
40	20000	0	IFM	L1	16 Mar 13:00	17 Mar 01:00	40000	5	Y
36	5004	0	IFM	L1	17 Mar 05:00	18 Mar 01:00	6000	8	N
30	5004	0	IFM	L1	18 Mar 01:00	18 Mar 21:00	6000	8	N

Fig. 6.6. The ATLAS application

The system architecture of the ATLAS system is shown in figure 6.8. The application runs on a UNIX workstation together with an Oracle data base. The different users connect to this application via their personal computers running an Xwindows emulation. The workstation is also connected to diffe-

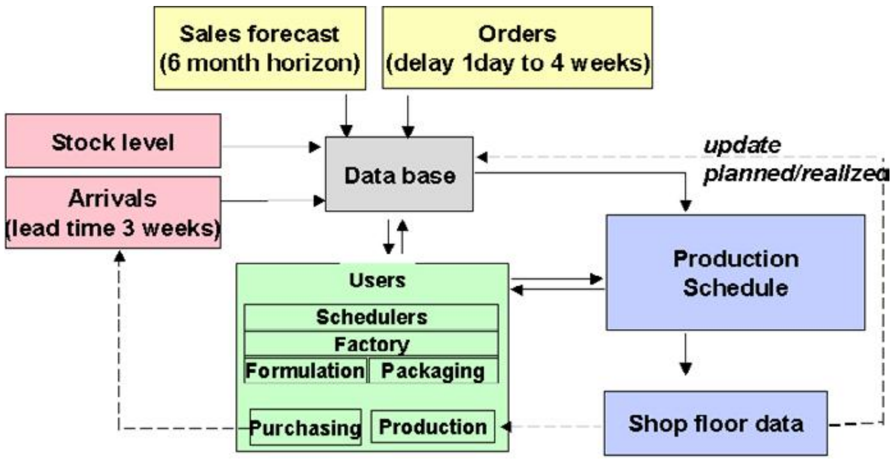


Fig. 6.7. The scheduling process

rent mainframe computers which keep information about orders, stock levels and deliveries. The master schedule is kept inside the data base, but each user can create and store local scenarios in file form. These scenarios can be "what-if" type evaluations, or alternative production plans, which can be exchanged between users.

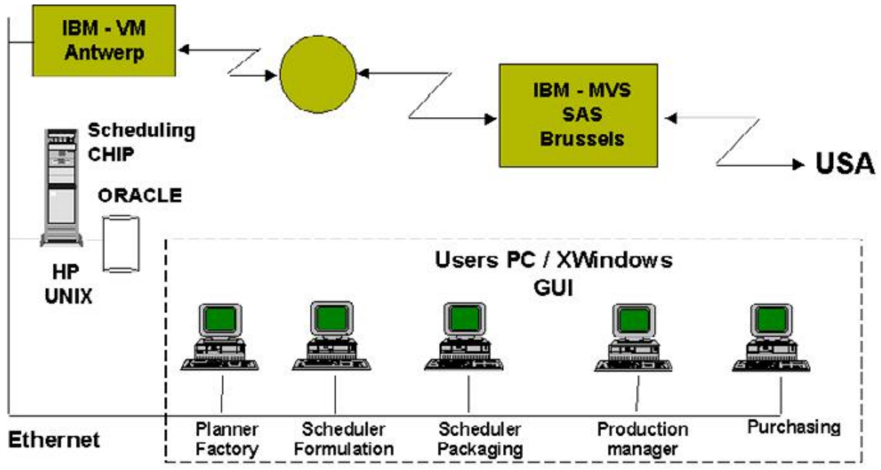


Fig. 6.8. ATLAS system architecture

The ATLAS system is operational since July 93, the complete project ended in spring 94. As an end-user system, the constraints are not visible to the user, all interaction is via the Gantt chart and stock level diagrams.

The user can move tasks manually and override the automated scheduler or change its strategy.

The system shows the type of complex scheduling problem which is well suited to the constraint approach. It is a real-life, not a pure problem, and does not fit easily into existing OR problem classifications. On the other hand, solutions which satisfy most constraints are hard to find. There are many forms of interaction between constraints and conflicts between different constraint types which can not be resolved automatically. The system is used as a decision support system, which helps the human scheduler, but does not replace him.

In important aspect during development was the possibility to incrementally express the constraints of the program. Rather than starting with a complete set of conditions, constraints were added by groups to understand the effect and requirements of each constraint type.

We discuss some implementation aspects below in section 6.8. A significant development effort was required to model and solve the problem and to generate the integration and graphical interface parts. A major revision of the application was performed in 1998 to provide an interface with SAP/R3.

6.6.2 FORWARD

The second case study is the FORWARD [GR97] system, a scheduling and simulation tool for oil refineries. It is used to plan refinery operations for production changes on a rather detailed level for next three days and at an overview level for the next three weeks. The system is currently in use at three sites in Europe and two refineries in Asia. Each refinery has its own particular features and operation procedures. This means that the program must be extensively customised for each site. The first installation is operational since September 94. FORWARD is a joint development of the engineering company TECHNIP and COSYTEC. It combines a non-linear simulation tool written in FORTRAN with a scheduling and optimisation part, which were developed in CHIP together with the graphical user interface.

To explain the function of FORWARD, we present an (idealised) overview diagram of a refinery in figure 6.9. Crude oils arrive by tanker or pipeline and are stored in large tanks in the tank farm. Different types of crude may be mixed to feed the process part of the refinery, starting with the crude distillation unit (CDU). A large number of other units are connected to different branches of the CDU output. They are used to maximise the percentage of light (valuable) products, which are obtained from the crude oil. The mixture of crude oils determines the throughput of the processing units and thus the economic efficiency of the refinery.

At the end of the process part semi-finished products are stored and then blended together to obtain commercial products like gasoline of defined qualities. Up to 12 different components can be used to generate gasoline for example. The ratio of the different products determines the cost of the

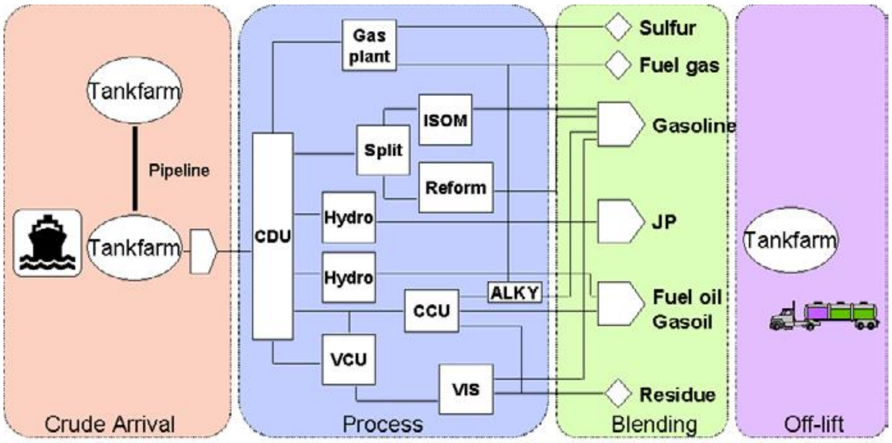


Fig. 6.9. Refinery schematic diagram

finished product and therefore the margin of the refinery against the market price. At the output side of the refinery, the off-lift of the finished products must be scheduled to satisfy customer demand and optimise resource use (pumps, berths, etc).

A refinery will have perhaps 30 different types of processing units, 250 tanks of different sizes in different parts of the refinery, connected by thousands of pipes and pumps which allow many, but not all product movements.

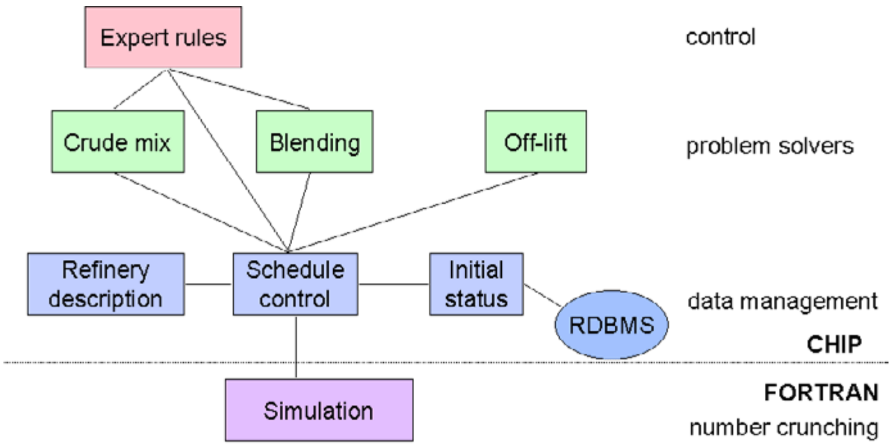


Fig. 6.10. FORWARD system modules

Oil mixes with non-linear blending laws for many properties. The FORWARD system tracks around 70 measured values and uses up to 1000 data points for each oil mix in the system. There is a library of several hundred crude oils, defining their properties in detail. The FORTRAN part of the FORWARD system uses these data and complex non-linear models of processing units to calculate flow-rates, yields and properties for each unit in the refinery. Different operating modes determine the type of behaviour expected from the unit.

The system modules in the FORWARD system are shown in figure 6.10. The *simulation part* in FORTRAN handles the calculation of unit yields and flow rates. It exchanges information about the refinery structure with the *scheduling control system.* This event processing system written in CHIP is also linked to the other parts of the system. The *refinery description module* is a graphical editor to enter and modify refinery descriptions interactively. All units, tanks and pipes in the refinery are entered using this module. In the *initial status module*, data from the refinery data base are loaded to initialise the simulation. An automatic comparison with the results of previous simulation runs detects inconsistencies and unforeseen events. On top of this data management part, there are several problem solvers, written in CHIP. One module controls the *mix of crude oils* in order to maximise the throughput of the CDU. A large non-linear optimisation module is responsible for *blending optimisation.* It is implemented on top of the linear rational solver of CHIP. For each blend, it calculates the optimal recipe to produce end products with very strict quality limits at minimal cost. As this optimiser is used inside the simulation tool, we often encounter overconstrained problems, which allow no solution. A special explanation tool, written as a meta program on top of the problem solver, applies constraint relaxation in a variable, four-level hierarchy. It either finds the maximal achievable quantity or determines the missing component.

Another problem solver is responsible for the scheduling of *ship arrivals* at the refinery. This has to take the limited berthing capacity at the refinery into account as well as the storage limits for finished products.

On top of these modules, an *expert rule system* allows to recognise and react to standard situations in the refinery. These rules explain for example which tank to use next when a crude oil tank becomes empty or full. The rules are entered graphically using a special rule editor, which guarantees syntactic correctness of the rules.

The FORWARD system clearly shows the flexibility obtained by the CLP approach. The five current users belong to different companies, with different management styles and slightly different objectives in running their refinery. Each site requires custom modifications to take new units or special physical properties into account. At some sites, pipeline operations must be scheduled concurrently with the refinery operations. Different users use different blending laws which directly affects the blending optimiser. This problem solver is customised for each user to reflect their different view point. The interfaces to other tools and to the data base are customised as well. As a large part of the system is developed in CHIP using CHIP++ objects, it is easy to extend or modify the function of different modules.

6.6.3 TACT

The TACT application was developed for a food-industry company in the UK [SCK00]. The problem can be described in a simplified form as follows (see figure 6.11): In the food industry, birds (chicken and turkeys) are grown on a large number of farms. When the birds reach a certain age, they are transported to the factories, where they are killed and processed for different types of products. For each day, an order book describes the different farms to be visited, the number of birds of different types to be collected and the order in which the birds must be delivered at the factories. The arrival of the lorries at the factories must be scheduled carefully, in order to avoid either running the factory out of stock or to deliver the birds too early, which would be unacceptable for quality control reasons.

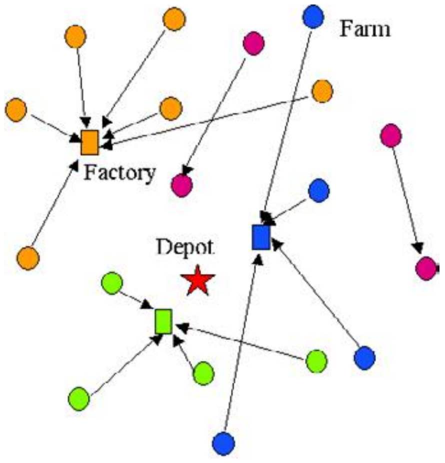


Fig. 6.11. TACT problem

In addition, other transport must be scheduled between breed and grow-out farms to distribute the next generation of birds at the correct time.

The order book is usually known well in advance, but often changes at the last minute in order to react quickly to some new event. Reaction time is crucial, since the affected personnel must be notified in time of any changes in the plan.

The company uses its own fleet of vehicles for the transport, but sometimes must hire in extra drivers or vehicles to handle the work load. The farms are at a distance of between 5 minutes and 3 hours driving time from the factories, so that geographical locations play an important role in the scheduling. Some farms can only be reached by small lorries, since they are at the end of difficult roads. For a number of farms, environmental constraints limit the time when lorries can drive to the farms.

The actual transportation process requires a number of interdependent resources:

- bird catching teams
- drivers
- lorries
- trailers
- fork lifts
- fork lift trailers

We will now describe some of the constraints attached to the different types of resources.

Catching teams. The catching team is a group of persons which on the farms catches the birds and collects them into crates. The rate of work, their working time and their preferred work (catching turkeys or chicken, male or female) varies for each team. The teams start work at the depot, drive to a farm with a mini-bus, collect a number of birds there and either continue to another farm or return to the depot. For sanitary reasons, it may be necessary to return to the depot to change before going to another farm. The total working time and the number of crates collected by day are limited, and between two working days the teams must have a certain rest time. This means that a team which works late on Monday can not be used very early on Tuesday. If the work load can not be handled by the given number of teams, it is possible to hire in outside help at a significantly higher cost.

Drivers. The drivers will always clock on at the depot, collect a lorry and make some trips to farms and from there to the processing sites. At the end of the day, the driver must drop the lorry at the depot and clock off there. In a normal day, they will make between two and five trips (depending on distances, etc). After delivery at a factory, the lorry is cleaned and can be used for another trip. The driver may change lorries several times during a day, he is not assigned full-time to one particular vehicle. They also do not have fixed clock-on and clock-off times, but follow a roster, which for each week gives the sequence in which drivers must clock-on. The usual working rules apply, i.e. maximal number of hours working, maximal number of hours consecutive driving, maximal mileage per day are all limited. While a lorry is loaded on a farm, the driver can take the legally required rest period. At the end of a trip, the driver may be required to change the lorry configuration, i.e. drop a trailer or change the large crates for turkey with smaller crates for chicken. This may require an empty trip to another factory in order to pick up these new modules.

Lorries. The factory fleet consists of different types of vehicles, some single lorries of 24 ton weight, lorries with trailers with a total weight of 32, 35, 38 or 42 ton total weight plus some articulated trailers which can be used with outside haulage. Not all lorries are available at all times, there are times reserved for maintenance or inspections. The birds are transported on the lorries in crates which are placed in modules. These modules are loaded and unloaded with fork lifts.

The number of lorries available is given. If for some reason more lorries are required, they can be rented, but at a significant extra cost.

Previous solution. Before the introduction of the system, the scheduling of the transport problem was done by hand. The scheduler started in the morning using the current order book for the next day. Usually, it took about 6-8 hours to find a solution. If the order book was changed in a significant way, much of the work had to be redone, delaying the publishing of the schedule. The two experts in the company could produce a very good schedule in most situations, but it was nearly impossible for an outsider to understand the process. Often, some rule would be bent in order to find a solution more quickly or to patch a solution after some change in the input data. As a result, it would be often unclear which constraints were actually handled correctly in a given schedule.

Solution Approach. Given the complexity of the transportation problem, it is understandable that only a decomposition can lead to manageable sub-problems of feasible size. In the manual solution, the scheduler were cutting the problem into three sub parts. In TACT, we follow the same approach (see figure 6.12).

Trip Cut. In a first step, called trip cut, the number of trips and the different lorry types for each entry in the order book is determined. This creates a set of trips, each transporting a given number of birds on a particular type of vehicle. The solver of this step is a small integer programming module written in CHIP. The constraints consist in a number of equations and inequalities and we are interested in a solution which utilises an integral number of lorries of each type while generating the smallest number of trips possible.

Line Balancing. The second step, called line balancing, is to schedule the trips generated in the first steps in such a way that the supply for the factories is guaranteed and that operational constraints of the loading and transport are taken into account. The main constraints in this problem are producer/consumer constraints for the different bird types. The birds should arrive just in time at each factory, too early and the limited storage space in the factory is exceeded, too late and a factory with several hundred employees grinds to a halt. Figure 6.13 below shows a typical constraint curve for one bird type over the period of a day.

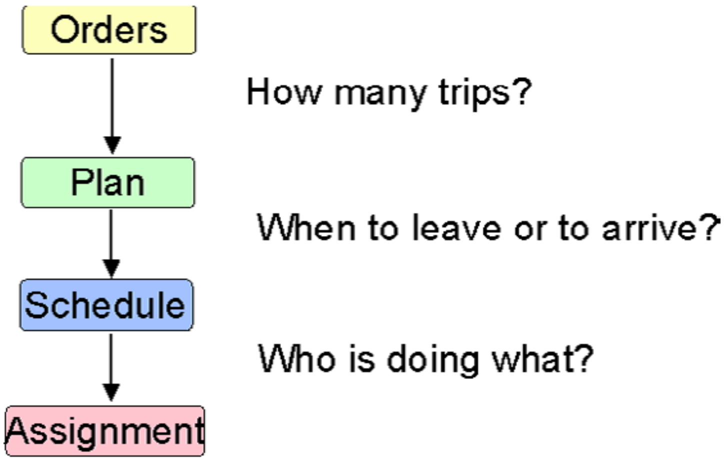


Fig. 6.12. Solution Approach

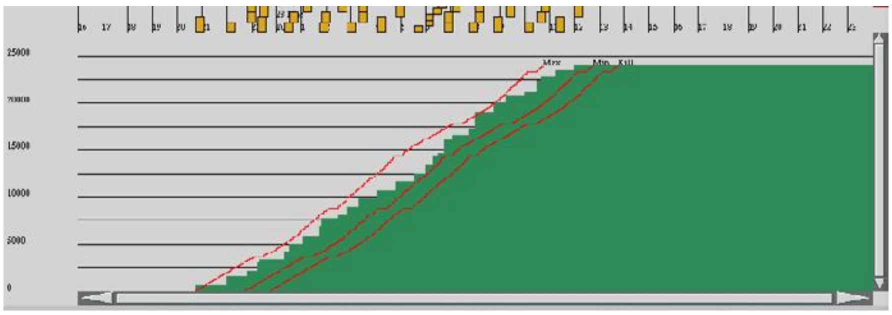


Fig. 6.13. Producer/consumer constraint for factory intake

Assignment. The third step consists in assigning individual resources to all activities. Depending on the type of resource, the timing may have to be re-evaluated at the same time. The constraint model is a typical multi-resource scheduling and assignment problem. The main resource constraint is handled by *diffn* constraints, variable transport (set-up) times are handled with the *cycle* constraint and overall resource limits are expressed with (redundant) *cumulative* constraints.

After each step, the results are presented to the user in the graphical user interface, where he can react to the system. Quite often, the user may decide to re-run the trip cut (first step) with some changes in the parameters before continuing with the second phase.

Each of these phases are run for the orders of a single day, although the generated schedules (30 hours) overlap in time. For each day, the schedule of the previous day is considered as fixed work in progress. Only the fork lift planning is made for one week at a time, in order to minimise the transport required.

Handling over-constrained problems. An important aspect in the resource assignment is the handling of over-constrained problems. If the resource assignment doesn't find a solution in a reasonable time this can either mean that there is no solution or that the heuristic is not good enough to find it. If there is no solution, this can be because there are not enough drivers, or not enough lorries of some type or not enough catching teams. It is important for the user to find the real cause of the problem, in order to avoid costly mistakes like renting another lorry when more drivers for the existing lorries are required.

In order to provide some form of automatic explanation, the system automatically relaxes different types of constraints if no solution can be found.

Another specialised method is the calculation of lower bounds for some type of resource in independent solvers. This method covers the given amount of work with work profiles of a given duration, ignoring all other constraints. A detailed description together with the constraint model of this lower bound calculation is given in [Sim96a]. Note that the latest version of CHIP includes these lower bound calculation as options for the global constraints cumulative, diffn and cycle.

The user can modify the schedule interactively and then check if the modifications satisfy the constraints. In some situations, the user may wish to violate a constraint in the system in order to solve a particular problem. The automated solver, for example, does not allocate over-time to catching teams. The scheduler, on the other hand, can decide to use over-time for some team and to discuss this with the affected persons, rather than calling in outside contractors for this work. This negotiation process can not be automated, but is crucial to finding a good compromise between different objectives.

Evaluation. The system has been operational since beginning of 1995 and has completely replaced the manual scheduling process. While a manual solution required up to 8 hours of work, it is now possible to find solutions in as little as 15 minutes (about 5 minutes of constraint solving). This gives more flexibility to the scheduler who can devote more time reacting to sudden changes in the order book or to compare different scenarios balancing contradictory objectives.

The TACT application is not used as a black-box problem solver, but rather as a decision support tool for the scheduler. It speeds up the generation of schedules and ensures that constraints and regulations are taken into account. The scheduler can control the different solvers by enabling/disabling constraints or selecting different heuristics.

It is difficult to compare resource utilisation results before and after the introduction of the system as business in general has changed significantly. But it is clear that the tool allows to analyse a problem earlier and in much more detail so that expensive last minute changes are avoided.

A return on investment in 6 month has been quoted by the customer, in particular due to reduced capital expenditure on new lorries.

In five years of operation, a number of changes were introduced into the system to handle new constraints or restrictions. The company has updated its lorry fleet, so that constraints on capacity and farm access had to be changed. The contractual status of the catching teams changed, so that work-rules, time limits and overtime pay had to be revised. The loading constraint balancing full and partially empty vehicles was completely revised. All these changes could be implemented quite rapidly without affecting other parts of the constraint model.

6.7 Application Framework

We now briefly discuss some aspects of the application framework used for many of the applications shown above. A typical decision support application consists of different parts. A central data model is used to drive the graphical user interface, the problem solver and reporting modules. The data are stored in a data base which exchanges data with the internal data model. Libraries are used for many of these functions in order to minimise application specific code.

The application framework for CHIP provides such building blocks for the data management and graphical user interface. Based on a central data model as a data description file, it automatically creates CHIP++ class descriptions, generates data base tables and code to upload and download data from the data base. In the same way, it is used to manipulate graphical interface modules like lists or dialog panels. All these components share the data description which is a single point of reference for the data model of the application.

The framework also contains libraries for often required graphical tools like Gantt charts or diagrams.

The system is general enough for different application domains, it has been used for problems of manufacturing, transport, chemical industry production scheduling, or human resource management. There are obvious benefits in the code size and development speed, but also improved quality and a decrease in maintenance requirements.

6.8 Analysis

In this section we want to evaluate some aspects of the application development with constraints. We first look at an analysis of the different parts of the ATLAS application and then compare a number of large scale CHIP applications.

6.8.1 Needs of an Industrial System

Table 6.2 shows the percentage of the overall size of different parts of the ATLAS application. The largest amount is taken up by the data base and integration parts, which are developed using SQL*Forms and Reports from Oracle. Of the CHIP part, the application specific graphics and graphical libraries on top of XGIP take a third of the overall program size. The problem solver only accounts for 9%, while the remainder is split between fixed parts for I/O, data management and dialogs.

Table 6.2. Percentage of application size

Part of application	Percentage of overall size
Data base/integration	46%
Graphics	19%
Graphics libraries	14%
Problem solver	9%
Static program parts (I/O, dialog description)	6%
Others	6%

The table only gives the percentages of code size of the finished application. In terms of development effort, a larger percentage must be attributed to the constraint solving part. Prototyping and experimenting with different strategies leads to re-writes or modifications of major parts of the constraint module. In terms of overall effort on the application, the solver accounts for perhaps 20% of the overall development time.

The problem solver is also the part with the high development risk. Very often, it is not clear at the beginning of the project which aspects of the system can be integrated in the solver. Yet without a problem solver, the overall application does not perform its job. This means that prototyping and risk reduction are key aspects to a successful CLP project.

But even taking this into account, the table clearly shows the importance of the integration and data manipulation parts of the finished system. With the application framework, we are working on reducing the amount of special purpose code required for this part of the system in the same way as the constraint system has reduced the amount of work on the problem solver.

6.8.2 Application Comparison

The following table 6.3 compares a number of large scale CHIP applications co-developed by COSYTEC. The entries are given in temporal sequence of their project start. The table shows the name of the system and the constraint solver used inside the program (F means the finite domain solver, R the linear rational solver). The next two columns state whether the program contains a graphical user interface and data management parts. The number of lines of the complete application is given next. An asterisk means that the complete system contains parts written in other languages, which are not counted here. The last two columns give the number of lines in the constraint solving part of the application and the elapsed project time. This number is not the total size of the project in man months.

Table 6.3. Application comparison

System	Solver	GUI	Data Management	Total Lines	Lines solver	Duration
Locarim	F	yes	yes	70000	3000	24 month
Forward	R,F	yes	yes	60000(*)	5000	24 month
ATLAS	F,R	yes	no	15000(*)	4000	18 month
TAP-AI	F	no	no	7000(*)	6000	6 month
TACT	F	yes	yes	36000	7000	10 month

It is interesting to see that graphical user interfaces and data management modules account for a large percentage of the overall code size. Building a complete end-user system also requires a significant time period for specification and changes of the business process to use the new system to advantage. The TACT application shows how the use of the application framework increasingly reduces both project time and code size.

The constraint solver parts are of a rather similar size. As new global constraints are added to the system, we can handle more and more complex problems which were unmanageable before. We therefore see an increase of complexity in more recent applications which has no significant impact on the project duration.

6.8.3 Why Use Prolog for CLP?

All applications mentioned above use the PROLOG version of the CHIP system. While the declarative structure of logic programming is often advocated as a main advantage of Prolog, we find other aspects of at least equal importance.

For application development, the interpreting environment of Prolog is very important. Changes and revisions can be loaded without recompiling and linking the complete system by reconsulting small parts of the system. This speeds up the development of the problem solver as well as the coding of the graphical user interface.

The built-in relational form of Prolog is also very convenient to store data and parameters inside the application.

The backtracking mechanism of Prolog is most important for developing complex assignment strategies in the problem solver. While predefined search procedures can give satisfactory results on small problems, they are not sufficient to solve large, complex problems. Developing such search strategies in a conventional language like C or C++ dramatically increases complexity.

Another useful aspect of logic programming are meta-programming facilities which can analyse programs and generate new code dynamically. Several of the presented applications use model generator programs or diagnosis engines as explanation facilities. Expert rule systems written in Prolog (with suitable graphical interfaces) allow end-user programming for example in the FORWARD or OPTISERVICE systems.

On the other hand, there are several problems which must be overcome when using logic programming for large scale application programming. The largest problem clearly is the lack of acceptance of PROLOG in the industry, which creates both commercial and technical problems. Writing efficient logic programs is clearly possible, but requires training and experience. This problem is aggravated for constraint programming where there is no strict separation between modelling and implementing tasks.

Other problems are of a more technical nature. Prolog does not immediately support a graphics model based on callbacks. This requires exchange of global data between queries, which is normally not well supported in Prolog. It also lacks proper data abstraction tools. Symbolic terms are rather weak data structures, which cause problems for re-use and specialisation of existing code. These problems are overcome in CHIP with the help of the CHIP++ object layer, which adds feature-terms, objects and class structure to PROLOG.

6.9 Does CLP Deliver?

In section 2.1, we mentioned several key advantages which were claimed for constraint logic programming. *Short development time* was the first of these advantages. This is clearly true for many of the prototypes which have been developed very rapidly in CHIP in order to understand and define the problem to be solved. For large scale applications, time limits are usually given by the project framework and possible changes to the business environment. Basically constraint systems enable us to remove the problem solver from the critical path in the project planning. Compared to a standard approach, constraint applications also use much *less code* than conventional programs. The use of the application framework increases this tendency by also shrinking the amount of custom programming for other parts of a system like graphics or data management. The programs are also quite *easy to modify* and to extend. A good example is the FORWARD system, which has been quite easily adapted to four different users and problems as described above. The *performance* of the problem solver is typically not the limiting factor in the application.

Answers are achieved from within seconds to within a few minutes for the complex constraint problems. This is usually quite acceptable compared to the time required to enter data or to make manual modifications.

6.10 Limitations

The list of applications shown above indicates that constraint programming is already used for a large variety of application domains. But this experience with application development also indicates a number of limitations and shortcomings of the current tools.

6.10.1 Stability

The most common problem stated by users of the constraint systems is the sometimes unpredictable behaviour of constraint models. Even small changes in a program or in the data can lead to a dramatic change in the performance. The process of performance debugging, designing and improving constraint programs for a stable execution over a variety of input data, is currently not well understood.

6.10.2 Learning Curve

Related to this problem is the long learning curve that novices have experienced. While simple applications can be build almost immediately, it can take much longer to become familiar with the full power of the constraint system.

6.10.3 Problem Size/Complexity

Another problem aspect is the conflict between problem size and complexity. For many small, but very hard problems the constraint systems must perform as much propagation as possible in order to avoid expensive, blind search. For larger, but simpler problems this propagation is too expensive. Much time is spent in the constraint reasoning when a simple enumeration would be more efficient. If a large problem contains a number of smaller, but difficult sub-problems, the difficulties in problem solving increase.

6.10.4 Cost Optimization

A particular problem in many constraint models is the cost optimisation. Depending on the cost function, in particular for additive costs with many contributing factors, constraints may find rather weak lower bounds of the cost. In optimisation problems this often leads to difficulties to improve an initial solution, which is found with a heuristic. Systematic exploration of the whole search space is not possible due to the large number of choices to be explored.

6.11 Future Trends

In this section we will present some ideas for the further development of finite domain constraint techniques. These ideas are intended to be non-exclusive, there are obviously many other interesting approaches (for example the combination of modelling with global constraints and stochastic search methods, or other hybrid methods). We group the ideas in four areas, starting with modelling issues, followed by constraint reasoning and a section on search and optimisation. The important aspect of usability and ease of use is discussed in the last section.

6.11.1 Modeling

Current constraint languages like CHIP already use a rather powerful set of constraint primitives and global constraints. Further development will be driven by two main factors:

- Requirements for particular applications often lead to new constraints, which are then extended to satisfy the rules for global constraint expressed above.
- A general analysis of possible constraint types may lead to a novel and more systematic classification of constraint pattern. Some early research in that direction can be found in [SAB00] [Bel00].

A more general question concerns the modelling language used to express constraint problems. At the moment, most constraint systems are either extensions of a programming language (often Prolog) or libraries which are used together with conventional programming languages (C, C++). Currently there is work on introducing constraint modelling languages [VH98] similar to algebraic modelling languages known from Operations Research [FGK93].

Another approach to modelling is the use of visual tools to express and generate constraint programs. A wide range of methods is possible, starting from application specific visual tools to dedicated visual constraint programming languages. In the FORWARD refinery scheduling [GR97] system for example, the model of the refinery is entered in a graphical drawing tool and the constraint model is generated from the layout. In [Lab98], a graphical tool to express search methods has been presented. A first version of a graphical specification tool for CHIP was presented in [Sim97a].

6.11.2 Constraint Reasoning

As mentioned above, for many large scale problems we must find a compromise between the use of expensive, but powerful methods and the need to find a solution rapidly for simple problems. In a typical global constraint, up to 40 propagation methods are working together. The study of the interaction of these methods and their possible control is one of the most challenging problems in constraint development.

Another current topic is the integration of constraint programming with integer programming techniques [BK98]. This integration can take very different forms. One possibility is the introduction of a global constraint concept in integer programming and its use in a branch-and-cut scheme [Kas98]. A second approach is a redundant modelling with both global constraints and integer programming methods. A difficult question in this approach is the integration of the search techniques of both methods which are quite different.

6.11.3 Search and Optimization

Controlling search is probably the least developed part of the constraint programming paradigm. For many problems the choice of a search routine is crucial to performance, but it is generally developed on a ad-hoc basis. The possibilities of partial search and local search methods as alternatives to depth first chronological backtracking are also not full understood.

The difference to integer programming is quite striking. In integer programming, cost and cost estimation are driving search to the exclusion of most user control. In constraint programming, the cost estimation often is very weak and does not guide the search, so that user defined heuristics are needed to find solutions. A combination of both techniques, improved cost reasoning and user control over the search should provide dramatic improvements in solution quality for many hard problems.

6.11.4 Ease of Use

The most limiting factor for a more wide-spread use of constraint programming at the moment is probably the difficulty of understanding and mastering the technology. Part of this difficulty is the inherent complexity of the problems tackled, but much further development is also needed in methodology [Ger98] and ease of use. Visualisation and debugging tools like the ones developed in DiSCiPl [SA00] [SAB00] [SCD00] can help understand the crucial problem of performance debugging.

6.12 Conclusions

We have shown in this paper a number of large scale applications developed with CHIP, a constraint logic programming system and have seen that complex end-user systems can be built with this technology. On the other hand, constraint logic programming does not offer a "silver bullet" for problem solving. There is a significant learning period before the technology can be mastered. This also requires application domain knowledge to model problems and to express solution strategies. Other aspects, like integration and graphical user interface, form a significant part of the overall development requirements. Clearly, constraint logic programming is now an industrial reality which can be applied to many domains where high quality decision support systems are required.

Acknowledgments

This overview would not have been possible without the work of the team at COSYTEC together withtheir customers and partners in different projects.

References

[AB93] A. Aggoun, N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling Problems. *Journal of Mathematical and Computer Modelling*, Vol. 17, No. 7, pages 57-73 Pergamon Press, 1993.

[BKC94] G. Baues, P. Kay, P. Charlier. Constraint Based Resource Allocation for Airline Crew Management. In *Proceedings ATTIS 94*, Paris, April 1994.

[BBC97] N. Beldiceanu, E. Bourreau, P. Chan, D. Rivreau. Partial Search Strategy in CHIP. In *Proceedings 2nd Int. Conf on Meta-heuristics*, Sophia-Antipolis, France, July 1997.

[BBR96] N. Beldiceanu, E. Bourreau, D. Rivreau, H. Simonis. Solving Resource-Constrained Project Scheduling Problems with CHIP. *Fifth International Workshop on Project Management and Scheduling*, Poznan, Poland, April 1996.

[BC94] N. Beldiceanu, E. Contejean. Introducing Global Constraints in CHIP. *Journal of Mathematical and Computer Modelling*, Vol. 20, No. 12, pp 97-123, 1994.

[Bel00] Nicolas Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In Rina Dechter, editor, *Principles and Practice of Constraint Programming - CP 2000*, volume 1894 of *Lecture Notes in Computer Science*, pages 52–66, Singapore, September 2000. Springer-Verlag.

[BCP92] J. Bellone, A. Chamard, C. Pradelles. PLANE -An Evolutive Planning System for Aircraft Production. In *Proceedings First International Conference on the Practical Application of Prolog*. 1-3 April 1992, London.

[Ber90] F. Berthier. Solving Financial Decision Problems with CHIP. In *Proceedings 2nd Conf Economics and AI*, Paris 223-238, June 1990.

[BLP95] R. Bisdorff, S. Laurent, E. Pichon. Knowledge Engineering with CHIP - Application to a Production Scheduling Problem in the Wire-Drawing Industry. In *Proceedings 3rd Conf Practical Applications of Prolog (PAP95)*, Paris, April 1995.

[BK98] A. Bockmayr, T. Kasper. Branch and Infer - A Unifying Framework for Integer and Finite Domain Constraint Programming. *INFORMS J. Computing* 10(3) 287-300, 1998.

[BDP94] P. Bouzimault, Y. Delon, L. Peridy. Planning Exams Using Constraint Logic Programming. In *Proceedings 2nd Conf Practical Applications of Prolog*, London, April 1994.

[COC97] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kucken, editors, *Programming Languages: Implementations, Logics, and Programming*, volume 1292 of *Lecture Notes in Computer Science*, pages 191–206, Southampton, September 1997. Springer-Verlag.

[CDF94] A. Chamard, F. Deces, A. Fischler. A Workshop Scheduler System written in CHIP. In *Proceedings 2nd Conf Practical Applications of Prolog*, London, April 1994.

[CHW98] P. Chan, K. Heus, G. Weil. Nurse Scheduling with Global Constraints in CHIP: Gymnaste. In *Proceedings Practical Applications of Constraint Technology (PACT 1998)*, London, March 1998.

[CF95] C. Chiopris, M. Fabris. Optimal Management of a Large Computer Network with CHIP. In *Proceedings 2nd Conf Practical Applications of Prolog*, London, April 1994.

[Col96] C. Collignon. Gestion Optimisee de Ressources Humaines pour l'Audiovisuel. In *Proceedings CHIP Users' Club*, Massy, France, November, 1996.

[Col90] A. Colmerauer. An Introduction to Prolog III. *CACM* 33(7), 52-68, July 1990.

[CGR95] T. Creemers, L. R. Giralt, J. Riera, C. Ferrarons, J. Rocca, X. Corbella. Constrained-Based Maintenance Scheduling on an Electric Power-Distribution Network. In *Proceedings Conf Practical Applications of Prolog (PAP95)*, Paris, April 1995.

[DVS87] M. Dincbas, H. Simonis, P. Van Hentenryck. Extending Equation Solving and Constraint Handling in Logic Programming. In *Colloquium on Resolution of Equations in Algebraic Structures (CREAS)*, Texas, May 1987.

[DVS88] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693-702, Tokyo, 1988.

[DVS88a] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun and T. Graf. Applications of CHIP to Industrial and Engineering Problems. In *Proceedings First International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, Tullahoma, Tennessee, USA, June 1988.

[DSV90] M. Dincbas, H. Simonis and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming* 8, pages 75-93, 1990.

[DS91] M. Dincbas, H. Simonis. APACHE - A Constraint Based, Automated Stand Allocation System. In *Proceedings of Advanced Software Technology in Air Transport (ASTAIR'91)*. Royal Aeronautical Society, London, UK, 23-24 October 1991, pages 267-282.

[DSV88] M. Dincbas, H. Simonis, P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *Proceedings European Conference on Artificial Intelligence (ECAI-88)*, Munich, W. Germany, August 1988.

[DSV92] M. Dincbas, H. Simonis, P. Van Hentenryck. Solving a Cutting-Stock Problem with the Constraint Logic Programming Language CHIP. *Journal of Mathematical and Computer Modelling*, Vol. 16, No. 1, pp. 95-105, Pergamon Press, 1992.

[DD96] A. Dubos, A. Du Jeu. Application EPPER Planification des Agents Roulants. In *Proceedings CHIP Users' Club*, Massy, France, November 1996.

[FGK93] R. Fourer, D. Gay, B.W. Kernigham. AMPL - A Modelling Language for Mathematical Programming. The Scientific Press, San Francisco, CA, 1993.

[FHK92] T. Fruewirth, A. Herold, V. Kuchenhoff, T. Le Provost, P. Lim, M. Wallace. Constraint Logic Programming - An Informal Introduction. In *Logic Programming in Action*. Springer Verlag LNCS 636, 3-35, 1992.

[Ger98] C. Gervet. LSCO Methodology- The CHIC2 Experience. In *DIMACS workshop of constraint programming and large scale discrete optimization*, Rutgers University, September 1998.

[GR97] F. Glaisner, L.M. Richard. FORWARD-C: A Refinery Scheduling System. In *Proceedings Practical Applications of Constraint Technology (PACT97)*, London, March 1997.

[GVP89] T. Graf, P. Van Hentenryck, C. Pradelles, L. Zimmer. Simulation of Hybrid Circuits in Constraint Logic Programming. In *Proceedings IJCAI89*, Detroit, August 1989.

[HG95] W.D. Harvey, M.L. Ginsberg. Limited Discrepancy Search. In *Proceedings IJCAI95*, 1995.

[JL87] J. Jaffar, J.L. Lassez. Constraint Logic Programming. In *Proceedings 14th POPL*, Munich ,1987.

[JM93] J. Jaffar M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503-581, 1994.

[Kas98] T. Kasper. A Unifying Logical Framework for Integer Programming and Finite Domain Constraint Programming. PhD Thesis, Universitat des Saarlandes, 1998.

[KS95] P. Kay, H. Simonis. Building Industrial CHIP Applications from Reusable Software Components. In *Proceedings Practical Applications of Prolog (PAP95)*, Paris, April 1995.

[Lab98] F. Laburthe. Contraintes et Algorithmes en Optimisation Combinatoire. PhD Thesis, University Paris VII, 1998.

[MS98] K. Mariott, P. Stuckey. Programming with Constraints - An Introduction. MIT Press, Cambridge MA, 1998.

[Per91] M. Perrett. Using Constraint Logic Programming Techniques in Container Port Planning. *ICL Technical Journal*, May, 1991, pp 537-545.

[Sch97] C. Schulte. Oz Explorer: A Visual Constraint Programming Tool. In *Proceedings of the Fourteenth International Conference On Logic Programming*, Leuven, Belgium, pages 286-300. The MIT Press, July 1997.

[SND88] H. Simonis, N. Nguyen, M. Dincbas. Verification of Digital Circuits using CHIP. In G. Milne (Ed.), *The Fusion of Hardware Design and Verification*, pages 421-442, North Holland, Amsterdam, 1988.

[SD93] H. Simonis, M. Dincbas. Propositional Calculus Problems in CHIP. In A. Colmerauer and F. Benhamou, Editors, *Constraint Logic Programming - Selected Research*, pages 269-285, MIT Press, 1993.

[Sim95] H. Simonis. Scheduling and Planning with Constraint Logic Programming. *Tutorial Practical Applications of Prolog (PAP95)*, London, UK, April 1995.

[SC95] H. Simonis, T. Cornelissens. Modelling Producer/Consumer Constraints. In *Proceedings Principles and Practice of Constraint Programming (CP95)*, Cassis, France, September 1995.

[Sim95a] H. Simonis. Application Development with the CHIP System. In *Proceedings Contessa Workshop*, Friedrichshafen, Germany, September 1995, Springer LNCS.

[Sim96] H. Simonis. A Problem Classification Scheme for Finite Domain Constraint Solving. In *Workshop on Constraint Applications, CP96*, Boston, August 1996.

[Sim96a] H. Simonis. Calculating Lower Bounds on a Resource Scheduling Problem. In *Workshop on Constraint Programming, ASIAN96*, Singapore, December 1996

[Sim97] H. Simonis. Standard Models for Finite Domain Constraint Solving. *Tutorial at Practical Applications of Constraint Technology (PACT97)*, London, UK, April 1997.

[Sim97a] H. Simonis. Visual CHIP - A Visual Language for Defining Constraint Programs. *CCL II workshop*, September 1997.

[Sim98] H. Simonis. More Standard Constraint Models. *Tutorial at Practical Applications of Constraint Technology (PACT98)*, London, UK, March 1998.

[SC98] H. Simonis, P. Charlier. COBRA - A System for Train Crew Scheduling. In *DIMACS Workshop on Constraint Programming and Large Scale Combinatorial Optimization*, Rutgers University, New Brunswick, NJ, September, 1998.

[SA00] H. Simonis and A. Aggoun. Search-tree visualization. In P. Deransart, J. Maluszynski, and M. Hermenegildo, editors, *Analysis and Visualisation Tools for Constraint Programming*. Springer LNCS 1870, 2000.

[SAB00] H. Simonis, A. Aggoun, N. Beldiceanu, and E. Bourreau. Global constraint visualization. In P. Deransart, J. Maluszynski, and M. Hermenegildo, editors, *Analysis and Visualisation Tools for Constraint Programming*. Springer LNCS 1870, 2000.

[SCK00] H. Simonis, P. Charlier, and P. Kay. Constraint handling in an integrated transportation problem. *IEEE Intelligent Systems and their applications*, 15(1):26–32, 2000.

[SCD00] H. Simonis, T. Cornelissens, V. Dumortier, G. Fabris, F. Nanni, and A. Tirabosco. Using constraint visualization tools. In P. Deransart, J. Maluszynski, and M. Hermenegildo, editors, *Analysis and Visualisation Tools for Constraint Programming*. Springer LNCS 1870, 2000.

[Smo95] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[SGK99] R. Szymanek, F. Gruian, K. Kuchcinski. Application of Constraint Programming to Digital Systems Design. In *Workshop on Constraint Programming for Decision and Control*, Institute for Automation, Silesian University of Technology, Gliwice, Poland, June 1999.

[VH89] P. Van Hentenryck. Constraint Satisfaction in Logic Programming. MIT Press, Boston, MA, 1989.

[VH98] P. Van Hentenryck. The OPL Optimization Programming Language. MIT Press, Cambridge MA, 1998.

[VSD92] P. Van Hentenryck, H. Simonis, M. Dincbas. Constraint Satisfaction using Constraint Logic Programming. *Journal of Artificial Intelligence*, Vol.58, No.1-3, pp.113-161, USA, 1992.

[VC88] P. Van Hentenryck, J-P. Carillon. Generality versus Specificity: an Experience with AI and OR Techniques. In *Proceedings American Association for Artificial Intelligence (AAAI-88)*, St. Paul, MI, August 1988.

[Wal96] M. Wallace. Practical Applications of Constraint Programming. *Constraints Journal*, Vol 1, Nr1, 2, Sept 1996, pp 139-168.

[WNS97] M. Wallace, S. Novello, and J. Schimpf. ECLiPSe - a platform for constraint programming. *ICL Systems Journal*, 12(1):159–200, 1997.