Lecture Notes in Computer Science 5438

*Commenced Publication in 1973*
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Michael Hanus (Ed.)

# Logic-Based Program Synthesis and Transformation

18th International Symposium, LOPSTR 2008
Valencia, Spain, July 17-18, 2008
Revised Selected Papers

Springer

Volume Editor

Michael Hanus
Christian-Albrechts-Universität Kiel
Institut für Informatik
24098 Kiel, Germany
E-mail: mh@informatik.uni-kiel.de

Library of Congress Control Number: 2009921732

CR Subject Classification (1998): D.2, D.1.6, D.1.1, F.3.1, I.2.2, F.4.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-642-00514-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-00514-5 Springer Berlin Heidelberg New York



Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12620142 06/3180 5 4 3 2 1 0

# Preface

This volume contains a selection of the papers presented at the 18th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2008) held during July 17–18, 2008 in Valencia, Spain. Information about the conference can be found at `http://www.informatik.uni-kiel.de/~mh/lopstr08`. Previous LOPSTR symposia were held in Lyngby (2007), Venice (2006 and 1999), London (2005 and 2000), Verona (2004), Uppsala (2003), Madrid (2002), Paphos (2001), Manchester (1998, 1992, and 1991), Leuven (1997), Stockholm (1996), Arnhem (1995), Pisa (1994), and Louvain-la-Neuve (1993).

The aim of the LOPSTR series is to stimulate and promote international research and collaboration on logic-based program development. LOPSTR traditionally solicits papers in the areas of specification, synthesis, verification, transformation, analysis, optimization, composition, security, reuse, applications and tools, component-based software development, software architectures, agent-based software development, and program refinement. LOPSTR has a reputation for being a lively, friendly forum for presenting and discussing work in progress. Formal proceedings are produced only after the symposium so that authors can incorporate feedback in the published papers.

I would like to thank all those who submitted contributions to LOPSTR in the categories of full papers and extended abstracts. Each submission was reviewed by at least three Program Committee members. The committee decided to accept three full papers for immediate inclusion in the final post-conference proceedings, and nine papers were accepted after revision and another round of reviewing. In addition to the accepted papers, the program also included an invited talk by Peter O'Hearn (University of London).

I am grateful to the Program Committee members who worked hard to produce high-quality reviews for the submitted papers under a tight schedule, as well as all the external reviewers involved in the paper selection. I also would like to thank Andrei Voronkov for his excellent EasyChair system that automates many of the tasks involved in chairing a conference.

LOPSTR 2008 was co-located with SAS 2008, PPDP 2008, and PLID 2008. Many thanks to the local organizers of these events, in particular, to Josep Silva, the LOPSTR 2008 local Organizing Committee Chair. Finally, I gratefully acknowledge the institutions that sponsored this event: Departamento de Sistemas Informáticos y Computación, EAPLS, ERCIM, Generalitat Valenciana, MEC (Feder) TIN2007-30509-E, and Universidad Politécnica de Valencia.

December 2008 Michael Hanus

# Conference Organization

## Program Chair

Michael Hanus
Institut für Informatik
Christian-Albrechts-Universität Kiel
D-24098 Kiel, Germany
Email: mh@informatik.uni-kiel.de

## Local Organization Chair

Josep Silva
Departamento de Sistemas Inform. y Comp.
Universitat Politècnica de Valencia Camino de la Vera s/n
E-46022 Valencia, Spain
Email: jsilva@dsic.upv.es

## Program Committee

| | |
|---|---|
| Slim Abdennadher | German University Cairo, Egypt |
| Danny De Schreye | K.U. Leuven, Belgium |
| Wlodek Drabent | Polish Academy of Sciences, Poland / Linköping University, Sweden |
| Gopal Gupta | University of Texas at Dallas, USA |
| Michael Hanus | University of Kiel, Germany (Chair) |
| Patricia Hill | University of Leeds, UK |
| Andy King | University of Kent, UK |
| Michael Leuschel | University of Düsseldorf, Germany |
| Torben Mogensen | DIKU, University of Copenhagen, Denmark |
| Mario Ornaghi | Università degli Studi di Milano, Italy |
| Étienne Payet | Université de La Réunion, France |
| Alberto Pettorossi | University of Rome Tor Vergata, Italy |
| Germán Puebla | Technical University of Madrid, Spain |
| C.R. Ramakrishnan | SUNY at Stony Brook, USA |
| Sabina Rossi | Università Ca' Foscari di Venezia, Italy |
| Chiaki Sakama | Wakayama University, Japan |
| Josep Silva | Technical University of Valencia, Spain |
| Wim Vanhoof | University of Namur, Belgium |
| Eelco Visser | Delft University of Technology, The Netherlands |

## Organizing Committee

Beatriz Alarcón
Antonio Bella
Santiago Escobar
Marco Feliu
Ana Funes
Salvador Lucas
José Hernández
Christophe Joubert
Marisa Llorens
Pedro Ojeda
María José Ramírez
Josep Silva (Chair)
Alicia Villanueva
Gustavo Arroyo
Aristides Dasso
Vicent Estruch
César Ferri
Carlos Herrero
Raúl Gutiérrez
José Iborra
Alexei Lescaylle
Rafael Navarro
Javier Oliver
Daniel Romero
Salvador Tamarit

## External Reviewers

Javier Álvez
François Degrave
Miguel Gómez-Zamalloa
Gerda Janssens
Adam Koprowski
Matteo Maffei
Alberto Momigliano
Pawel Pietrzak
Maurizio Proietti
Son Tran
Dean Voets
Rafael Caballero
Samir Genaim
Roberta Gori
Christophe Joubert
Kenneth MacKenzie
Isabella Mastroeni
Frank Pfenning
Paolo Pilozzi
Jan-Georg Smaus
Germán Vidal
Damiano Zanardini

# Table of Contents

# Space Invading Systems Code

Cristiano Calcagno[2], Dino Distefano[1], Peter O'Hearn[1], and Hongseok Yang[1]

[1] Queen Mary University of London
[2] Imperial College

## 1 Introduction

Space Invader is a static analysis tool that aims to perform accurate, automatic verification of the way that programs use pointers. It uses separation logic assertions [10,11] to describe states, and works by performing a proof search, using abstract interpretation to enable convergence. As well as having roots in separation logic, Invader draws on the fundamental work of Sagiv et. al. on shape analysis [12]. It is complementary to other tools – e.g., SLAM [1], Blast [8], ASTRÉE [6] – that use abstract interpretation for verification, but that use coarse or limited models of the heap.

Space Invader began life as a theoretical prototype working on a toy language [7], which was itself an outgrowth of a previous toy-language tool [3]. Then, in May of 2006, spurred by discussions with Byron Cook, we decided to move beyond our toy languages and challenge programs, and test our ideas against real-world systems code, starting with a Windows device driver, and then moving on to various open-source programs. (Some of our work has been done jointly with Josh Berdine and Cook at Microsoft Research Cambridge, and a related analysis tool, SLAyer, is in development there.)

As of the summer of 2008, Space Invader has proven pointer safety (no null or dangling pointer dereferences, or leaks) in several entire industrial programs of up to 10K LOC, and more partial properties of larger codes. There have been three key innovations driven by the problems encountered with real-world code.

- *Adaptive analysis.* Device drivers use complex variations on linked lists – for example, multiple circular lists sharing a common header, several of which have nested sublists – and these variations are different in different drivers. In the adaptive analysis predicates are discovered by scrutinizing the linking structure of the heap, and then fed to a higher-order predicate that describes linked lists. This allows for the description of complex, nested (though linear) data structures, as well as for adapting to the varied data structures found in different programs [2].
- *Near-perfect Join.* The adaptive analysis allowed several driver routines to be verified, but it timed out on others. The limit was around 1K LOC, when given a nested data structure and a procedure with non-trivial control flow (several loops and conditionals). The problem was that there were thousands of nodes at some program points in the analysis, representing huge disjunctions. In response, we discovered a partial join operator which lost enough

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 1–3, 2009.

information to, in many cases (though crucially, not always), leave us with only one heap. The join operator is *partial* because, although it is often defined, a join which *always* collapses two nodes into one will be too imprecise to verify the drivers: it will have false alarms. Our goal was to *prove* pointer safety of the drivers, so to discharge even 99.9% of the heap dereference sites was considered a failure: *not* to have found a proof.

The mere idea of a join is of course standard: The real contribution is existence of a partial join operator that leads to speed-ups which allow entire drivers to be analyzed, while retaining enough precision for the goal of proving pointer safety with zero false alarms [9].

– *Compositionality.* The version of Space Invader with adaptation and join was a top-down, whole-program analysis (like all previous heap verification methods). This meant the user had to either supply preconditions manually, or provide a "fake main program" (i.e., supply an environment). Practically, the consequence was that it was time-consuming to even get started to apply the analysis to a new piece of code, or to large codes. We discovered a method of inferring a preconditon and postcondition for a procedure, without knowing its calling context: the method aims to find the "footprint" of the code [4], a description of the cells it accesses. The technique – which involves the use of abductive inference to infer assertions describing missing portions of heap – leads to a compositional analysis which has been applied to larger programs, such as a complete linux distribution of 2.5M LOC [5].

The compositional and adaptive verification techniques fit together particularly well. If you want to *automatically* find a spec of the data structure usage in a procedure in some program you don't know, without having the calling context of the procedure, you really need an analysis method that will find heap predicates for you, without requiring you (the human) to supply those predicates on a case-by-case basis. Of course, the adaptive analysis selects its predicates from some pre-determined stock, and is ultimately limited by that, but the adaptive capability is handy to have, nonetheless.

We emphasize that the results of the compositional version of Space Invader (code name: Abductor) are partial: it is able to prove some procedures, but it might fail to prove others; in linux it finds Hoare triples for around 60,000 procedures, while leaving unproven some 40,000 others[1]. This, though, is one of the benefits of compositional methods. It is possible to get accurate results on parts of a large codebase, without waiting for the "magical abstract domain" that can automatically prove all of the procedures in all of the code we would want to consider.

[1] Warning: there are caveats concerning Abductor's limitations, such as how it ignores concurrency. These are detailed in [5].

## References


1. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys. (2006)
2. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis of composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
3. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Automatic modular assertion checking with separation logic. In: 4th FMCO, pp. 115–137 (2006)
4. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Footprint analysis: A shape analysis that discovers preconditions. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 402–418. Springer, Heidelberg (2007)
5. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
6. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005)
7. Distefano, D., O'Hearn, P., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL: Principles of Programming Languages (2002)
9. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
10. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)
11. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
12. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM TOPLAS 20(1), 1–50 (1998)

# Test Data Generation of Bytecode by CLP Partial Evaluation

Elvira Albert[1], Miguel Gómez-Zamalloa[1], and Germán Puebla[2]

[1] DSIC, Complutense University of Madrid, E-28040 Madrid, Spain
[2] CLIP, Technical University of Madrid, E-28660 Boadilla del Monte, Madrid, Spain

**Abstract.** We employ existing *partial evaluation* (PE) techniques developed for Constraint Logic Programming (CLP) in order to automatically generate *test-case generators* for glass-box testing of bytecode. Our approach consists of two independent CLP PE phases. (1) First, the bytecode is transformed into an equivalent (decompiled) CLP program. This is already a well studied transformation which can be done either by using an ad-hoc decompiler or by specialising a bytecode interpreter by means of existing PE techniques. (2) A second PE is performed in order to supervise the generation of test-cases by execution of the CLP decompiled program. Interestingly, we employ control strategies previously defined in the context of CLP PE in order to capture *coverage criteria* for glass-box testing of bytecode. A unique feature of our approach is that, this second PE phase allows generating not only test-cases but also test-case *generators*. To the best of our knowledge, this is the first time that (CLP) PE techniques are applied for test-case generation as well as to generate test-case generators.

## 1 Introduction

Bytecode (e.g., Java bytecode [19] or .Net) is becoming widely used, especially, in the context of mobile applications for which the source code is not available and, hence, there is a need to develop verification and validation tools which work directly on bytecode programs. Reasoning about complex bytecode programs is rather difficult and time consuming. In addition to object-oriented features such as objects, virtual method invocation, etc., bytecode has several low-level language features: it has an unstructured control flow with several sources of branching (e.g., conditional and unconditional jumps) and uses an operand stack to perform intermediate computations.

Test data generation (TDG) aims at automatically generating test-cases for interesting test *coverage criteria*. The coverage criteria measure how well the program is exercised by a test suite. Examples of coverage criteria are: *statement coverage* which requires that each line of the code is executed; *path coverage* which requires that every possible trace through a given part of the code is executed; etc. There are a wide variety of approaches to TDG (see [27] for a survey).Our work focuses on *glass-box* testing, where test-cases are obtained from the concrete program in contrast to *black-box* testing, where they are deduced from a specification of the program. Also, our focus is on *static* testing,

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 4–23, 2009.

where we assume no knowledge about the input data, in contrast to *dynamic* approaches [9,14] which execute the program to be tested for concrete input values.

The standard approach to generating test-cases statically is to perform a *symbolic* execution of the program [7,22,23,17,13], where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of *constraints* consisting of the conditions to execute the different paths. This happens, for instance, in branching instructions, like if-then-else, where we might want to generate test-cases for the two alternative branches and hence accumulate the conditions for each path as constraints. The symbolic execution approach has been combined with the use of *constraint solvers* [23,13] in order to: handle the constraints systems by solving the feasibility of paths and, afterwards, to instantiate the input variables. For the particular case of Java bytecode, a symbolic JVM machine (SJVM) which integrates several constraints solvers has been designed in [23]. A SJVM requires non-trivial extensions w.r.t. a JVM: (1) it needs to execute the bytecode symbolically as explained above, (2) it must be able to backtrack, as without knowledge about the input data, the execution engine might need to execute more than one path. The backtracking mechanism used in [23] is essentially the same as in logic programming.

We propose a novel approach to TDG of bytecode which is based on PE techniques developed for CLP and which, in contrast to previous work, does not require the devising a dedicated symbolic virtual machine. Our method comprises two CLP PE phases which are independent. In fact, they rely on different execution and control strategies:

1. *The decompilation of bytecode into a CLP program.* This has been the subject of previous work [15,3,12] and can be achieved automatically by relying on the first Futamura projection by means of partial evaluation for logic programs, or alternatively by means of an adhoc decompiler [21].
2. *The generation of test-cases.* This is a novel application of PE which allows generating test-case generators from CLP decompiled bytecode. In this case, we rely on a CLP partial evaluator which is able to solve the constraint system, in much the same way as a symbolic abstract machine would do. The two control operators of a CLP partial evaluator play an essential role: (1) The local control applied to the decompiled code will allow capturing interesting coverage criteria for TDG of the bytecode. (2) The global control will enable the generation of *test-case generators*. Intuitively, the TDG generators we produce are CLP programs whose execution in CLP returns further test-cases on demand without the need to start the TDG process from scratch.

We argue that our CLP PE based approach to TDG of bytecode has several advantages w.r.t. existing approaches based on symbolic execution: (i) It is more *generic*, as the same techniques can be applied to other both low and high-level imperative languages. In particular, once the CLP decompilation is done, the language features are abstracted away and, the whole part related to TDG generation is totally *language independent*. This avoids the difficulties of dealing

with recursion, procedure calls, dynamic memory, etc. that symbolic abstract machines typically face. (ii) It is more *flexible*, as different coverage criteria can be easily incorporated to our framework just by adding the appropriate local control to the partial evaluator. (iii) It is more *powerful* as we can generate test-case generators. (iv) It is *simpler* to implement compared to the development of a dedicated SJVM, as long as a CLP partial evaluator is available.

The rest of the paper is organized as follows. The next section recalls some preliminary notions. Sec. 3 describes the notion of CLP *block-level* decompilation which corresponds to the first phase above. The second phase is explained in the remainder of the paper. Sec. 4 presents a naïve approach to TDG using CLP decompiled programs. In Sec. 5, we introduce the block count-k coverage criterion and outline an evaluation strategy for it. In Sec. 6, we present our approach to TDG by partial evaluation of CLP. Sec. 7 discusses related work and concludes.

## 2 Preliminaries and Notation in Constraint Logic Programs

We now introduce some basic notions about *Constraint Logic Programming* (CLP). See e.g. [20] for more details. A *constraint store*, or *store* for short, is a conjunction of expressions built from predefined predicates (such as term equations and equalities or inequalities over the integers) whose arguments are constructed using predefined functions (such as addition, multiplication, etc.). We let $\overline{\exists}_L\theta$ be the constraint store $\theta$ restricted to the variables of the syntactic object $L$. An *atom* has the form $p(t_1, ..., t_n)$ where $p$ is a predicate symbol and the $t_i$ are terms. A *literal* $L$ is either an atom or a constraint. A *goal* $L_1, \ldots, L_n$ is a possibly empty finite conjunction of literals. A *rule* is of the form $H$ :- $B$ where $H$, the *head*, is an atom and $B$, the *body*, is a goal. A *constraint logic program*, or *program*, is a finite set of rules. We use *mgu* to denote a most general unifier for two unifiable terms.

The operational semantics of a program $P$ is in terms of its *derivations* which are sequences of reductions between *states*. A *state* $\langle G \mid \theta \rangle$ consists of a goal $G$ and a constraint store $\theta$. A state $\langle L, G \mid \theta \rangle$ where $L$ is a literal can be *reduced* as follows:

1. If $L$ is a constraint and $\theta \wedge L$ is satisfiable, it is reduced to $\langle G \mid \theta \wedge L \rangle$.
2. If $L$ is an atom, it is reduced to $\langle B, G \mid \theta \wedge \theta' \rangle$ for some renamed apart rule ($L'$ :- $B$) in $P$ such that $L$ and $L'$ unify with mgu $\theta'$.

A *derivation* from state $S$ for program $P$ is a sequence of states $S_0 \rightarrow_P S_1 \rightarrow_P ... \rightarrow_P S_n$ where $S_0$ is $S$ and there is a reduction from each $S_i$ to $S_{i+1}$. Given a non-empty derivation $D$, we denote by *curr_state(D)* and *curr_store*($D$) the last state in the derivation, and the store in this last state, respectively. E.g., if $D$ is the derivation $S_0 \rightarrow_P^* S_n$, where $\rightarrow^*$ denotes a sequence of steps, with $S_n = \langle G \mid \theta \rangle$ then $curr\_state(D) = S_n$ and $curr\_store(D) = \theta$. A query is a pair $(L, \theta)$ where $L$ is a literal and $\theta$ a store for which the CLP system starts a computation from $\langle L \mid \theta \rangle$.

The observational behavior of a program is given by its "answers" to queries. A finite derivation $D$ from a query $Q = (L, \theta)$ for program $P$ is *finished* if $curr\_state(D)$ cannot be reduced. A finished derivation $D$ from a query $Q = (L, \theta)$ is *successful* if $curr\_state(D) = \langle \epsilon \parallel \theta' \rangle$, where $\epsilon$ denotes the empty conjunction. The constraint $\exists_L \theta'$ is an *answer* to $Q$. A finished derivation is *failed* if the last state is not of the form $\langle \epsilon \parallel \theta \rangle$. Since evaluation trees may be infinite, we allow *unfinished* derivations, where we decide not to further perform reductions. Derivations can be organized in execution trees: a state $S$ has several children when its leftmost atom unifies with several program clauses.

## 3 Decompilation of Bytecode to CLP

Let us first briefly describe the bytecode language we consider. It is a very simple imperative low-level language in the spirit of Java bytecode, without object-oriented features and restricted to manipulate only integer numbers. It uses an operand stack to perform computations and has an unstructured control flow with explicit conditional and unconditional `goto` instructions. A bytecode program is organized in a set of methods which are the basic (de)compilation units of the bytecode. The code of a method $m$ consists of a sequence of bytecode instructions $BC_m = < pc_0 : bc_0, \ldots, pc_{n_m} : bc_{n_m} >$ with $pc_0, \ldots, pc_{n_m}$ being consecutive natural numbers. The instruction set is:

$$\begin{aligned} BcInst ::= {} & \mathsf{push(x)} \mid \mathsf{load(v)} \mid \mathsf{store(v)} \mid \mathsf{add} \mid \mathsf{sub} \mid \mathsf{mul} \mid \mathsf{div} \mid \mathsf{rem} \mid \mathsf{neg} \mid \\ & \mathsf{if} \diamond \mathsf{(pc)} \mid \mathsf{if0} \diamond \mathsf{(pc)} \mid \mathsf{goto(pc)} \mid \mathsf{return} \mid \mathsf{call(mn)} \end{aligned}$$

where $\diamond$ is a comparison operator (eq, le, gt, etc.), $v$ a local variable, $x$ an integer, $pc$ an instruction index and $mn$ a method name. The instructions push, load and store transfer values or constants from a local variable to the stack (and vice versa); add, sub, mul, div, rem and neg perform arithmetic operations, rem is the division remainder and neg the negation; if and if0 are conditional branching instructions (with the special case of comparisons with 0); goto is an unconditional branching; return marks the end of methods returning an integer and call invokes a method.

Figure 1 depicts the control flow graphs (CFGs) [1] and, within them, the bytecode instructions associated to the methods `lcm` (on the left), `gcd` (on the right) and `abs` (at the bottom). A Java-like source code for them is shown to the left of the figure. It is important to note that we show source code only for clarity, as our approach works directly on the bytecode. The use of the operand stack can be observed in the example: the bytecode instructions at *pc* `0` and `1` in `lcm` load the values of parameters `x` and `y` (resp.) to the stack before invoking the method `gcd`. Method parameters and local variables in the program are referenced by consecutive natural numbers starting from `0` in the bytecode. The result of executing the method `gcd` has been stored on the top of the stack. At *pc* `3`, this value is popped and assigned to variable `2` (called `gcd` in the Java program). The branching at the end of $\texttt{Block}_1$ is due to the fact that the division bytecode instruction `div` can throw an exception if the divisor is zero (control

```
int lcm(int x,int y){
  int gcd = gcd(x,y);
  return abs(x*y/gcd);
}

int gcd(int x,int y){
  int res;
  while (y != 0){
    res = x%y;
    x = y; y = res;
  }
  return abs(x);
}

int abs(int x){
  if (x >= 0)
    return x;
  else return -x;
}
```

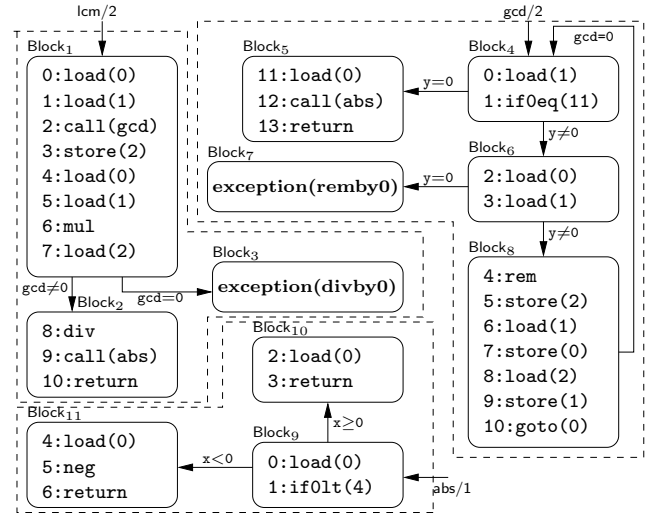



**Fig. 1.** Working example. Source code and CFGs for the bytecode.

goes to $\texttt{Block}_3$). In the bytecode for `gcd`, we find: conditional jumps, like `if0eq` at *pc* 1, which corresponds to the loop guard, and unconditional jumps, like `goto` in *pc* `10`, where the control returns to the loop entry. Note that the bytecode instruction `rem` can throw an exception as before.

### 3.1 Decompilation by PE and Block-Level Decompilation

The decompilation of low-level code to CLP has been the subject of previous research, see [15,3,21] and their references. In principle, it can be done by defining an adhoc decompiler (like [2,21]) or by relying on the technique of PE (like [15,3]). The decompilation of low-level code to CLP by means of PE consists in specializing a bytecode interpreter implemented in CLP together with (a CLP representation of) a bytecode program. As the first Futamura projection [11] predicts, we obtain a CLP residual program which can be seen as a decompiled and translated version of the bytecode into high-level CLP source. The approach to TDG that will be presented in the remaining of this paper is independent of the technique used to generate the CLP decompilation. Thus, we will not explain the decompilation process (see [15,3,21]) but rather only state the decompilation requirements our method imposes.

The *correctness* of decompilation must ensure that there is a one to one correspondence between execution paths in the bytecode and derivations in the CLP decompiled program. In principle, depending on the particular type of decompilation –and even on the options used within a particular method– we can obtain different correct decompilations which are valid for the purpose of execution. However, for the purpose of generating useful test-cases, additional requirements

```
lcm([X,Y],Z) :- gcd([X,Y],GCD),P #= X*Y,
                lcm1c([GCD,P],Z).
lcm1c([GCD,P],Z) :- GCD #\= 0,D #= P/GCD,
                    abs([D],Z).
lcm1c([0,_],divby0).

abs([X],Z) :- abs9c(X,Z).
abs9c(X,X) :- X #>= 0.
abs9c(X,Z) :- X #< 0, Z #= -X.

gcd([X,Y],Z) :- gcd4(X,Y,Z).

gcd4(X,Y,Z) :- gcd4c(X,Y,Z).

gcd4c(X,0,Z) :- abs([X],Z).
gcd4c(X,Y,Z) :- Y #\= 0,
                gcd6c(X,Y,Z).

gcd6c(X,Y,Z) :- Y #\= 0,
                R #= X mod Y,
                 gcd4(Y,R,Z).
gcd6c(_,0,remby0).
```

**Fig. 2.** Block-level decompilation to CLP for working example

are needed: we must be able to define coverage criteria on the CLP decompilation which produce test-cases which cover the *equivalent* coverage criteria for the bytecode. The following notion of *block-level* decompilation, introduced in [12], provides a sufficient condition for ensuring that equivalent coverage criteria can be defined.

**Definition 1 (block-level decompilation).** *Given a bytecode program BC and its CLP-decompilation P, a block-level decompilation ensures that, for each block in the CFGs of BC, there exists a* single *corresponding rule in P which contains all bytecode instructions within the block.*

The above notion was introduced in [12] to ensure optimality in decompilation, in the sense that each program point in the bytecode is traversed, and decompiled code is generated for it, at most once. According to the above definition there is a one to one correspondence between blocks in the CFG and rules in $P$, as the following example illustrates. The block-level requirement is usually an implicit feature of adhoc decompilers (e.g., [2,21]) and can be also enforced in decompilation by PE (e.g., [12]).

*Example 1.* Figure 2 shows the code of the block-level decompilation to CLP of our running example which has been obtained using the decompiler in [12] and uses CLP(FD) built-in operations (in particular those in the `clpfd` library of `Sicstus` Prolog). The input parameters to methods are passed in a list (first argument) and the second argument is the output value. We can observe that each block in the CFG of the bytecode of Fig. 1 is represented by a corresponding clause in the above CLP program. For instance, the rules for `lcm` and `lcm1c` correspond to the three blocks in the CFG for method `lcm`. The more interesting case is for method `gcd`, where the `while` loop has been converted into a cycle in the decompiled program formed by the predicates `gcd4`, `gcd4c`, and `gcd6c`. In this case, since `gcd4` is the head of a loop, there is one more rule (`gcd`) than blocks in the CFG. This additional rule corresponds to the method *entry*. Bytecode instructions are decompiled and translated to their corresponding operations in CLP; conditional statements are captured by the continuation rules.

For instance, in `gcd4`, the bytecode instruction at *pc* 0 is executed to unify a stack position with the local variable `y`. The conditional `if0eq` at *pc* 1 leads to two continuations, i.e. two rules for predicate `gcd4c`: one for the case when `y=0` and another one for `y`$\neq$`0`. Note that we have explicit rules to capture the exceptional executions (which will allow generating test-cases which correspond to exceptional executions). Note also that in the decompiled program there is no difference between calls to blocks and method calls. E.g., the first rule for `lcm` includes in its body a method call to `gcd` and a block call `lcm1c`.

## 4 Test Data Generation Using CLP Decompiled Programs

Up to now, the main motivation for CLP decompilation has been to be able to perform static analysis on a decompiled program in order to infer properties about the original bytecode. If the decompilation approach produces CLP programs which are executable, then such decompiled programs can be used not only for static analysis, but also for dynamic analysis and execution. Note that this is not always the case, since there are approaches (like [2,21]) which are aimed at producing static analysis targets only and their decompiled programs cannot be executed.

### 4.1 Symbolic Execution for Glass-Box Testing

A novel interesting application of CLP decompilation which we propose in this work is the automatic generation of glass-box test data. We will aim at generating test-cases which traverse as many different execution paths as possible. From this perspective, different test data should correspond to different execution paths. With this aim, rather than executing the program starting from different input values, a well-known approach consists in performing *symbolic execution* such that a single symbolic run captures the behaviour of (infinitely) many input values. The central idea in symbolic execution is to use constraint variables instead of actual input values and to capture the effects of computation using constraints (see Sec. 1).

Several symbolic execution engines exist for languages such as Java [4] and Java bytecode [23,22]. An important advantage of CLP decompiled programs w.r.t. their bytecode counterparts is that symbolic execution does not require, at least in principle, to build a dedicated symbolic execution mechanism. Instead, we can simply run the decompiled program by using the standard CLP execution mechanism with all arguments being distinct free variables. E.g., in our case we can execute the query `lcm`(`[X,Y]`, `Z`). By running the program without input values on a block level decompiled program, each successful execution corresponds to a different computation path in the bytecode. Furthermore, along the execution, a constraint store on the program's variables is obtained which

can be used for inferring the conditions that the input values (in our case `X` and `Y`) must satisfy for the execution to follow the corresponding computation path.

### 4.2 From Constraint Stores to Test Data

An inherent assumption in the symbolic execution approach, regardless of whether a dedicated symbolic execution engine is built or the default CLP execution is used, is that all valuations of constraint variables which satisfy the constraints in the store (if any) result in input data whose computation traverses the same execution path. Therefore, it is irrelevant, from the point of view of the execution path, which actual values are chosen as representatives of a given store. In any case, it is often required to find a valuation which satisfies the store. Note that this is a strict requirement if we plan to use the bytecode program for testing, though it is not strictly required if we plan to use the decompiled program for testing, since we could save the final store and directly use it as input test data. Then, execution for the test data should load the store first and then proceed with execution. In what follows, we will concentrate on the first alternative, i.e., we generate actual values as test data.

This postprocessing phase is straightforward to implement if we use CLP(FD) as the underlying constraint domain, since it is possible to enumerate values for variables until a solution which is consistent with the set of constraints is found (i.e., we perform *labeling*). Note, however, that it may happen that some of the computed stores are indeed inconsistent and that we cannot find any valuation of the constraint variables which simultaneously satisfies all constraints in the store. This may happen for unfeasible paths, i.e., those which do not correspond to any actual execution. Given a decompiled method `M`, an integer subdomain `[RMin,RMax]`, the predicate `generate_test_data/4` below produces, on backtracking, a (possibly infinite) set of values for the variables in `Args` and the result value in `Z`.

```
generate_test_data(M,Args,[RMin,RMax],Z) :-
    domain(Args,RMin,RMax), Goal =..[M,Args,Z],
    call(Goal), once(labeling([ff],Args)).
```

Note that the generator first imposes an integer domain for the program variables by means of the call to `domain/3`; then builds the `Goal` and executes it by means of `call(Goal)` to generate the constraints; and finally invokes the enumeration predicate `labeling/2` to produce actual values compatible with the constraints[1]. The test data obtained are in principle specific to some integer subdomain; indeed our bytecode language only handles integers. This is not necessarily a limitation, as the subdomain can be adjusted to the underlying bytecode machine limitations, e.g., $[-2^{31}, 2^{31} - 1]$ in the Java virtual machine. Note that if the variables take floating point values, then other constraint domains such as CLP(R) or CLP(Q) should be used and then, other mechanisms for generating actual values should be used.

[1] We are using the `clpfd` library of `Sicstus` Prolog. See [26] for details on predicates `domain/3`, `labeling/2`, etc.

## 5 An Evaluation Strategy for *Block-Count*($k$) Coverage

As we have seen in the previous section, an advantage of using CLP decompiled programs for test data generation is that there is no need to build a symbolic execution engine. However, an important problem with symbolic execution, regardless of whether it is performed using CLP or a dedicated execution engine, is that the execution tree to be traversed is in most cases infinite, since programs usually contain iterative constructs such as loops and recursion which induce an infinite number of execution paths when executed without input values.

*Example 2.* Consider the evaluation of the call `lcm([X,Y],Z)`, depicted in Fig. 3. There is an infinite derivation (see the rightmost derivation in the tree) where the cycle {`gcd4,gcd4c,gcd6c`} is traversed forever. This happens because the value in the second argument position of `gcd4c` is not ground during symbolic computation.

Therefore, it is essential to establish a *termination criterion* which guarantees that the number of paths traversed remains finite, while at the same time an interesting set of test data is generated.

### 5.1 *Block-count*($k$): A Coverage Criteria for Bytecode

In order to reason about how interesting a set of test data is, a large series of *coverage criteria* have been developed over the years which aim at guaranteeing that the program is exercised on interesting control and/or data flows. In this section we present a coverage criterion of interest to bytecode programs. Most existing coverage criteria are defined on high-level, structured programming languages. A widely used control-flow based coverage criterion is loop-count($k$), which dates back to 1977 [16], and limits the number of times we iterate on loops to a threshold $k$. However, bytecode has an unstructured control flow: CFGs can contain multiple different shapes, some of which do not correspond to any of the loops available in high-level, structured programming languages. Therefore, we introduce the block-count($k$) coverage criterion which is not explicitly based on limiting the number of times we iterate on loops, but rather on counting how many times we visit each block in the CFG within each computation. Note that the execution of each method call is considered as an independent computation.

**Definition 2 (block-count($k$)).** *Given a natural number $k$, a set of computation paths satisfies the* block-count($k$) criterion *if the set includes all finished computation paths which can be built such that the number of times each block is visited within each computation does not exceed the given $k$.*

Therefore, if we take $k = 1$, this criterion requires that all non-cyclic paths be covered. Note that $k = 1$ will in general not visit all blocks in the CFG, since traversing the loop body of a `while` loop requires $k \geq 2$ in order to obtain a finished path. For the case of structured CFGs, block-count($k$) is actually equivalent to loop-count($k'$), by simply taking $k'$ to be $k$-1. We prefer to formulate things in terms of block-count($k$) since, formulating loop-count($k$) on unstructured CFGs is awkward.

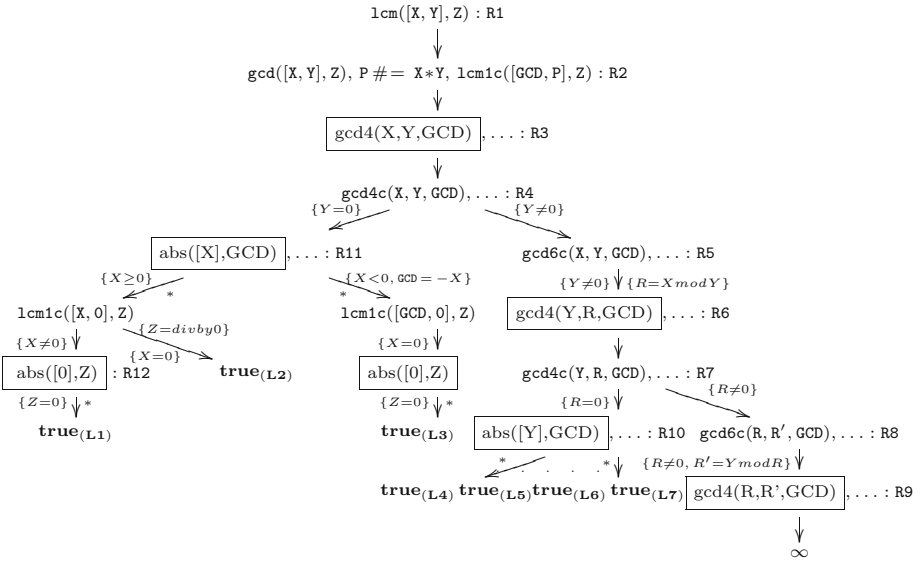



**Fig. 3.** An evaluation tree for `lcm([X,Y],Z)`

### 5.2 An Intra-procedural Evaluation Strategy for Block-Count($k$)

Fig. 3 depicts (part of) an evaluation tree for `lcm([X,Y],Z)`. Each node in the tree represents a state, which as introduced in Sec. 2, consists of a goal and a store. In order not to clutter the figure, for each state we only show the relevant part of the goal, but not the store. Also, an arc in the tree may involve several reduction steps. In particular, the constraints which precede the leftmost atom (if any) are always processed. Likewise, at least one reduction step is performed on the leftmost atom w.r.t. the program rule whose head unifies with the atom. When more than one step is performed, the arc is labelled with "$*$". Arcs are annotated with the constraints processed at each step. Each branch in the tree represents a *derivation*.

Our aim is to supervise the generation of the evaluation tree so that we generate sufficiently many derivations so as to satisfy the block-count($k$) criterion while, at the same time, guaranteeing termination.

**Definition 3 (intra-procedural evaluation strategy).** *The following two conditions provide an evaluation strategy which ensures block-count($k$) in intra-procedural bytecode (i.e., we consider a single CFG for one method):*

*(i) annotate every state in the evaluation tree with a multiset, which we refer to as* visited*, and which contains the predicates which have been already reduced during the derivation;*

*(ii) atoms can only be reduced if there are at most $k-1$ occurrences of the corresponding predicate in* visited*.*

It is easy to see that this evaluation strategy is guaranteed to always produce a finite evaluation tree since there is a finite number of rules which can unify with any given atom and therefore non-termination can only be introduced by cycles which are traversed an unbounded number of times. This is clearly avoided by limiting the number of times which resolution can be performed w.r.t. the same predicate.

*Example 3.* Let us consider the rightmost derivation in Fig. 3, formed by goals R1 to R9. Observe the framed atoms for `gcd4`, the goals R3, R6 and R9 contain an atom for `gcd4` as the leftmost literal. If we take $k = 1$ then resolvent R6 cannot be further reduced since the termination criterion forbids it, as `gcd4` is already once in the multiset of visited predicates. If we take $k = 2$ then R6 can be reduced and the termination criterion is fired at R9, which cannot be further reduced.

### 5.3 An Inter-procedural Evaluation Strategy Based on Ancestors

The strategy of limiting the number of reductions w.r.t. the same predicate guarantees termination. Furthermore, it also guarantees that the block-count($k$) criterion is achieved, but only if the program consists of a single CFG, i.e., at most one method. If the program contains more than one method, as in our example, this evaluation strategy may force termination too early, without achieving block-count($k$) coverage.

*Example 4.* Consider the predicate `abs`. Any successful derivation which does not correspond to exceptions in the bytecode program has to execute this predicate twice, once from the body of method `lcm` and another one from the body of method `gcd`. Therefore, if we take $k = 1$, the leftmost derivation of the tree in Fig. 3 will be stopped at R12, since the atom to be reduced is considered to be a repeated call to predicate `abs`. Thus, the test-case for the successful derivation L1 is not obtained. As a result, our evaluation strategy would not achieve the block-count($k$) criterion.

The underlying problem is that we are in an inter-procedural setting, i.e., bytecode programs contain method calls. In this case –meanwhile decompiled versions of bytecode programs without method calls always consist of binary rules– decompiled programs may have rules with several atoms in their body. This is indeed the case for the rule for `lcm` in Ex. 1, which contains an atom for predicate `gcd` and another one for predicate `lcm1c`. Since under the standard left-to-right computation rule, the execution of `gcd` is finished by the time execution reaches `lcm1c` there is no need to take the computation history of `gcd` into account when supervising the execution of `lcm1c`. In our example, the execution of `gcd` often involves an execution of `abs` which is finished by the time the call to `abs` is performed within the execution of `lcm1c`. This phenomenon is well known problem in the context of partial evaluation. There, the notion of *ancestor* has been introduced [5] to allow supervising the execution of conjuncts independently by

only considering visited predicates which are actually ancestors of the current goal. This allows improving accuracy in the specialization.

Given a reduction step where the leftmost atom $A$ is substituted by $B_1, \ldots, B_m$, we say that $A$ is the *parent* of the instance of $B_i$ for $i = 1, \ldots, m$ in the new goal and in each subsequent goal where the instance originating from $B_i$ appears. The *ancestor* relation is the transitive closure of the parent relation. The multiset of ancestors of the atom for `abs` in goal R12 in the SLD tree is {`lcm1c,lcm`}, as `lcm1c` is its parent and `lcm` the parent of its parent. Importantly, `abs` is not in such multiset. Therefore, the leftmost computation in Fig. 3 will proceed upon R12 thus producing the corresponding test-case for every $k \geq 1$. The evaluation strategy proposed below relies on the notion of ancestor sequence.

**Definition 4 (inter-procedural evaluation strategy).** *The following two conditions provide an evaluation strategy which ensures block-count($k$) in inter-procedural bytecode (i.e., we consider several CFGs and methods):*

*(i) annotate every atom in the evaluation tree with a multiset which contains its ancestor sequence which we refer to as* ancestors*;*
*(ii) atoms can only be reduced if there are at most $k - 1$ occurrences of the corresponding predicate in its* ancestors*.*

The next section provides practical means to implement this strategy.

## 6 Test Data Generation by Partial Evaluation

We have seen in Sec. 5 that a central issue when performing symbolic execution for TDG consists in building a finite (possibly unfinished) evaluation tree by using a non-standard execution strategy which ensures both a certain coverage criterion and termination. An important observation is that this is exactly the problem that *unfolding rules*, used in partial evaluators of (C)LP, solve. In essence, partial evaluators are non-standard interpreters which receive a set of partially instantiated atoms and evaluate them as determined by the so-called unfolding rule. Thus, the role of the unfolding rule is to supervise the process of building finite (possibly unfinished) SLD trees for the atoms. This view of TDG as a PE problem has important advantages. First, as we show in Sec. 6.1, we can directly apply existing, powerful, unfolding rules developed in the context of PE. Second, in Sec. 6.2, we show that it is possible to explore additional abilities of partial evaluators in the context of TDG. Interestingly, the generation of a residual program from the evaluation tree returns a program which can be used as a *test-case generator* for obtaining further test-cases.

### 6.1 Using an Unfolding Rule for Implementing Block-Count($k$)

Sophisticated unfolding rules exist which incorporate non-trivial mechanisms to stop the construction of SLD trees. For instance, unfolding rules based on comparable atoms allow expanding derivations as long as no previous *comparable* atom (same predicate symbol) has been already visited. As already discussed, the

use of ancestors [5] can reduce the number of atoms for which the comparability test has to be performed.

In PE terminology, the evaluation strategy outlined in Sec. 5 corresponds to an unfolding rule which allows $k$ comparable atoms in every ancestor sequence. Below, we provide an implementation, predicate `unfold/3`, of such an unfolding rule. The CLP decompiled program is stored as `clause/2` facts. Predicate `unfold/3` receives as input parameters an atom as the initial goal to evaluate, and the value of constant $k$. The third parameter is used to return the resolvent associated with the corresponding derivation.

```
unfold(A,K,[load_st(St)|Res]) :-
    unf([A],K,[],Res),
    collect_vars([A|Res],Vars),
    save_st(Vars,St).

unf([],_K,_AS,[]).
unf([A|R],K,AncS,Res) :-
    constraint(A),!, call(A),
    unf(R,K,AncS,Res).
unf(['$pop$'|R],K,[_|AncS],Res) :-
    !, unf(R,K,AncS,Res).
unf([A|R],K,AncS,Res) :-
    clause(A,B), functor(A,F,Ar),
    (check(AncS,F,Ar,K) ->
       append(B,['$pop$'|R],NewGoal),
       unf(NewGoal,K,[F/Ar|AncS],Res)
    ;  Res = [A|R]).

check([],_,_,K) :- K > 0.
check([F/Ar|As],F,Ar,K) :- !, K > 1,
    K1 is K - 1, check(As,F,Ar,K1).
check([_|As],F,Ar,K) :- check(As,F,Ar,K).
```

Predicate `unfold/3` first calls `unf/4` to perform the actual unfolding and then, after collecting the variables from the resolvent and the initial atom by means of predicate `collect_vars/2`, it saves the store of constraints in variable `St` so that it is included inside the call `load_st(St)` in the returned resolvent. The reason why we do this will become clear in Sect. 6.2. Let us now explain intuitively the four rules which define predicate `unf/4`. The first one corresponds to having an empty goal, i.e., the end of a successful derivation. The second rule corresponds to the first case in the operational semantics presented in Sec. 2, i.e., when the leftmost literal is a constraint. Note that in CLP there is no need to add an argument for explicitly passing around the store, which is implicitly maintained by the execution engine by simply executing constraints by means of predicate `call/1`. The second case of the operational semantics in Sec. 2, i.e., when the leftmost literal is an atom, corresponds to the fourth rule. Here, on backtracking we look for all rules asserted as `clause/2` facts whose head unifies with the leftmost atom. Note that depending on whether the number of occurrences of comparable atoms in the ancestors sequence is smaller than the given $k$ or not, the derivation continues or it is stopped. The termination check is performed by predicate `check/4`.

In order to keep track of ancestor sequences for every atom, we have adopted the efficient implementation technique, proposed in [25], based on the use of a global *ancestor stack*. Essentially, each time an atom `A` is unfolded using a rule $\mathtt{H} : -\mathtt{B_1}, \ldots, \mathtt{B_n}$, the predicate name of `A`, $pred(A)$, is pushed on the ancestor stack (see third argument in the recursive call). Additionally, a `$pop$` mark is added to the new goal after $\mathtt{B_1}, \ldots, \mathtt{B_n}$ (call to `append/3`) to delimit the scope of the predecessors of `A` such that, once those atoms are evaluated, we find the mark `$pop$` and can remove $pred(A)$ from the ancestor stacks. This way, the

ancestor stack, at each stage of the computation, contains the ancestors of the next atom which will be selected for resolution. If predicate `check/4` detects that the number of occurrences of $pred(A)$ is greater than $k$, the derivation is stopped and the current goal is returned in `Res`.The third rule of `unf/4` corresponds to the case where the leftmost atom is a $pop$ literal. This indicates that the the execution of the atom which is on top of the ancestor stack has been completed. Hence, this atom is popped from the stack and the $pop$ literal is removed from the goal.

*Example 5.* The execution of `unfold(lcm([X,Y],Z),2,[_])` builds a finite (and hence unfinished) version of the evaluation tree in Fig. 3. For $k = 2$, the infinite branch is stopped at goal R9, since the ancestor stack at this point is `[gcd6c, gcd4c,gcd4,gcd6c,gcd4c,gcd4,lcm]` and hence it already contains `gcd4` twice. This will make the `check/4` predicate fail and therefore the derivation is stopped. More interestingly, we can generate test-cases, if we consider the following call:

```
findall(([X,Y],Z),unfold([gen_test_data(lcm,[X,Y],[-1000,1000],Z)],2,[_]),TCases).
```

where `generate_test_data` is defined as in Sec. 4. Now, we get on backtracking, concrete values for variables `X`, `Y` and `Z` associated to each finished derivation of the tree.[2] They correspond to test data for the block-count(2) coverage criteria of the bytecode. In particular, we get the following set of test-cases: TCases = [([1,0],0), ([0,0],divby0), ([-1000,0],0), ([0,1],0), ([-1000,1],1000), ([-1000,-1000], 1000),([1,-1],1)] which correspond, respectively, to the leaves labeled as **(L1)**,...,**(L7)** in the evaluation tree of Fig. 3. Essentially, they constitute a particular set of concrete values that traverses all possible paths in the bytecode, including exceptional behaviours, and where the loop body is executed at most once.

The soundness of our approach to TDG amounts to saying that the above implementation, executed on the CLP decompiled program, ensures termination and block-count($k$) coverage on the original bytecode.

**Proposition 1 (soundness).** *Let $m$ be a method with $n$ arguments and $BC_m$ its bytecode instructions. Let* $\mathtt{m}([\mathtt{X_1},\ldots,\mathtt{X_n}],\mathtt{Y})$ *be the corresponding decompiled method and let the CLP block-level decompilation of $BC_m$ be asserted as a set of* `clause/2` *facts. For every positive number $k$, the set of successful derivations computed by* $\mathtt{unf}(\mathtt{m}([\mathtt{X_1},\ldots,\mathtt{X_n}],\mathtt{Y}),\mathtt{k},[],[],\_)$ *ensures block-count($k$) coverage of $BC_m$.*

Intuitively, the above result follows from the facts that: (1) the decompilation is correct and block-level, hence all traces in the bytecode are derivations in the decompiled program as well as loops in bytecode are cycles in CLP; (2) the unfolding rule computes all feasible paths and traverses cycles at most $k$ times.

[2] We force to consider just finished derivations by providing `[_]` as the obtained resultant.

### 6.2 Generating Test Data Generators

The final objective of a partial evaluator is to generate optimized *residual* code. In this section, we explore the applications of the code generation phase of partial evaluators in TDG. Let us first intuitively explain how code is generated. Essentially, the residual code is made up by a set of *resultants* or residual rules (i.e., a program), associated to the root-to-leaf derivations of the computed evaluation trees. For instance, consider the rightmost derivation of the tree in Fig. 3, the associated resultant is a rule whose head is the original atom (applying the mgu's to it) and the body is made up by the atoms in the leaf of the derivation. If we ignore the constraints gathered along the derivation (which are encoded in `load_st(S)` as we explain below), we obtain the following resultant:

```
lcm([X,Y],Z) :- load_st(S), gcd4(R,R',GCD), P #= X*Y, lcm1c([GCD,P],Z).
```

The residual program will be (hopefully) executed more efficiently than the original one since those computations that depend only on the static data are performed once and for all at specialization time. Due to the existence of incomplete derivations in evaluation trees, the residual program might not be complete (i.e., it can miss answers w.r.t. the original program). The partial evaluator includes an *abstraction* operator which is encharged of ensuring that the atoms in the leaves of incomplete derivations are "covered" by some previous (partially evaluated) atom and, otherwise, adds the uncovered atoms to the set of atoms to be partially evaluated. For instance, the atoms `gcd4(R,R',GCD)` and `lcm1c([GCD,P],Z)` above are not covered by the single previously evaluated atom `lcm([X,Y],Z)` as they are not instances of it. Therefore, a new unfolding process must be started for each of the two atoms. Hence the process of building evaluation trees by the unfolding operator is iteratively repeated while new atoms are uncovered. Once the final set of trees is obtained, the resultants are generated from their derivations as described above.

Now, we want to explore the issues behind the application of a full partial evaluator, with its code generation phase, for the purpose of TDG. Novel interesting questions arise: (*i*) *what kind of partial evaluator do we need to specialize decompiled CLP programs?;* (*ii*) *what do we get as residual code?;* (*iii*) *what are the applications of such residual code?* Below we try to answer these questions.

As regards question (*i*), we need to extend the mechanisms used in standard PE of logic programming to support constraints. The problem has been already tackled, e.g., by [8] to which we refer for more details. Basically, we need to take care of constraints at three different points: first, during the execution, as already done by `call` within our unfolding rule `unfold/3`; second, during the abstraction process, we can either define an accurate abstraction operator which handles constraints or, as we do below, we can take a simpler approach which safely ignores them; third, during code generation, we aim at generating *constrained* rules which integrate the *store* of constraints associated to their corresponding derivations. To handle the last point, we enhance our schema with the next two basic operations on constraints which are used by `unfold/3` and were left

unexplained in Sec. 6.1. The store is saved and projected by means of predicate `save_st/2`, which given a set of variables in its first argument, saves the current store of the CLP execution, projects it to the given variables and returns the result in its second argument. The store is loaded by means of `load_st/1` which given an explicit store in its argument adds the constraints to the current store. Let us illustrate this process by means of an example.

*Example 6.* Consider a partial evaluator of CLP which uses as control strategies: predicate `unfold/3` as unfolding rule and a simple abstraction operator based on the combination of the *most specific generalization* and a check of comparable terms (as the unfolding does) to ensure termination. Note that the abstraction operator ignores the constraint store. Given the entry, `gen_test_data(lcm,[X,Y],[-1000,1000],Z)`, we would obtain the following residual code for $k = 2$:

```
gen_test_data(lcm,[1,0],[-1000,1000],0).
gen_test_data(lcm,[0,0],[-1000,1000],
              divby0).
...
gen_test_data(lcm,[X,Y],[-1000,1000],Z) :-
    load_st(S1), gcd4(R,R',GCD),
    P #= X*Y, lcm1c([GCD,P],Z),
    once(labeling([ff],[X,Y])).
```

```
gcd4(R,0,R) :- load_st(S2).
gcd4(R,0,GCD) :- load_st(S3).
gcd4(R,R',GCD) :- load_st(S4),
    gcd4(R',R'',GCD).

lcm1c([GCD,P],Z) :- load_st(S5).
lcm1c([GCD,P],Z) :- load_st(S6).
lcm1c([0,_P],divby0).
```

The residual code for `gen_test_data/4` contains eight rules. The first seven ones are facts corresponding to the seven successful branches (see Fig. 3). Due to space limitations here we only show two of them. Altogether they represent the set of test-cases for the block-count(2) coverage criteria (those in Ex. 6.1). It can be seen that all rules (except the facts[3]) are constrained as they include a residual call to `load_st/1`. The argument of `load_st/1` contains a syntactic representation of the store at the last step of the corresponding derivation. Again, due to space limitations we do not show the stores. As an example, `S1` contains the store associated to the rightmost derivation in the tree of Fig. 3, namely {`X in -1000..1000, Y in (-1000..-1)∨(1..1000), R in (-999.. -1) ∨ (1..999), R' in -998..998, R = X mod Y, R' = Y mod R`}. This store acts as a guard which comprises the constraints which avoid the execution of the paths previously computed to obtain the seven test-cases above.

We can now answer issue (*ii*): it becomes apparent from the example above that we have obtained a program which is a *generator* of test-cases for larger values of $k$. The execution of the generator will return by backtracking the (infinite) set of values exercising all possible execution paths which traverse blocks more than twice. In essence, our test-case generators are CLP programs whose execution in CLP returns further test-cases on demand for the bytecode under test and without the need of starting the TDG process from scratch.

[3] For the facts, there is no need to consider the store, because a call to `labeling` has removed all variables.

Here, it comes issue (*iii*): Are the above generators useful? How should we use them? In addition to execution (see inherent problems in Sec. 4), we might further partially evaluate them. For instance, we might partially evaluate the above specialized version of `gen_test_data/4` (with the same entry) in order to incrementally generate test-cases for larger values of $k$. It is interesting to observe that by using $k = 1$ for all atoms different from the initial one, this further specialization will just increment the number of `gen_test_data/4` facts (producing more concrete test-cases) but the rest of the residual program will not change, in fact, there is no need to re-evaluate it later.

## 7 Conclusions and Related Work

We have proposed a methodology for test data generation of imperative, low-level code by means of existing partial evaluation techniques developed for constraint logic programs. Our approach consist of two separate phases: (1) the compilation of the imperative bytecode to a CLP program and (2) the generation of test-cases from the CLP program. It naturally raises the question whether our approach can be applied to other imperative languages in addition to bytecode. This is interesting as existing approaches for Java [23], and for C [13], struggle for dealing with features like recursion, method calls, dynamic memory, etc. during symbolic execution. We have shown in the paper that these features can be uniformly handled in our approach after the transformation to CLP. In particular, all kinds of loops in the bytecode become uniformly represented by recursive predicates in the CLP program. Also, we have seen that method calls are treated in the same way as calls to blocks. In principle, this transformation can be applied to any language, both to high-level and to low-level bytecode, the latter as we have seen in the paper. In every case, our second phase can be applied to the transformed CLP program.

Another issue is whether the second phase can be useful for test-case generation of CLP programs, which are not necessarily obtained from a decompilation of an imperative code. Let us review existing work for declarative programs. Test data generation has received comparatively less attention than for imperative languages. The majority of existing tools for functional programs are based on black-box testing [6,18]. Test cases for logic programs are obtained in [24] by first computing constraints on the input arguments that correspond to execution paths of logic programs and then solving these constraints to obtain test inputs for the corresponding paths. This corresponds essentially to the naive approach discussed in Sec. 4, which is not sufficient for our purposes as we have seen in the paper. However, in the case of the generation of test data for regular CLP programs, we are interested not only in successful derivations (execution paths), but also in the failing ones. It should be noted that the execution of CLP decompiled programs, in contrast to regular CLP programs, for any actual input values is guaranteed to produce exactly one solution because the operational semantics of bytecode is deterministic. For functional logic languages, specific coverage criteria are defined in [10] which capture the control flow of these languages as well

as new language features are considered, namely laziness. In general, declarative languages pose different problems to testing related to their own execution models –like laziness in functional languages and failing derivations in (C)LP– which need to be captured by appropriate coverage criteria. Having said this, we believe our ideas related to the use of PE techniques to generate test data generators and the use of unfolding rules to supervise the evaluation could be adapted to declarative programs and remains as future work.

Our work is a proof-of-concept that partial evaluation of CLP is a powerful technique for carrying out TDG in imperative low-level languages. To develop our ideas, we have considered a simple imperative bytecode language and left out object-oriented features which require a further study. Also, our language is restricted to integer numbers and the extension to deal with real numbers is subject of future work. We also plan to carry out an experimental evaluation by transforming Java bytecode programs from existing test suites to CLP programs and then trying to obtain useful test-cases. When considering realistic programs with object-oriented features and real numbers, we will surely face additional difficulties. One of the main practical issues is related to the scalability of our approach. An important threaten to scalability in TDG is the so-called infeasibility problem [27]. It happens in approaches that do not handle constraints along the construction of execution paths but rather perform two independent phases (1) path selection and 2) constraint solving). As our approach integrates both parts in a single phase, we do not expect scalability limitations in this regard. Also, a challenging problem is to obtain a decompilation which achieves a manageable representation of the heap. This will be necessary to obtain test-cases which involve data for objects stored in the heap. For the practical assessment, we also plan to extend our technique to include further coverage criteria. We want to consider other classes of coverage criteria which, for instance, generate test-cases which cover a certain statement in the program.

**Acknowledgments.** This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

## References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers - Principles, Techniques and Tools. Addison-Wesley, Reading (1986)
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: Cost Analysis of Java Bytecode. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 157–172. Springer, Heidelberg (2007)
3. Albert, E., Gómez-Zamalloa, M., Hubert, L., Puebla, G.: Verification of Java Bytecode using Analysis and Transformation of Logic Programs. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 124–139. Springer, Heidelberg (2006)

4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. LNCS, vol. 4334. Springer, Heidelberg (2007)
5. Bruynooghe, M., De Schreye, D., Martens, B.: A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. New Generation Computing 1(11), 47–79 (1992)
6. Claessen, K., Hughes, J.: Quickcheck: A lightweight tool for random testing of haskell programs. In: ICFP, pp. 268–279 (2000)
7. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Trans. Software Eng. 2(3), 215–222 (1976)
8. Craig, S.-J., Leuschel, M.: A compiler generator for constraint logic programs. In: Ershov Memorial Conference, pp. 148–161 (2003)
9. Ferguson, R., Korel, B.: The chaining approach for software test data generation. ACM Trans. Softw. Eng. Methodol. 5(1), 63–86 (1996)
10. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: PPDP, pp. 63–74 (2007)
11. Futamura, Y.: Partial evaluation of computation process - an approach to a compiler-compiler. Systems, Computers, Controls 2(5), 45–50 (1971)
12. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Modular Decompilation of Low-Level Code by Partial Evaluation. In: 8th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), pp. 239–248. IEEE Computer Society, Los Alamitos (2008)
13. Gotlieb, A., Botella, B., Rueher, M.: A clp framework for computing structural test data. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS, vol. 1861, pp. 399–413. Springer, Heidelberg (2000)
14. Gupta, N., Mathur, A.P., Soffa, M.L.: Generating test data for branch coverage. In: Automated Software Engineering, pp. 219–228 (2000)
15. Henriksen, K.S., Gallagher, J.P.: Abstract interpretation of pic programs through logic programming. In: SCAM 2006: Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation, pp. 184–196. IEEE Computer Society, Los Alamitos (2006)
16. Howden, W.E.: Symbolic testing and the dissect symbolic evaluation system. IEEE Transactions on Software Engineering 3(4), 266–278 (1977)
17. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976)
18. Koopman, P., Alimarine, A., Tretmans, J., Plasmeijer, R.: Gast: Generic automated software testing. In: IFL, pp. 84–100 (2002)
19. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, Reading (1996)
20. Marriot, K., Stuckey, P.: Programming with Constraints: An Introduction. MIT Press, Cambridge (1998)
21. Méndez-Lojo, M., Navas, J., Hermenegildo, M.: A Flexible (C)LP-Based Approach to the Analysis of Object-Oriented Programs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915. Springer, Heidelberg (2008)
22. Meudec, C.: Atgen: Automatic test data generation using constraint logic programming and symbolic execution. Softw. Test., Verif. Reliab. 11(2), 81–96 (2001)
23. Müller, R.A., Lembeck, C., Kuchen, H.: A symbolic java virtual machine for test case generation. In: IASTED Conf. on Software Engineering, pp. 365–371 (2004)
24. Mweze, N., Vanhoof, W.: Automatic generation of test inputs for mercury programs. In: Pre-proceedings of LOPSTR 2006 (July 2006) (extended abstract)

25. Puebla, G., Albert, E., Hermenegildo, M.: Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 149–165. Springer, Heidelberg (2005)
26. Swedish Institute for Computer Science, PO Box 1263, S-164 28 Kista, Sweden. SICStus Prolog 3.8 User's Manual, 3.8 edition (October 1999), `http://www.sics.se/sicstus/`
27. Zhu, H., Patrick, A., Hall, V., John, H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. 29(4), 366–427 (1997)

# A Modular Equational Generalization Algorithm*

María Alpuente[1], Santiago Escobar[1], José Meseguer[2], and Pedro Ojeda[1]

[1] Universidad Politécnica de Valencia, Spain
{alpuente,sescobar,pojeda}@dsic.upv.es
[2] University of Illinois at Urbana–Champaign, USA
meseguer@cs.uiuc.edu

**Abstract.** This paper presents a modular equational generalization algorithm, where function symbols can have any combination of associativity, commutativity, and identity axioms (including the empty set). This is suitable for dealing with functions that obey algebraic laws, and are typically mechanized by means of equational atributes in rule-based languages such as ASF+SDF, Elan, OBJ, Cafe-OBJ, and Maude. The algorithm computes a complete set of least general generalizations modulo the given equational axioms, and is specified by a set of inference rules that we prove correct. This work provides a missing connection between least general generalization and computing modulo equational theories, and opens up new applications of generalization to rule-based languages, theorem provers and program manipulation tools such as partial evaluators, test case generators, and machine learning techniques, where function symbols obey algebraic axioms. A Web tool which implements the algorithm has been developed which is publicly available.

## 1 Introduction

The problem of ensuring termination of program manipulation techniques arises in many areas of computer science, including automatic program analysis, synthesis, verification, specialisation, and transformation. An important component for ensuring termination of these techniques is a generalization algorithm (also called anti–unification) that, for a pair of input expressions, returns its least general generalization (lgg), i.e., a generalization that is more specific than any other such generalization. Whereas unification produces most general unifiers that, when applied to two expressions, make them equivalent to the most general common instance of the inputs [21], generalization abstracts the inputs by computing their most specific generalization. As in unification, where the most general unifier (mgu) is of interest, in the sequel we are interested in the least general generalization (lgg) or, as we shall see for the equational case treated in

* This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grant TIN 2007-68093-C02-02, Integrated Action HA 2006-0007, UPV PAID-06-07 project, and Generalitat Valenciana under grants GVPRE/2008/113 and BFPI/2007/076; also by NSF Grant CNS 07-16638.

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 24–39, 2009.

this paper, in a minimal and complete set of lggs, which is the dual analogue of a minimal and complete set of unifiers for equational unification problems [6].

For instance, in the partial evaluation (PE) of logic programs [17], the general idea is to construct a set of finite (possibly partial) deduction trees for a set of initial calls, and then extract from those trees a new program $P$ that allows any instance of the calls to be executed. To ensure that the partially evaluated program *covers* all these calls, most PE procedures recursively specialize some calls that are dynamically produced during this process. This requires some kind of generalization in order to enforce the termination of the process: if a call occurring in $P$ that is not sufficiently covered by the program *embeds* an already evaluated call, both calls are generalized by computing their lgg. In the literature on partial evaluation, least general generalization is also known as *most specific generalization* (msg) and *least common anti–instance* (lcai) [24].

The computation of lggs is also central to most program synthesis and learning algorithms such as those developed in the area of inductive logic programming [25], and also to conjecture lemmas in inductive theorem provers such as Nqthm [10] and its ACL2 extension [20]. Least general generalization was originally introduced by Plotkin in [28], see also [31]. Actually, Plotkin's work originated from the consideration in [30] that, since unification is useful in automatic deduction by the resolution method, its dual might prove helpful for induction. Anti-unification is also used in test case generation techniques to achieve appropriate coverage [7].

Quite often, however, all the above applications of generalization may have to be carried out in contexts in which the function symbols satisfy certain *equational axioms*. For example, in rule-based languages such as ASF+SDF [8], Elan [9], OBJ [18], CafeOBJ [14], and Maude [11] some function symbols may be declared to obey given algebraic laws (the so-called *equational attributes* of OBJ, CafeOBJ and Maude), whose effect is to compute with equivalence classes modulo such axioms while avoiding the risk of non–termination. Similarly, theorem provers, both general first-order logic ones and inductive theorem provers, routinely support commonly occurring equational theories for some function symbols such as associativity-commutativity. In yet another area, [15,16] describes rule-based applications to security protocol verification, where symbolic reachability analyses modulo algebraic properties allow one to reason about security in the face of attempted attacks on low-level algebraic properties of the functions used in the protocol (e.g. commutative encryption). A survey of algebraic properties used in cryptographic protocols can be found in [13].

Surprisingly, unlike the dual case of equational unification, which has been thoroughly investigated (see, e.g., the surveys [32,6]), to the best of our knowledge there seems to be no treatment of generalization modulo an equational theory $E$. This paper makes a novel contribution in this area by developing a modular family of E-generalization algorithms where the theory $E$ is parametric: any binary function symbol in the given signature can have any combination of associativity, commutativity, and identity axioms (including the empty set of such axioms).

To better motivate our work, let us first recall the standard generalization problem. Let $t_1$ and $t_2$ be two terms. We want to find a term $s$ that generalizes both $t_1$ and $t_2$. In other words, both $t_1$ and $t_2$ must be substitution instances of $s$. Such a term is, in general, not unique. For example, let $t_1$ be the term $f(f(a,a),b)$ and let $t_2$ be $f(f(b,b),a)$. Then $s = x$ trivially generalizes the two terms, with $x$ being a variable. Another possible generalization is $f(x,y)$, with $y$ being also a variable. The term $f(f(x,x),y)$ has the advantage of being the most 'specific' or *least general generalization (lgg)* (modulo variable renaming).

In the case where the function symbols do not satisfy any algebraic axioms, the lgg is unique up to variable renaming. However, when we want to reason *modulo* certain axioms for the different function symbols in our problem, lggs no longer need to be unique. This is analogous to equational unification problems, where in general there is no single mgu, but a set of them. Let us, for example, consider that the above function symbol $f$ is associative and commutative. Then the term $f(f(x,x),y)$ is not the only least general generalization of $f(f(a,a),b)$ and $f(f(b,b),a)$, because another incomparable generalization exists, namely, $f(f(x,a),b)$.

Similarly to the case of equational unification [32], things are not so easy as for syntatic generalization, and the dual problem of computing least general $E$-generalizations is a difficult one, particularly in managing the algorithmic complexity of the problem. The significance of equational generalization was already pointed out by Pfenning in [27]: "It appears that the intuitiveness of generalizations can be significantly improved if anti–unification takes into account additional equations which come from the object theory under consideration. It is conceivable that there is an interesting theory of equational anti–unification to be discovered". In this work, we do not address the $E$-generalization problem in its fullest generality. Instead, we study in detail a *modular* algorithm for a *parametric* family of commonly occurring equational theories, namely, for all theories $(\Sigma, E)$ such that each binary function symbol $f \in \Sigma$ can have any combination of the following axioms: (i) *associativity* $(A_f)$ $f(x, f(y,z)) = f(f(x,y),z)$; (ii) *commutativity* $(C_f)$ $f(x,y) = f(y,x)$, and (iii) *identity* $(U_f)$ for a constant symbol, say, $e$, i.e., $f(x,e) = x$ and $f(e,x) = x$. In particular, $f$ may not satisfy any such axioms, which when it happens for all binary symbols $f \in \Sigma$ gives us the standard generalization algorithm as a special case.

**Our Contribution**

The main contributions of the paper can be summarized as follows:

- A modular equational generalization algorithm specified as a set of inference rules, where different function symbols satisfying different associativity and/or commutativity and/or identity axioms have different inference rules. To the best of our knowledge, this is the first equational least general generalization algorithm in the literature.
- Correctness and termination results for our $E$–generalization algorithm.
- A prototypical implementation of the $E$-generalization algorithm which is publicly available.

The algorithm is *modular* in the precise sense that the combination of different equational axioms for different function symbols is automatic and seamless: the inference rules can be applied to generalization problems involving each symbol with no need whatsoever for any changes or adaptations. This is similar to, but much simpler and easier than, modular methods for combining $E$-unification algorithms (e.g., [6]). We illustrate our inference system with several examples.

As already mentioned, our $E$-generalization algorithm should be of interest to developers of rule-based languages, theorem provers and equational reasoning programs, as well as program manipulation tools such as program analyzers, partial evaluators, test case generators, and machine learning tools, for declarative languages and reasoning systems supporting commonly occurring equational axioms such as associativity, commutativity and identity efficiently in a built–in way. For instance, this includes many theorem provers, and a variety of rule-based languages such as ASF+SDF, OBJ, CafeOBJ, Elan, and Maude.

**Related Work**

Although generalization goes back to work of Plotkin [28], Reynolds [31], and Huet [19], and has been studied in detail by other authors (see for example the survey [21]), to the best of our knowledge, we are not aware of any existing equational generalization algorithm modulo any combination of associativity, commutativity and identity axioms. While Plotkin [28] and Reynolds [31] gave imperative–style algorithms for generalization, which are both essentially the same, Huet's generalization algorithm was formulated as a pair of recursive equations [19]. Least general generalization in an order–sorted typed setting was studied in [1]. In [3], we specified the generalization process by means of an inference system and then extended it naturally to order-sorted generalization. Pfenning [27] gave an algorithm for generalization in the higher-order setting of the calculus of constructions which does not consider either order-sorted theories or equational axioms.

**Plan of the Paper**

After some preliminaries in Section 2, we present in Section 3 a syntactic generalization algorithm as a special case to motivate its equational extension. Then in Section 4 we show how this calculus naturally extends to a new, modular generalization algorithm modulo ACU. We illustrate the use of the inference rules with several examples. Finally, we prove the correctness of our inference system. Section 5 concludes. Proofs of the technical results can be found in [2].

## 2 Preliminaries

We follow the classical notation and terminology from [33] for term rewriting and from [22,23] for rewriting logic. We assume an *unsorted signature* $\Sigma$ with a finite number of function symbols. We assume an enumerable set of variables $\mathcal{X}$. A *fresh* variable is a variable that appears nowhere else. We write $\mathcal{T}(\Sigma, \mathcal{X})$ and

$\mathcal{T}(\Sigma)$ for the corresponding term algebras. For a term $t$, we write $Var(t)$ for the set of all variables in $t$. The set of positions of a term $t$ is written $Pos(t)$, and the set of non-variable positions $Pos_\Sigma(t)$. The root position of a term is $\Lambda$. The subterm of $t$ at position $p$ is $t|_p$ and $t[u]_p$ is the term $t$ where $t|_p$ is replaced by $u$. By $root(t)$ we denote the symbol occurring at the root position of $t$.

A *substitution* $\sigma$ is a mapping from a finite subset of $\mathcal{X}$, written $Dom(\sigma)$, to $\mathcal{T}(\Sigma, \mathcal{X})$. The set of variables introduced by $\sigma$ is $Ran(\sigma)$. The identity substitution is *id*. Substitutions are homomorphically extended to $\mathcal{T}(\Sigma, \mathcal{X})$. The application of a substitution $\sigma$ to a term $t$ is denoted by $t\sigma$. The restriction of $\sigma$ to a set of variables $V$ is $\sigma|_V$. Composition of two substitutions is denoted by juxtaposition, i.e., $\sigma\sigma'$. We call a substitution $\sigma$ a *renaming* if there is another substitution $\sigma^{-1}$ such that $\sigma\sigma^{-1}|_{Dom(\sigma)} = id$.

A *$\Sigma$-equation* is an unoriented pair $t = t'$. An *equational theory* $(\Sigma, E)$ is a set of $\Sigma$-equations. An equational theory $(\Sigma, E)$ is *regular* if for each $t = t' \in E$, we have $Var(t) = Var(t')$. Given $\Sigma$ and a set $E$ of $\Sigma$-equations, equational logic induces a congruence relation $=_E$ on terms $t, t' \in \mathcal{T}(\Sigma, \mathcal{X})$ (see [23]).

The *$E$-subsumption* preorder $\leq_E$ (simply $\leq$ when $E$ is empty) holds between $t, t' \in \mathcal{T}(\Sigma, \mathcal{X})$, denoted $t \leq_E t'$ (meaning that $t$ is more general than $t'$ modulo $E$), if there is a substitution $\sigma$ such that $t\sigma =_E t'$; such a substitution $\sigma$ is said to be an *$E$-matcher* for $t'$ in $t$. The $E$-renaming equivalence $t \simeq_E t'$ (or $\simeq$ if $E$ is empty), holds if there is a renaming $\theta$ such that $t\theta =_E t'$. We write $t <_E t'$ (or $<$ if $E$ is empty) if $t \leq_E t'$ and $t \not\simeq_E t'$.

## 3 Syntactic Least General Generalization

In this section we revisit syntactic generalization [19], giving a novel inference system that will be useful in our subsequent extension of this algorithm to the equational setting given in Section 4.

Most general unification of a (unifiable) set $M$ is the least upper bound (most general instance) of $M$ under $\leq$. Generalization corresponds to the greatest lower bound. Given a non–empty set $M$ of terms, the term $w$ is a *generalization* of $M$ if, for all $s \in M$, $w \leq s$. A term $w$ is the *least general generalization* (lgg) of $M$ if $w$ is a generalization of $M$ and, for each other generalization $u$ of $M$, $u \leq w$.

The non-deterministic generalization algorithm $\lambda$ of Huet [19] (also treated in detail in [21]) is as follows. Let $\Phi$ be any bijection between $\mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{T}(\Sigma, \mathcal{X})$ and a set of variables $V$. The recursive function $\lambda$ on $\mathcal{T}(\Sigma, \mathcal{X}) \times \mathcal{T}(\Sigma, \mathcal{X})$ that computes the lgg of two terms is given by:

- $\lambda(f(s_1, \ldots, s_m), f(t_1, \ldots, t_m)) = f(\lambda(s_1, t_1), \ldots, \lambda(s_m, t_m))$, for $f \in \Sigma$
- $\lambda(s, t) = \Phi(s, t)$, otherwise.

Central to this algorithm is the global function $\Phi$ that is used to guarantee that the same disagreements are replaced by the same variable in both terms.

In [3], we have provided a novel set of inference rules for computing the (syntactic) least generalization of two terms, that uses a local store of already solved generalization sub-problems. The advantage of using such a store is that, differently from the global repository $\Phi$, our stores are local to the computation traces.

This non–globality of the stores is the key for both, the order–sorted version of [3] and the equational generalization algorithm developed in this work, which computes a complete and minimal set of least general $E$-generalizations.

In our reformulation [3], we represent a generalization problem between terms $s$ and $t$ as a *constraint* $s \overset{x}{\triangleq} t$, where $x$ is a fresh variable that stands for the (most general) generalization of $s$ and $t$. By means of this representation, any generalization $w$ of $s$ and $t$ is given by a suitable substitution $\theta$ such that $x\theta = w$.

We compute the least general generalization of $s$ and $t$, written $lgg(s,t)$, by means of a transition system $(Conf, \rightarrow)$ [29] where $Conf$ is a set of *configurations* and the transition relation $\rightarrow$ is given by a set of inference rules. Besides the *constraint component*, i.e., a set of constraints of the form $t_i \overset{x_i}{\triangleq} t_{i'}$, and the *substitution component*, i.e., the partial substitution computed so far, configurations also include an extra component, called the *store*.

**Definition 1.** *A configuration* $\langle CT \mid S \mid \theta \rangle$ *consists of three components: (i) the* constraint component CT*, i.e., a conjuntion* $s_1 \overset{x_1}{\triangleq} t_1 \wedge \ldots \wedge s_n \overset{x_n}{\triangleq} t_n$ *that represents the* set of unsolved constraints*, (ii) the* store component S*, that records the* set of already solved constraints*, and (iii) the* substitution component $\theta$*, that consists of bindings for some variables previously met during the computation.*

Starting from the initial configuration $\langle t \overset{x}{\triangleq} t' \mid \emptyset \mid id \rangle$, configurations are transformed until a terminal configuration $\langle \emptyset \mid S \mid \theta \rangle$ is reached. Then, the lgg of $t$ and $t'$ is given by $x\theta$. As we shall see, $\theta$ is unique up to renaming.

The transition relation $\rightarrow$ is given by the smallest relation satisfying the rules in Figure 1. In this paper, variables of terms $t$ and $s$ in a generalization problem $t \overset{x}{\triangleq} s$ are considered as constants, and are never instantiated. The meaning of the rules is as follows.

- The rule **Decompose** is the syntactic decomposition generating new constraints to be solved.
- The rule **Solve** checks that a constraint $t \overset{x}{\triangleq} s \in CT$ with $root(t) \not\equiv root(s)$, is not already solved. If not already there, the solved constraint $t \overset{x}{\triangleq} s$ is added to the store $S$.
- The rule **Recover** checks if a constraint $t \overset{x}{\triangleq} s \in CT$ with $root(t) \not\equiv root(s)$, is already solved, i.e., if there is already a constraint $t \overset{y}{\triangleq} s \in S$ for the same *conflict pair* $(t,s)$, with variable $y$. This is needed when the input terms of the generalization problem contain the same conflict pair more than once, e.g., the lgg of $f(a,a,a)$ and $f(b,b,a)$ is $f(y,y,a)$.

Note that the inference rules of Figure 1 are non–deterministic (i.e., they depend on the chosen constraint of the set $CT$). However, they are confluent up to variable renaming (i.e., the chosen transition is irrelevant for computation of terminal configurations). This justifies the well-known fact that the least general generalization of two terms is unique up to variable renaming [21].

**Decompose**

$$\frac{f \in (\Sigma \cup \mathcal{X})}{\langle f(t_1,\ldots,t_n) \overset{x}{\triangleq} f(t'_1,\ldots,t'_n) \wedge CT \mid S \mid \theta\rangle \rightarrow \langle t_1 \overset{x_1}{\triangleq} t'_1 \wedge \ldots \wedge t_n \overset{x_n}{\triangleq} t'_n \wedge CT \mid S \mid \theta\sigma\rangle}$$

where $\sigma = \{x \mapsto f(x_1,\ldots,x_n)\}$, $x_1,\ldots,x_n$ are fresh variables, and $n \geq 0$

**Solve**

$$\frac{root(t) \not\equiv root(t') \wedge \nexists y : t \overset{y}{\triangleq} t' \in S}{\langle t \overset{x}{\triangleq} t' \wedge CT \mid S \mid \theta\rangle \rightarrow \langle CT \mid S \wedge t \overset{x}{\triangleq} t' \mid \theta\rangle}$$

**Recover**

$$\frac{root(t) \not\equiv root(t')}{\langle t \overset{x}{\triangleq} t' \wedge CT \mid S \wedge t \overset{y}{\triangleq} t' \mid \theta\rangle \rightarrow \langle CT \mid S \wedge t \overset{y}{\triangleq} t' \mid \theta\sigma\rangle}$$

where $\sigma = \{x \mapsto y\}$

**Fig. 1.** Rules for least general generalization

$$\begin{array}{c}
lgg(f(g(a),g(y),a), f(g(b),g(y),b)) \\
\downarrow \text{ Initial Configuration} \\
\langle f(g(a),g(y),a) \overset{x}{\triangleq} f(g(b),g(y),b) \mid \emptyset \mid id\rangle \\
\downarrow \text{ Decompose} \\
\langle g(a) \overset{x_1}{\triangleq} g(b) \wedge g(y) \overset{x_2}{\triangleq} g(y) \wedge a \overset{x_3}{\triangleq} b \mid \emptyset \mid \{x \mapsto f(x_1,x_2,x_3)\}\rangle \\
\downarrow \text{ Decompose} \\
\langle a \overset{x_4}{\triangleq} b \wedge g(y) \overset{x_2}{\triangleq} g(y) \wedge a \overset{x_3}{\triangleq} b \mid \emptyset \mid \{x \mapsto f(g(x_4),x_2,x_3), x_1 \mapsto g(x_4)\}\rangle \\
\downarrow \text{ Solve} \\
\langle g(y) \overset{x_2}{\triangleq} g(y) \wedge a \overset{x_3}{\triangleq} b \mid a \overset{x_4}{\triangleq} b \mid \{x \mapsto f(g(x_4),x_2,x_3), x_1 \mapsto g(x_4)\}\rangle \\
\downarrow \text{ Decompose} \\
\langle y \overset{x_5}{\triangleq} y \wedge a \overset{x_3}{\triangleq} b \mid a \overset{x_4}{\triangleq} b \mid \{x \mapsto f(g(x_4),g(x_5),x_3), x_1 \mapsto g(x_4), x_2 \mapsto g(x_5)\}\rangle \\
\downarrow \text{ Decompose} \\
\langle a \overset{x_3}{\triangleq} b \mid a \overset{x_4}{\triangleq} b \mid \{x \mapsto f(g(x_4),g(y),x_3), x_1 \mapsto g(x_4), x_2 \mapsto g(y), x_5 \mapsto y\}\rangle \\
\downarrow \text{ Recover} \\
\langle \emptyset \mid a \overset{x_4}{\triangleq} b \mid \{x \mapsto f(g(x_4),g(y),x_4), x_1 \mapsto g(x_4), x_2 \mapsto g(y), x_5 \mapsto y, x_3 \mapsto x_4\}\rangle
\end{array}$$

**Fig. 2.** Computation trace for (syntactic) generalization of terms $f(g(a),g(y),a)$ and $f(g(b),g(y),b)$

*Example 1.* Let $t = f(g(a),g(y),a)$ and $s = f(g(b),g(y),b)$ be two terms. We apply the inference rules of Figure 1 and the substitution obtained by the lgg algorithm is $\theta = \{x \mapsto f(g(x_4),g(y),x_4), x_1 \mapsto g(x_4), x_2 \mapsto g(y), x_5 \mapsto y, x_3 \mapsto x_4\}$, where the lgg is $x\theta = f(g(x_4),g(y),x_4)$. Note that variable $x_4$ is repeated, to ensure the least general generalization. The execution trace is showed in Figure 2.

Termination and confluence (up to variable renaming) of the transition system $(Conf, \rightarrow)$ are straightforward. Soundness and completeness are proved as follows.

**Theorem 1.** [3] *Given terms $t$ and $t'$ and a fresh variable $x$, $u$ is the lgg of $t$ and $t'$ iff $\langle t \overset{x}{\triangleq} t' \mid \emptyset \mid id\rangle \rightarrow^* \langle \emptyset \mid S \mid \theta\rangle$ and there is a renaming $\rho$ s.t. $u\rho = x\theta$.*

Let us mention that the generalization algorithm of [3] recalled above can also be used to compute (thanks to associativity and commutativity of lgg) the lgg

of an arbitrary set of terms by successively computing the lgg of two elements of the set in the obvious way.

## 4 Least General Generalizations modulo E

When we have an equational theory $E$, the notion of least general generalization has to be broadened, because, there may exist $E$-generalizable terms that do not have a unique least general generalization. Similarly to the dual case of $E$-unification, we have to talk about a *set* of least general generalizations.

For a set $M$ of terms, we define the set of most specific generalizations of $M$ modulo $E$ as the set of *maximal lower bounds* of $M$ under $<_E$, i.e., $lgg_E(M) = \{u \mid \forall m \in M, u \leq_E m \wedge \nexists u' \ (u <_E u' \wedge \forall m \in M, u' \leq_E m)\}$.

*Example 2.* Consider terms $t = f(a, a, b)$ and $s = f(b, b, a)$ where $f$ is associative and commutative, and $a$ and $b$ are constants. Terms $u = f(x, x, y)$ and $u' = f(x, a, b)$ are generalizations of $t$ and $s$ but they are not comparable, i.e., no one is an instance of the other modulo the $AC$ axioms of $f$.

Given a finite set of equations $E$, and given two terms $t$ and $s$, we can always recursively enumerate the set which is by construction a complete set of generalizations of $t$ and $s$. For this, we only need to recursively enumerate all pairs of terms $(u, u')$ with $t =_E u$ and $s =_E u'$ and compute $lgg(u, u')$. Of course, this set $gen_E(t, s)$ may easily be infinite. However, if the theory $E$ has the additional property that each $E$-equivalence class is *finite* and can be effectively generated, then the above process becomes a terminating *algorithm*, generating a finite complete set of generalizations of $t$ and $s$.

In any case, for any finite set of equations $E$, we can always mathematically characterize a *minimal complete set* of $E$-generalizations, namely as a set $lgg_E(t, s)$ defined as follows.

**Definition 2.** *Let $t$ and $s$ be terms and let $E$ be an equational theory. A complete set of generalizations of $t$ and $s$ modulo $E$, denoted by $gen_E(t, s)$, is defined as follows: $gen_E(t, s) = \{v \mid \exists u, u' \ s.t. \ t =_E u \wedge s =_E u' \wedge lgg(u, u') = v\}$.*
*The set of least general generalizations of $t$ and $s$ modulo $E$ is defined as follows:*

$$lgg_E(t, s) = maximal_{<_E}(gen_E(t, s))$$

*where $maximal_{<_E}(S) = \{s \in S \mid \nexists s' \in S : s <_E s'\}$. Lggs are equivalent modulo renaming and, therefore, we remove from $lgg_E(t, t')$ renamed versions of terms.*

The following result is immediate.

**Theorem 2.** *Given terms $t$ and $s$ in an equational theory $E$, $lgg_E(t, s)$ is a minimal, correct, and complete set of lggs modulo $E$ of $t$ and $s$ (up to renaming).*

However, note that in general the relation $t <_E t'$ is *undecidable*, so that the above set, although definable at the mathematical level, cannot be effectively

computed. Nevertheless, when: (i) each $E$-equivalence class is *finite* and can be effectively generated; and (ii) there is an $E$-matching algorithm, then we also have an effective algorithm for computing $lgg_E(t,s)$, since the relation $t \leq_E t'$ is precisely the $E$-matching relation.

In summary, when $E$ is finite and satisfies conditions (i) and (ii), the above definitions give us an effective, although horribly inefficient, procedure to compute a finite, minimal, and complete set of least general generalizations $lgg_E(t,s)$. This naive algorithm could be used when $E$ consists of associativity and/or commutativity axioms for some functions symbols, because such theories (a special case of our proposed parametric family of theories) all satisfy conditions (i)–(ii). However, when we add identiy axioms, $E$-equivalence classes become infinite, so that the above approach no longer gives us a lgg algorithm modulo $E$.

In the following sections, we do provide a modular, minimal, terminating, sound, and complete algorithm for equational theories containing different axioms such as associativity, commutativity, and identity (and their combinations). The set $lgg_E(t,s)$ of least general $E$-generalizations is computed in two phases: (i) first a complete set of $E$-generalizations is computed by the inference rules given below, and then (ii) they are filtered to obtain $lgg_E(t,s)$ by using the fact that for all theories $E$ in the parametric family of theories we consider in this paper, there is a matching algorithm modulo $E$. We consider that a given function symbol $f$ in the signature $\Sigma$ obeys a subset of axioms $ax(f) \subseteq \{A_f, C_f, U_f\}$. In particular, $f$ may not satisfy any such axioms, $ax(f) = \emptyset$.

Let us provide our inference rules for equational generalization in a stepwise manner. First, $ax(f) = \emptyset$, then, $ax(f) = \{C_f\}$, then, $ax(f) = \{A_f\}$, then, $ax(f) = \{A_f, C_f\}$, and finally, $\{U_f\} \in ax(f)$. Technically, variables of the original terms are handled in our rules as constants, thus without any attribute, i.e., for any variable $x \in X$, we consider $ax(x) = \emptyset$.

### 4.1 Basic Rules for Least General $E$–Generalization

Let us start with a set of basic rules that are the equational version of the syntactic generalization rules of Section 3. The rule $Decompose_E$ applies to function symbols obeying no axioms, $ax(f) = \emptyset$. Specific rules for decomposing constraints involving terms that are rooted by symbols obeying equational axioms, such as ACU and their combinations, are given below.

Concerning the rules $Solve_E$ and $Recover_E$, the main difference w.r.t. the corresponding syntactic generalization rules given in Section 3 is in the fact that the checks to the store consider the constraints modulo $E$: in the rules below, we write $t \overset{y}{\triangleq} t' \in^E S$ to express that there exists $s \overset{y}{\triangleq} s' \in S$ such that $t =_E s$ and $t' =_E s'$.

Finally, regarding the rule $Solve_E$, note that this rule cannot be applied to any constraint $t \overset{x}{\triangleq} s$ such that either $t$ or $s$ are rooted by a function symbol $f$ with $U_f \in ax(f)$. For function symbols with an identity element, a specially–tailored rule $Expand_U$ is given in Section 4.5 that gives us the opportunity to solve a constraint (conflict pair) $f(t_1,t_2) \overset{x}{\triangleq} s$, such that $root(s) \not\equiv f$, with a

**Decompose**$_E$

$$\frac{f \in (\Sigma \cup \mathcal{X}) \wedge ax(f) = \emptyset}{\langle f(t_1,\ldots,t_n) \overset{x}{\triangleq} f(t'_1,\ldots,t'_n) \wedge CT \mid S \mid \theta\rangle \Rightarrow \langle t_1 \overset{x_1}{\triangleq} t'_1 \wedge \ldots \wedge t_n \overset{x_n}{\triangleq} t'_n \wedge CT \mid S \mid \theta\sigma\rangle}$$

where $\sigma = \{x \mapsto f(x_1,\ldots,x_n)\}$, $x_1,\ldots,x_n$ are fresh variables, and $n \geq 0$

**Solve**$_E$

$$\frac{f = root(t) \wedge g = root(t') \wedge f \not\equiv g \wedge U_f \not\in ax(f) \wedge U_g \not\in ax(g) \wedge \nexists y : t \overset{y}{\triangleq} t' \in^E S}{\langle t \overset{x}{\triangleq} t' \wedge CT \mid S \mid \theta\rangle \Rightarrow \langle CT \mid S \wedge t \overset{x}{\triangleq} t' \mid \theta\rangle}$$

**Recover**$_E$

$$\frac{root(t) \not\equiv root(t') \wedge \exists y : t \overset{y}{\triangleq} t' \in^E S}{\langle t \overset{x}{\triangleq} t' \wedge CT \mid S \mid \theta\rangle \Rightarrow \langle CT \mid S \mid \theta\sigma\rangle}$$

where $\sigma = \{x \mapsto y\}$

**Fig. 3.** Basic rules for least general $E$–generalization

**Decompose**$_C$

$$\frac{C_f \in ax(f) \wedge A_f \not\in ax(f) \wedge i \in \{1,2\}}{\langle f(t_1,t_2) \overset{x}{=} f(s_1,s_2) \wedge CT \mid S \mid \theta\rangle \Rightarrow \langle t_1 \overset{x_1}{=} s_i \wedge t_2 \overset{x_2}{=} s_{(i \bmod 2)+1} \wedge CT \mid S \mid \theta\sigma\rangle}$$

where $\sigma = \{x \mapsto f(x_1,x_2)\}$, and $x_1, x_2$ are fresh variables

**Fig. 4.** Decomposition rule for a commutative function symbol $f$

generalization $f(y,z)$ more specific than $x$, by first introducing the constraint $f(t_1,t_2) \overset{x}{\triangleq} f(s,e)$.

### 4.2 Least General Generalization modulo $C$

In this section we extend the basic set of equational generalization rules by adding a specific inference rule $Decompose_C$, given in Figure 4, for dealing with commutativity function symbols. This inference rule replaces the syntactic decomposition inference rule for the case of a binary commutative symbol $f$, i.e., the two possible rearrangements of the terms $f(t_1,t_2)$ and $f(s_1,s_2)$ are considered. Just notice that this rule is (don't know) non-deterministic, hence all four combinations must be explored.

*Example 3.* Let $t = f(a,b)$ and $s = f(b,a)$ be two terms where $f$ is commutative, i.e., $C_f \in ax(f)$. By applying the rules $Solve_E$, $Recover_E$, and $Decompose_C$ above, we end in a terminal configuration $\langle\emptyset \mid S \mid \theta\rangle$, where $\theta = \{x \mapsto f(b,a), x_3 \mapsto b, x_4 \mapsto a\}$, thus we conclude that the lgg modulo $C$ of $t$ and $s$ is $x\theta = f(b,a)$.

### 4.3 Least General Generalization modulo $A$

In this section we provide a specific inference rule $Decompose_A$ for handling function symbols obeying the associativity axiom (but not the commutativity one). A specific set of rules for dealing with AC function symbols is given in the next subsection.

**Decompose$_A$**

$$\frac{A_f \in ax(f) \wedge C_f \notin ax(f) \wedge m \geq 2 \wedge n \geq m \wedge k \in \{1, \ldots, (n-m)+1\}}{\begin{array}{c}\langle f(t_1, \ldots, t_n) \stackrel{x}{=} f(s_1, \ldots, s_m) \wedge CT \mid S \mid \theta\rangle \Rightarrow \\ \langle f(t_1, \ldots, t_k) \stackrel{x_1}{=} s_1 \wedge f(t_{k+1}, \ldots, t_n) \stackrel{x_2}{=} f(s_2, \ldots, s_m) \wedge CT \mid S \mid \theta\sigma\rangle\end{array}}$$

where $\sigma = \{x \mapsto f(x_1, x_2)\}$, and $x_1, x_2$ are fresh variables

**Fig. 5.** Decomposition rule for an associative (non–commutative) function symbol $f$

The *Decompose$_A$* rule is given in Figure 5. We use flattened versions of the terms which use poly-variadic versions of the associative symbols, i.e., being $f$ an associative symbol,

$$flat(f(t_1, \ldots, f(s_1, \ldots, s_k), \ldots, t_n)) = flat(f(t_1, \ldots, s_1, \ldots, s_k, \ldots, t_n))$$

and, otherwise, $flat(f(t_1, \ldots, t_n)) = f(flat(t_1), \ldots, flat(t_n))$. Given an associative symbol $f$ and a term $f(t_1, \ldots, t_n)$ we call *alien $f$-terms* (or simply *alien terms*) to those terms among $t_1, \ldots, t_n$ that are not rooted by $f$. In the following, being $f$ an associative poly-varyadic symbol, $f(t)$ represents the term $t$ itself, since symbol $f$ needs at least two arguments. The inference rule of Figure 5 replaces the syntactic decomposition inference rule for the case of an associative function symbol $f$, where all *prefixes* of $t_1, \ldots, t_n$ and $s_1, \ldots, s_m$ are considered. Just notice that this rule is (don't know) non-deterministic, hence all possibilities must be explored.

Note that this is better than generating all terms in the corresponding equivalence class, as explained in Section 4, since we will eagerly stop in a constraint $t \stackrel{x}{=} f(t_1, \ldots, t_n)$ if $root(t) \not\equiv f$ without considering all the combinations in the equivalence class of $f(t_1, \ldots, t_n)$.

We give the rule *Decompose$_A$* for the case when, in the generalization problem $f(t_1, ..., t_n) \stackrel{x}{\triangleq} f(s_1, ..., s_m)$, we have that $n \geq m$. For the other way round, i.e., $n < m$, a similar rule would be needed, that we omit since it is perfectly analogous.

*Example 4.* Let $f(f(a, c), b)$ and $f(c, b)$ be two terms where $f$ is associative, i.e., $A_f \in ax(f)$. By applying the rules *Solve$_E$*, *Recover$_E$*, and *Decompose$_A$* above, we end in a terminal configuration $\langle \emptyset \mid S \mid \theta\rangle$, where $\theta = \{x \mapsto f(x_3, b), x_4 \mapsto b\}$, thus we compute that the lgg modulo $A$ of $t$ and $s$ is $f(x_3, b)$. The computation trace is shown in Figure 6.

### 4.4 Least General Generalization modulo *AC*

In this section we provide a specific inference rule *Decompose$_{AC}$* for handling function symbols obeying both the associativity and commutativity axioms. Note that we use again flattened versions of the terms, as in the associative case of Section 4.3. Actually, the new decomposition rule for the case AC is similar to the decompose inference rule for associative function symbols, except that

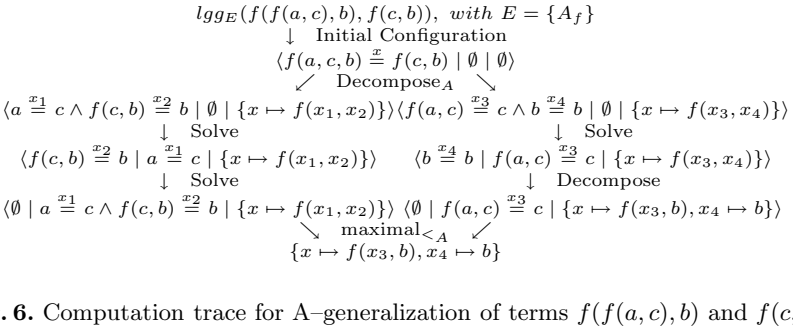



**Fig. 6.** Computation trace for A–generalization of terms $f(f(a,c),b)$ and $f(c,b)$)

**Decompose**$_{AC}$

$$\frac{\{A_f, C_f\} \subseteq ax(f) \wedge n \geq m \ \wedge \{i_1, \ldots, i_{m-1}\} \uplus \{i_m, \ldots, i_n\} = \{1, \ldots, n\}}{\begin{array}{c}\langle f(t_1, \ldots, t_n) \stackrel{x}{=} f(s_1, \ldots, s_m) \wedge C \mid S \mid \theta \rangle \Rightarrow \\ \langle t_{i_1} \stackrel{x_1}{=} s_1 \wedge \ldots \wedge t_{i_{m-1}} \stackrel{x_{m-1}}{=} s_{m-1} \wedge f(t_{i_m}, \ldots, t_{i_n}) \stackrel{x_m}{=} s_m \wedge C \mid S \mid \theta\sigma \rangle\end{array}}$$

where $\sigma = \{x \mapsto f(x_1, \ldots, x_m)\}$, and $x_1, \ldots, x_m$ are fresh variables

**Fig. 7.** Decomposition rule for an associative–commutative function symbol $f$

all permutations of $f(t_1, \ldots, t_n)$ and $f(s_1, \ldots, s_m)$ are considered. Just notice that this rule is (don't know) non-deterministic, hence all possibilities must be explored.

Similarly to the rule *Decompose*$_A$, we give the rule *Decompose*$_{AC}$ for the case when, in the generalization problem $f(t_1, ..., t_n) \stackrel{x}{\triangleq} f(s_1, ..., s_m)$, we have that $n \geq m$. For the other way round, i.e., $n < m$, a similar rule would be needed, that we omit since it is perfectly analogous. To simplify, we write $\{i_1, \ldots, i_k\} \uplus \{i_{k+1}, \ldots, i_n\} = \{1, \ldots, n\}$ to denote that the sequence $\{i_1, \ldots, i_n\}$ is a permutation of the sequence $\{1, \ldots, n\}$ and, given an element $k \in \{1, \ldots, n\}$, we split the sequence $\{i_1, \ldots, i_n\}$ in the two parts, $\{i_1, \ldots, i_k\}$ and $\{i_{k+1}, \ldots, i_n\}$.

*Example 5.* Let $t = f(a, f(a, b))$ and $s = f(f(b, b), a)$ be two terms where $f$ is associative and commutative, i.e., $\{A_f, C_f\} \subseteq ax(f)$. By applying the rules *Solve*$_E$, *Recover*$_E$, and *Decompose*$_{AC}$ above, we end in two terminal configurations whose respective substitution components are $\theta_1 = \{x \mapsto f(x_1, x_1, x_3), x_2 \mapsto x_1\}$ and $\theta_2 = \{x \mapsto f(x_4, a, b), x_5 \mapsto a, x_6 \mapsto b\}$, thus we compute that the lggs modulo $AC$ of $t$ and $s$ are $f(x_1, x_1, x_3)$ and $f(x_4, a, b)$. The corresponding computation trace is shown in Figure 8.

### 4.5 Least General Generalization modulo $U$

Finally, let us introduce the inference rule of Figure 9 for handling function symbols $f$ which have an identity element $e$. This rule considers the identity

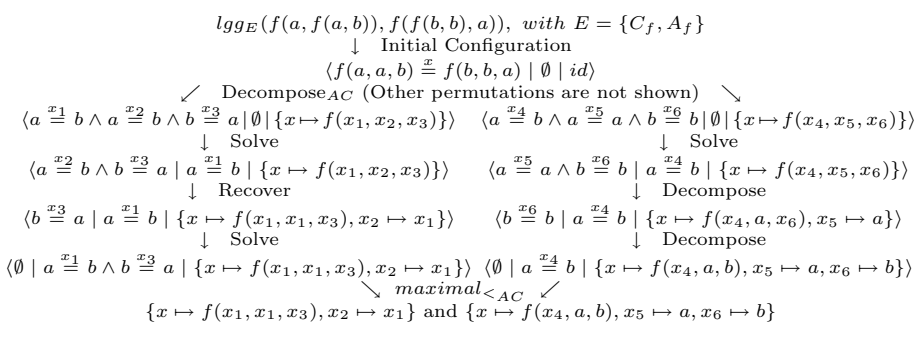



**Fig. 8.** Computation trace for AC–generalizations of terms $f(a,f(a,b))$ and $f(f(b,b),a)$

$$\textbf{Expand}_U\ \frac{U_f\in ax(f)\wedge root(t)\equiv f\wedge root(s)\not\equiv f\wedge s'\in\{f(e,s),f(s,e)\}}{\langle t\stackrel{x}{=}s\wedge CT\mid S\mid\theta\rangle\Rightarrow\langle t\stackrel{x}{=}s'\wedge CT\mid S\mid\theta\rangle}$$

**Fig. 9.** Inference rule for expanding function symbol $f$ with identity element $e$

axioms in a rather lazy or on-demand manner. The rule corresponds to the case when the root symbol $f$ of the term $t$ in the left–hand side of the constraint $t\stackrel{x}{=}s$ obeys the unity axioms. A companion rule for handling the case when the root symbol of the term $s$ in the right–hand side obeys the unity axiom is omitted, that is perfectly analogous.

*Example 6.* Let $t=f(a,b,c,d)$ and $s=f(a,c)$ be two terms where $\{A_f,C_f,U_f\}\subseteq ax(f)$. By applying the rules $Solve_E$, $Recover_E$, $Decompose_{AC}$, and $Expand_U$ above, we end in a terminal configuration $\langle\emptyset\mid S\mid\theta\rangle$, where $\theta=\{x\mapsto f(a,f(c,f(x_5,x_6))),\ x_1\mapsto a,x_2\mapsto f(c,f(x_5,x_6)),x_3\mapsto c,x_4\mapsto f(x_5,x_6)\}$, thus we compute that the lgg modulo $ACU$ of $t$ and $s$ is $f(a,c,x_5,x_6)$. The computation trace is shown in Figure 10.

### 4.6 A General ACU-Generalization Method

For the general case when different function symbols satisfying different associativity and/or commutativity and/or identity axioms are considered, we can use the rules above all together, with no need whatsoever for any changes or adaptations.

The key property of all the above inference rules is their *locality*: they are local to the given top function symbol in the left term (or right term in some cases) of the constraint they are acting upon, irrespective of what other function symbols and what other axioms may be present in the given signature $\Sigma$ and theory $E$. Such a locality means that these rules are *modular*, in the sense that they do not need to be changed or modified when new function symbols are added to the signature and new $A$, and/or $C$, and/or $U$ axioms are added to $E$. However, when new axioms are added to $E$, some rules that applied before (for example decomposition for an $f$ which before satisfied $ax(f)=\emptyset$, but now has $ax(f)\neq\emptyset$)

$$lgg_E(f(a,b,c,d), f(a,c)),\ with\ E = \{C_f, A_f, U_f\}$$
$$\downarrow \text{ Initial Configuration}$$
$$\langle f(a,b,c,d) \stackrel{x}{=} f(a,c) \mid \emptyset \mid id \rangle$$
$$\downarrow \text{ Decompose}_{AC} \text{ (Other permutations are not shown)}$$
$$\langle a \stackrel{x_1}{=} a \wedge f(b,c,d) \stackrel{x_2}{=} c \mid \emptyset \mid \{x \mapsto f(x_1,x_2)\} \rangle$$
$$\downarrow \text{ Decompose}$$
$$\langle f(b,c,d) \stackrel{x_2}{=} c \mid \emptyset \mid \{x \mapsto f(a,x_2), x_1 \mapsto a\} \rangle$$
$$\downarrow \text{ Expand}_U$$
$$\langle f(b,c,d) \stackrel{x_2}{=} f(c,e) \mid \emptyset \mid \{x \mapsto f(a,x_2), x_1 \mapsto a\} \rangle$$
$$\downarrow \text{ Decompose}_{AC} \text{ (Other permutations are not shown)}$$
$$\langle c \stackrel{x_3}{=} c \wedge f(b,d) \stackrel{x_4}{=} e \mid \emptyset \mid \{x \mapsto f(a,f(x_3,x_4)), x_1 \mapsto a, x_2 \mapsto f(x_3,x_4)\} \rangle$$
$$\downarrow \text{ Decompose}$$
$$\langle f(b,d) \stackrel{x_4}{=} e \mid \emptyset \mid \{x \mapsto f(a,f(c,x_4)), x_1 \mapsto a, x_2 \mapsto f(c,x_4), x_3 \mapsto c\} \rangle$$
$$\downarrow \text{ Expand}_U$$
$$\langle f(b,d) \stackrel{x_4}{=} f(e,e) \mid \emptyset \mid \{x \mapsto f(a,f(c,x_4)), x_1 \mapsto a, x_2 \mapsto f(c,x_4), x_3 \mapsto c\} \rangle$$
$$\downarrow \text{ Decompose}_{AC} \text{ (Other permutations are not shown)}$$
$$\langle b \stackrel{x_5}{=} e \wedge d \stackrel{x_6}{=} e \mid \emptyset \mid \{x \mapsto f(a,f(c,f(x_5,x_6))), x_1 \mapsto a, x_2 \mapsto f(c,f(x_5,x_6)), x_3 \mapsto c, x_4 \mapsto f(x_5,x_6)\} \rangle$$
$$\downarrow \text{ Solve}$$
$$\langle d \stackrel{x_6}{=} e \mid b \stackrel{x_5}{=} e \mid \{x \mapsto f(a,f(c,f(x_5,x_6))), x_1 \mapsto a, x_2 \mapsto f(c,f(x_5,x_6)), x_3 \mapsto c, x_4 \mapsto f(x_5,x_6)\} \rangle$$
$$\downarrow \text{ Solve}$$
$$\langle \emptyset \mid b \stackrel{x_5}{=} e \wedge d \stackrel{x_6}{=} e \mid \{x \mapsto f(a,f(c,f(x_5,x_6))), x_1 \mapsto a, x_2 \mapsto f(c,f(x_5,x_6)), x_3 \mapsto c, x_4 \mapsto f(x_5,x_6)\} \rangle$$
$$\downarrow \text{ maximal}_{<_{ACU}}$$
$$\{x \mapsto f(a,f(c,f(x_5,x_6))), x_1 \mapsto a, x_2 \mapsto f(c,f(x_5,x_6)), x_3 \mapsto c, x_4 \mapsto f(x_5,x_6)\}$$

**Fig. 10.** Computation trace for ACU–generalization of terms $f(a,b,c,d)$ and $f(a,c)$

may not apply, and, conversely, some rules that did not apply before now may apply (because new axioms are added to $f$). But *the rules themselves do not change!* They are the same and can be used to compute the set of lggs of two terms modulo *any* theory $E$ in the *parametric* family $\mathbb{E}$ of theories of the form $E = \bigcup_{f \in \Sigma} ax(f)$, where $ax(f) \subseteq \{A_f, C_f, U_f\}$. Termination of the algorithm is straightforward.

**Theorem 3.** *Every derivation stemming from an initial configuration* $\langle t \stackrel{x}{\triangleq} s \mid \emptyset \mid id \rangle$ *terminates.*

The correctness and completness of our algorithm is ensured by:

**Theorem 4.** *Given terms* $t$ *and* $s$*, an equational theory* $E \in \mathbb{E}$*, and a fresh variable* $x$*, then* $lgg_E(t,s) = maximal_{<_E}(\{x\theta \mid \langle t \stackrel{x}{\triangleq} s \mid \emptyset \mid id \rangle \Rightarrow^* \langle \emptyset \mid S \mid \theta \rangle\})$*, up to renaming.*

The ACU least general generalization algorithm presented here has been implemented in Maude [11], with a Web interface written in Java. The core of the tool contains about 200 lines of Maude code. The Web tool is publicly available together with a set of examples at the following url: `http://www.dsic.upv.es/users/elp/toolsMaude/Lgg2.html`

## 5 Conclusions and Future Work

We have presented a modular equational generalization algorithm that computes a minimal and complete set of least general generalizations for two terms

modulo any combination of associativity, commutativity and identity axioms for the binary symbols in the theory. Our algorithm is directly applicable to any untyped declarative language and reasoning systems. However, it would be highly desirable to support generalization modulo equational theories $(\Sigma, E)$ where $\Sigma$ is a typed signature such as for example an order–sorted signature, since a number of rule–based languages such as ASF+SDF [8], Elan [9], OBJ [18], CafeOBJ [14], and Maude [11] support order–sorted or many–sorted signatures. All existing generalization algorithms, with the exception of [27] and [1], assume an untyped setting. Moreover, the algorithm for generalization in the calculus of constructions of [27] cannot be used for order-sorted theories. In [3], we have developed an order–sorted generalization algorithm for the case where the set $E$ of axioms is empty. It would be very useful to combine the order–sorted and the $E$–generalization inference systems into a single calculus supporting both types and equational axioms. However, this combination seems to us non–trivial and is left for future work.

In our own work, we plan to use the above–mentioned order-sorted equational generalization algorithm as a key component of a narrowing-based partial evaluator (PE) for programs in order-sorted rule-based languages such as OBJ, CafeOBJ, and Maude. This will make available for such languages useful narrowing–driven PE techniques developed for the untyped setting in, e.g., [4,5]. We are also considering adding this generalization mechanism to an inductive theorem prover such a Maude's ITP [12] to support automatic conjecture of lemmas. This will provide a typed analogue of similar automatic lemma conjecture mechanisms in untyped inductive theorem provers such as Nqthm [10] and its ACL2 successor [20].

## References

1. Aït-Kaci, H., Sasaki, Y.: An Axiomatic Approach to Feature Term Generalization. In: Flach, P.A., De Raedt, L. (eds.) ECML 2001. LNCS, vol. 2167, pp. 1–12. Springer, Heidelberg (2001)
2. Alpuente, M., Escobar, S., Meseguer, J., Ojeda, P.: A modular Equational Generalization Algorithm. Technical Report DSIC-II/6/08, DSIC-UPV (2008)
3. Alpuente, M., Escobar, S., Meseguer, J., Ojeda, P.: Order–Sorted Generalization. Electr. Notes Theor. Comput. Sci. (to appear) (2008)
4. Alpuente, M., Falaschi, M., Vidal, G.: Partial evaluation of functional logic programs. ACM Trans. Program. Lang. Syst. 20(4), 768–844 (1998)
5. Alpuente, M., Lucas, S., Hanus, M., Vidal, G.: Specialization of functional logic programs based on needed narrowing. TPLP 5(3), 273–303 (2005)
6. Baader, F., Snyder, W.: Unification theory. In: Handbook of Automated Reasoning. Elsevier, Amsterdam (1999)
7. Belli, F., Jack, O.: Declarative paradigm of test coverage. Softw. Test., Verif. Reliab. 8(1), 15–47 (1998)
8. Bergstra, J.A., Heering, J., Klint, P.: Algebraic Specification. ACM Press, New York (1989)
9. Borovanský, P., Kirchner, C., Kirchner, H., Moreau, P.-E.: ELAN from a rewriting logic point of view. Theoretical Computer Science 285, 155–185 (2002)
10. Boyer, R., Moore, J.: A Computational Logic. Academic Press, London (1980)

11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. Springer, New York (2007)
12. Clavel, M., Palomino, M.: The ITP tool's manual. Universidad Complutense, Madrid (April 2005), http://maude.sip.ucm.es/itp/
13. Cortier, V., Delaune, S., Lafourcade, P.: A Survey of Algebraic Properties used in Cryptographic Protocols. Journal of Computer Security 14(1), 1–43 (2006)
14. Diaconescu, R., Futatsugi, K.: CafeOBJ Report. AMAST Series in Computing, vol. 6. World Scientific, Singapore (1998)
15. Escobar, S., Meseguer, J., Thati, P.: Narrowing and rewriting logic: from foundations to applications. Electr. Notes Theor. Comput. Sci. 177, 5–33 (2007)
16. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL Protocol Analyzer and its meta-logical properties. Theoretical Computer Science 367(1-2), 162–202 (2006)
17. Gallagher, J.P.: Tutorial on specialisation of logic programs. In: Proc. PEPM 1993, pp. 88–98. ACM, New York (1993)
18. Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.-P.: Introducing OBJ. In: Software Engineering with OBJ: Algebraic Specification in Action, pp. 3–167. Kluwer, Dordrecht (2000)
19. Huet, G.: Resolution d'Equations dans des Langages d'Order 1, 2,. . . ,$\omega$. PhD thesis, Univ. Paris VII (1976)
20. Kaufmann, M., Manolios, P., Moore, J.S.: Computer-Aided Reasoning: An Approach. Kluwer, Dordrecht (2000)
21. Lassez, J.-L., Maher, M.J., Marriott, K.: Unification Revisited. In: Minker, J. (ed.) Foundations of Deductive Databases and Logic Programming, pp. 587–625. Morgan Kaufmann, Los Altos (1988)
22. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
23. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
24. Mogensen, T.Æ.: Glossary for partial evaluation and related topics. Higher-Order and Symbolic Computation 13(4) (2000)
25. Muggleton, S.: Inductive Logic Programming: Issues, Results and the Challenge of Learning Language in Logic. Artif. Intell. 114(1-2), 283–296 (1999)
26. Østvold, B.: A functional reconstruction of anti-unification. Technical Report DART/04/04, Norwegian Computing Center (2004), http://publications.nr.no/nr-notat-dart-04-04.pdf
27. Pfenning, F.: Unification and anti-unification in the calculus of constructions. In: Proc. LICS 1991, pp. 74–85. IEEE Computer Society, Los Alamitos (1991)
28. Plotkin, G.D.: A note on inductive generalization. In: Machine Intelligence, vol. 5, pp. 153–163. Edinburgh University Press (1970)
29. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. 60-61, 17–139 (2004)
30. Popplestone, R.J.: An experiment in automatic induction. In: Machine Intelligence, vol. 5, pp. 203–215. Edinburgh University Press (1969)
31. Reynolds, J.: Transformational systems and the algebraic structure of atomic formulas. Machine Intelligence 5, 135–151 (1970)
32. Siekmann, J.H.: Unification Theory. Journal of Symbolic Computation 7, 207–274 (1989)
33. TeReSe (ed.): Term Rewriting Systems. Cambridge University Press, Cambridge (2003)

# A Transformational Approach to Polyvariant BTA of Higher-Order Functional Programs*

Gustavo Arroyo[1], J. Guadalupe Ramos[2], Salvador Tamarit[1], and Germán Vidal[1]

[1] DSIC, Technical University of Valencia, E-46022, Valencia, Spain
{garroyo,stamarit,gvidal}@dsic.upv.es
[2] Instituto Tecnológico de la Piedad, La Piedad, Michoacan, México
guadalupe@dsic.upv.es

**Abstract.** We introduce a *transformational* approach to improve the first stage of offline partial evaluation of functional programs, the so called binding-time analysis (BTA). For this purpose, we first introduce an improved defunctionalization algorithm that transforms higher-order functions into first-order ones, so that existing techniques for termination analysis and propagation of binding-times of first-order programs can be applied. Then, we define another transformation (tailored to defunctionalized programs) that allows us to get the accuracy of a polyvariant BTA from a monovariant BTA over the transformed program. Finally, we show a summary of experimental results that demonstrate the usefulness of our approach.

## 1 Introduction

Partial evaluation [14] aims at specializing programs w.r.t. part of their input data (the *static* data). Partial evaluation may proceed either online or offline. *Online* techniques implement a single, monolithic procedure that specializes the program while dynamically checking that the termination of the process is kept. *Offline* techniques, on the other hand, have two clearly separated phases. The aim of the first phase, the so called *binding-time analysis* (BTA), is basically the propagation of the static information provided by the user. A BTA should also ensure that the specialization process terminates; for this purpose, it often includes a termination analysis of the program. The output of this phase is an *annotated* program so that the second phase—the proper specialization—only needs to follow these annotations (and, thus, it runs faster than in the online approach).

Narrowing-driven partial evaluation [2] is a powerful technique for the specialization of functional (logic) programs based on the *narrowing* principle [21], a conservative extension of rewriting to deal with logic variables (i.e., *unknown*

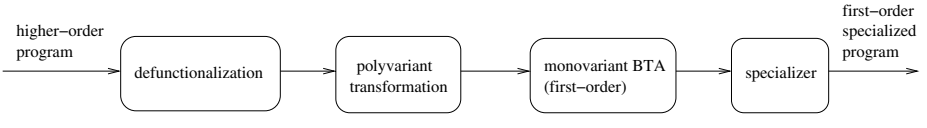
* This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008, by SES-ANUIES and by DGEST (México).

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 40–54, 2009.

information in our context). An offline approach to narrowing-driven partial evaluation has been introduced in [17]. In order to improve its accuracy, [6] adapts a *size-change analysis* [15] to the setting of narrowing. This analysis is then used to detect potential sources of non-termination, so that the arguments that may introduce infinite loops at partial evaluation time are annotated to be *generalized* (i.e., replaced by fresh variables).

Unfortunately, the size-change analysis of [6] and the associated BTA suffer from several limitations. Firstly, the size-change analysis is only defined for *first-order* functional programs, thus limiting its applicability. And, secondly, the associated binding-time analysis is *monovariant*, i.e., a single sequence of binding-times[1] is associated to the arguments of a given function and, thus, all calls to the same function are treated in the same way, which implies a considerable loss of accuracy.

In this work, we present a transformational approach to overcome the above drawbacks. Basically, we first transform the original higher-order program by *defunctionalization* [18]. In particular, we introduce an extension of previous defunctionalization techniques (like [4,12]) that is specially tailored to improve the accuracy of the size-change analysis. Then, we introduce a source-to-source transformation that aims at improving the accuracy of both the size-change analysis and the associated BTA by making explicit the binding-times of every argument of a function. In this way, every function call with different binding-times can be treated differently. Thanks to this transformation, one can get the same accuracy by using a monovariant BTA over the transformed program as by using a *polyvariant* BTA (where several binding-times can be associated to a given function) over the original program.




The paper is organized as follows. After some preliminaries, Sect. 3 introduces our defunctionalization technique in a stepwise manner. Then, Sect. 4 presents a polyvariant transformation that makes explicit the binding-times of functions. Section 5 shows a summary of the experimental results conducted to evaluate the usefulness of the approach. Finally, Sect. 6 discusses some related work and concludes.

## 2 Preliminaries

Term rewriting [7] offers an appropriate framework to model the first-order component of many functional (logic) programming languages. Therefore, in the remainder of this paper we follow the standard framework of term rewriting for developing our results.

[1] We consider the usual binding-times: static (definitely known at partial evaluation time) and dynamic (possibly unknown at partial evaluation time).

A *term rewriting system* (TRS for short) is a set of rewrite rules $l = r$ such that $l$ is a nonvariable term and $r$ is a term whose variables appear in $l$; terms $l$ and $r$ are called the left-hand side and the right-hand side of the rule, respectively. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}$ are the root symbols of the left-hand sides of the rules and the *constructors* are $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. We restrict ourselves to finite signatures and TRSs. We denote the domain of terms and *constructor terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$ and $\mathcal{T}(\mathcal{C}, \mathcal{V})$, respectively, where $\mathcal{V}$ is a set of variables with $\mathcal{F} \cap \mathcal{V} = \emptyset$.

A TRS $\mathcal{R}$ is *constructor-based* if the left-hand sides of its rules have the form $f(s_1, \ldots, s_n)$ where $s_i$ are constructor terms, i.e., $s_i \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, for all $i = 1, \ldots, n$. The set of variables appearing in a term $t$ is denoted by $Var(t)$. A term $t$ is *linear* if every variable of $\mathcal{V}$ occurs at most once in $t$. $\mathcal{R}$ is left-linear if $l$ is linear for all rules $l = r \in \mathcal{R}$. The *definition* of $f$ in $\mathcal{R}$ is the set of rules in $\mathcal{R}$ whose root symbol in the left-hand side is $f$.

The root symbol of a term $t$ is denoted by $root(t)$. A term $t$ is *operation-rooted* (resp. *constructor-rooted*) if $root(t) \in \mathcal{D}$ (resp. $root(t) \in \mathcal{C}$). As it is common practice, a *position* $p$ in a term $t$ is represented by a sequence of natural numbers, where $\epsilon$ denotes the root position. Positions are used to address the nodes of a term viewed as a tree: $t|_p$ denotes the *subterm* of $t$ at position $p$ and $t[s]_p$ denotes the result of *replacing the subterm* $t|_p$ by the term $s$. A term $t$ is *ground* if $Var(t) = \emptyset$. A term $t$ is a *variant* of term $t'$ if they are equal modulo variable renaming. A *substitution* $\sigma$ is a mapping from variables to terms such that its domain $\mathcal{D}om(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is finite. The identity substitution is denoted by $id$. Term $t'$ is an *instance* of term $t$ if there is a substitution $\sigma$ with $t' = \sigma(t)$. A *unifier* of two terms $s$ and $t$ is a substitution $\sigma$ with $\sigma(s) = \sigma(t)$. In the following, we write $\overline{o_n}$ for the *sequence of objects* $o_1, \ldots, o_n$.

In the remainder of this paper, we consider *inductively sequential* TRSs [3] as *programs*, a subclass of left-linear constructor-based TRSs. Essentially, a TRS is inductively sequential when all its operations are defined by rewrite rules that, recursively, make on their arguments a case distinction analogous to a data type (or structural) induction, i.e., typical functional programs.

## 3 Defunctionalization

In this section, we introduce a stepwise transformation that takes a higher-order program and returns a first-order program (a term rewrite system). In contrast to standard defunctionalization algorithms, we perform a restricted form of variable instantiation and unfolding (see step 2 below) so that more higher-order information is made explicit. Although it may increase code size, the next steps of the BTA—size-change analysis and propagation of binding-times—can exploit it for producing more accurate results; hopefully, this code size increase is then removed in the specialization phase (see the benchmarks in Sect. 5).

We present our improved defunctionalization technique in a stepwise manner:

1. The first step makes both applications and partial function calls explicit.

2. Then, the second step instantiates functional variables with all possible partial function calls.
3. Finally, if after reducing applications[2] as much as possible the program still contains some applications, the third step adds an appropriate definition for the application function.

### 3.1 Making Applications and Partial Calls Explicit

First, we make every application of the higher-order program explicit by using the fresh function apply. Also, we distinguish partial applications from total functions. In particular, partial applications are represented by means of the fresh constructor symbol partcall. Total calls are denoted in the usual way, e.g., $f(\overline{t_n})$ for some defined function symbol $f/n$. A partial call is denoted by $\mathsf{partcall}(f, k, \overline{t_m})$ where $f/n$ is a defined function symbol, $0 < k \leq n$, and $m + k = n$, i.e., $\overline{t_m}$ are the first $m$ arguments of $f/n$ but there are still $k$ missing arguments (if $k = n$, then the partial application has no arguments, i.e., we have $\mathsf{partcall}(f, n)$).[3] For simplicity, in the following we consider that partcall is a variadic function; nevertheless, one can formalize it using a function with three arguments so that the third argument is a (possibly empty) list with the already applied arguments.

Once all applications are made explicit with apply and partcall, we apply the following transformation to the right-hand sides as much as possible:

$$\mathsf{apply}(\mathsf{partcall}(f, k, \overline{t_m}), t_{m+1}) = \begin{cases} f(\overline{t_{m+1}}) & \text{if } k = 1 \\ \mathsf{partcall}(f, k-1, \overline{t_{m+1}}) & \text{if } k > 1 \end{cases} \qquad (*)$$

This is useful to avoid unnecessary applications in the defunctionalized program when enough information is available to reduce them statically.

In the following, we assume that every program contains an entry function, called main, which is defined by a single rule of the form $(\mathsf{main}\ x_1\ \ldots\ x_n\ =\ r)$, with $x_1, \ldots, x_n \in \mathcal{V}$ different variables, and that the program contains no call to this distinguished function. Furthermore, we consider that run time calls to main have only constructor terms (i.e., values) as arguments; this is required to avoid the introduction of higher-order expressions which are not in the program.

*Example 1.* Consider the following higher-order program $\mathcal{R}_1$ (as it is common practice, we use a curried notation for higher-order programs):

$$\begin{array}{ll} \mathsf{main}\ xs = map\ inc\ xs & map\ f\ [\,] \quad\ = [\,] \\ inc\ x \quad = Succ\ x & map\ f\ (x : xs) = f\ x : map\ f\ xs \end{array}$$

[2] Basically, every expression of the form $\mathsf{apply}(\mathsf{partcall}(\ldots), \ldots)$ can be reduced, where apply is the application function and partcall denotes a partial function call.

[3] A similar representation is used in FlatCurry, the intermediate representation of the functional logic programming language Curry [11].

where natural numbers are built from $Z$ and $Succ$ and lists are built from [] and ":". First, we make all applications and partial calls explicit:

$$\begin{array}{ll} \mathsf{main}(xs) & = \mathsf{apply}(\mathsf{apply}(\mathsf{partcall}(map, 2), \mathsf{partcall}(inc, 1)), xs) \\ map(f, [\,]) & = [\,] \\ map(f, x : xs) & = \mathsf{apply}(f, x) : \mathsf{apply}(\mathsf{apply}(\mathsf{partcall}(map, 2), f), xs) \\ inc(x) & = Succ(x) \end{array}$$

Then, we reduce all calls to apply with a partial call as a first argument using the transformation $(*)$ above so that we get the transformed program $\mathcal{R}_2$:

$$\begin{array}{ll} \mathsf{main}(xs) = map(\mathsf{partcall}(inc, 1), xs) & map(f, [\,]) = [\,] \\ inc(x) = Succ(x) & map(f, x : xs) = \mathsf{apply}(f, x) : map(f, xs) \end{array}$$

### 3.2 Instantiation of Functional Variables

In the next step, we instantiate the functional variables of the program so that some applications can hopefully be reduced. This is the main difference w.r.t. previous defunctionalization algorithms. In the following, we say that a variable is a *functional* variable if it can be bound (at run time) to a partial call. Now, we replace every functional variable by all possible partial applications.

Let $\text{PCALLS}_{\mathcal{R}}$ be the set of function symbols that appear in the partcall's of $\mathcal{R}$, i.e.,

$$\text{PCALLS}_{\mathcal{R}} = \{f/n \mid \mathsf{partcall}(f, k, t_1, \ldots, t_m) \text{ occurs in } \mathcal{R}, \text{ with } k + m = n\}$$

Then, for each function $f/n \in \text{PCALLS}_{\mathcal{R}}$ with type[4]

$$\tau_1 \mapsto \ldots \mapsto \tau_n \mapsto \ldots \mapsto \tau_k \mapsto \tau_{k+1} \qquad (n \leq k)$$

we replace each rule $l[x]_p = r$ where $x$ is a functional variable of type

$$\tau'_j \mapsto \ldots \mapsto \tau'_k \mapsto \tau'_{k+1}$$

and $1 \leq j \leq n$ (so that some argument is still missing), by the instances

$$\sigma(l[x]_p = r) \qquad \text{where } \sigma = \{x \mapsto \mathsf{partcall}(f, n - j + 1, x_1, \ldots, x_{j-1})\}$$

with $x_1, \ldots, x_{j-1}$ different fresh variables (if $j = 1$, no argument is added to the partial call). Clearly, we refer above to the types inferred in the original higher-order program. Nevertheless, if no type information is available, one could just instantiate the rules with all possible partial calls; this might introduce some useless rules but would be safe. For instance, consider the rule

$$f(x, xs) \;=\; map(x, xs)$$

[4] Observe that $n < k$ implies that function $f$ returns a functional value.

where $x$ is a functional variable of type $I\!N \mapsto I\!N$. Given the function $sum/2 \in$ $\text{PCALLS}_{\mathcal{R}}$ with type $I\!N \mapsto I\!N \mapsto I\!N$, we replace the rule above by the following instance:

$$f(\mathsf{partcall}(sum, 1, n), xs) \;=\; map(\mathsf{partcall}(sum, 1, n), xs)$$

The instantiation of rules is applied repeatedly until no rule with a functional variable appears in the program.[5] Then, as in the previous step, we apply the transformation $(*)$ above as much as possible to the right-hand sides of the program. The following example illustrates this instantiation process.

*Example 2.* Consider again the transformed program $\mathcal{R}_2$ of Example 1. We have $\text{PCALLS}_{\mathcal{R}_2} = \{inc/1\}$. There is only one functional variable $f$ (with type $\tau_1 \mapsto \tau_2$) in the rules defining $map$, hence we produce the following instantiated rules:

$$\begin{array}{l} map(\mathsf{partcall}(inc, 1), [\,]) \quad = [\,] \\ map(\mathsf{partcall}(inc, 1), x : xs) = \mathsf{apply}(\mathsf{partcall}(inc, 1), x) : map(\mathsf{partcall}(inc, 1), xs) \end{array}$$

Now, by reducing all calls to apply with a partcall as a first argument, we get the transformed program $\mathcal{R}_3$:

$$\begin{array}{ll} \mathsf{main}(xs) & = map(\mathsf{partcall}(inc, 1), xs) \\ map(\mathsf{partcall}(inc, 1), [\,]) & = [\,] \\ map(\mathsf{partcall}(inc, 1), x : xs) & = inc(x) : map(\mathsf{partcall}(inc, 1), xs) \\ inc(x) & = Succ(x) \end{array}$$

Note that $map(\mathsf{partcall}(inc, 1), \ldots)$ should be understood as a fresh function, e.g., we could rewrite the program as follows:

$$\begin{array}{llll} \mathsf{main}(xs) = map_{inc}(xs) & & map_{inc}([\,]) & = [\,] \\ inc(x) \quad = Succ(x) & & map_{inc}(x : xs) & = inc(x) : map_{inc}(xs) \end{array}$$

Observe that no call to apply occurs in the final program and, thus, there is no need to add a definition for apply (i.e., the next step would not be necessary for this example). Determining the class of programs for which we can guarantee that no occurrence of apply appears in the transformed programs is an interesting subject for future work.

Let us note that this step of the transformation is safe since main can only be called with constructor terms as arguments. Otherwise, functional variables should be instantiated with all possible partial applications and not only those in $\text{PCALLS}_{\mathcal{R}}$. On the other hand, not all instantiations of functional variables will be *reachable* from main. The use of a *closure analysis* may improve the accuracy of the transformed program (but it will also add a significant time overhead in the defunctionalization process).

[5] Note that a function may have several functional arguments and, thus, we could apply the instantiation process to the instantiations of a rule previously considered.

### 3.3 Adding an Explicit Definition of apply

In contrast to standard defunctionalization techniques (like [4,12]), the transformation process so far may produce a first-order program with no occurrences of function apply in many common cases (as in the previous example).

In other cases, however, some occurrences of apply remain in the transformed program and a proper definition of apply should be added. This is the case, e.g., when there is a call to apply with a function call as a first argument. In this case, the value of the partial call will not be known until run time and, thus, we add the following sequence of rules:

$$\begin{array}{l}\mathsf{apply}(\mathsf{partcall}(f,n),x_1) = \mathsf{partcall}(f,n-1,x_1)\\ \mathsf{apply}(\mathsf{partcall}(f,n-1,x_1),x_2) = \mathsf{partcall}(f,n-2,x_1,x_2)\\ \ldots\\ \mathsf{apply}(\mathsf{partcall}(f,1,x_1,\ldots,x_{n-1}),x_n) = f(x_1,\ldots,x_n)\end{array}$$

for each function symbol $f/n \in \text{PCALLS}_{\mathcal{R}}$.

Our defunctionalization process can be effectively applied not only to programs using simple constructs such as $(map\ f\ \ldots)$ but also to programs that make essential use of higher-order features, as the following example illustrates.

*Example 3.* Consider the following higher-order program from [20]:

$$\begin{array}{ll}\mathsf{main}\ x\ y = f\ x\ y & \\ g\ r\ a = r\ (r\ a) & f\ Z = inc\\ inc\ n = Succ\ n & f\ (Succ\ n) = g\ (f\ n)\end{array}$$

where natural numbers are built from $Z$ and $Succ$. The first step of the defunctionalization process returns

$$\begin{array}{ll}\mathsf{main}(x,y) = \mathsf{apply}(f(x),y) & \\ g(r,a) = \mathsf{apply}(r,\mathsf{apply}(r,a)) & f(Z) = \mathsf{partcall}(inc,1)\\ inc(n) = Succ(n) & f(Succ(n)) = \mathsf{partcall}(g,1,f(n))\end{array}$$

Here, $\text{PCALLS}_{\mathcal{R}} = \{inc/1, g/2\}$. We only have a functional variable $r$ in the rule defining function $g$ (with associated type $\mathbb{N} \mapsto \mathbb{N}$) and, therefore, the following instances of the rules defining function $g$ are added:

$$\begin{array}{l}g(\mathsf{partcall}(inc,1),a) = \mathsf{apply}(\mathsf{partcall}(inc,1),\mathsf{apply}(\mathsf{partcall}(inc,1),a))\\ g(\mathsf{partcall}(g,1,x),a) = \mathsf{apply}(\mathsf{partcall}(g,1,x),\mathsf{apply}(\mathsf{partcall}(g,1,x),a))\end{array}$$

By reducing all calls to apply with a partcall as a first argument, we get

$$\begin{array}{ll}\mathsf{main}(x,y) = \mathsf{apply}(f(x),y) & f(Z) = \mathsf{partcall}(inc,1)\\ g(\mathsf{partcall}(inc,1),a) = inc(inc(a)) & f(Succ(n)) = \mathsf{partcall}(g,1,f(n))\\ g(\mathsf{partcall}(g,1,x),a) = g(x,g(x,a)) & inc(n) = Succ(n)\end{array}$$

Finally, since an occurrence of function apply remains in the program, we add the following rules:

$$\begin{array}{l}\mathsf{apply}(\mathsf{partcall}(inc,1),x) = inc(x)\\ \mathsf{apply}(\mathsf{partcall}(g,2),x) = \mathsf{partcall}(g,1,x)\\ \mathsf{apply}(\mathsf{partcall}(g,1,x),y) = g(x,y)\end{array}$$

The correctness of our defunctionalization transformation is an easy extension of that in [4,12] by considering that function apply is strict in its first argument and, thus, our main extension, the instantiation of functional variables, is safe.

Note also that our approach is also safe at partial evaluation time where missing information (in the form of logical variables) might appear since the evaluation of higher-order calls containing free variables as functions is not allowed in current implementations of narrowing (i.e., such calls are *suspended* to avoid the use of higher-order unification [13]).

Regarding the code size increase due to our defunctionalization algorithm, the fact that it makes more higher-order information explicit comes at a cost: in the worst case, the source program can grow exponentially in the number of functions (e.g., when the program contains partial calls to all defined functions). Nevertheless, this case will happen only rarely and thus the code size increase is generally reasonable. Furthermore, the subsequent specialization phase is usually able to reduce the code (see Sect. 5).

## 4 Polyvariant Transformation

In this section, we introduce a source-to-source transformation that, given a program $\mathcal{R}$, returns a new program $\mathcal{R}'$ that is semantically equivalent to $\mathcal{R}$ but can be more accurately analyzed. Basically, our aim is to get the same information from a monovariant BTA over the transformed program $\mathcal{R}'$ as from a polyvariant BTA over the original program $\mathcal{R}$.

Intuitively speaking, we make a copy of each function definition for every call with different binding-times for its arguments. For simplicity, we only consider the basic *binding-times* S (static, known value) and D (dynamic, possibly unknown value). The least upper bound over binding-times is defined as follows:

$$\mathrm{S} \sqcup \mathrm{S} = \mathrm{S} \qquad \mathrm{S} \sqcup \mathrm{D} = \mathrm{D} \qquad \mathrm{D} \sqcup \mathrm{S} = \mathrm{D} \qquad \mathrm{D} \sqcup \mathrm{D} = \mathrm{D}$$

The least upper bound operation can be extended to sequences of binding-times in the natural way, e.g.,

$$\mathrm{SDS} \sqcup \mathrm{SSD} = \mathrm{SDD} \qquad \mathrm{SDS} \sqcup \mathrm{DSD} = \mathrm{DDD} \qquad \mathrm{SDS} \sqcup \mathrm{DSS} = \mathrm{DDS}$$

A binding-time *environment* is a substitution mapping variables to binding-times. We will use the following auxiliary function $B_e$ (adapted from [14]) for computing the binding-time of an expression:

$$\begin{array}{ll} B_e[[x]]\rho & = \rho(x) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{if } x \in \mathcal{V}) \\ B_e[[\mathsf{h}(t_1,\ldots,t_n)]]\rho & = B_e[[t_1]]\rho \sqcup \ldots \sqcup B_e[[t_n]]\rho \qquad\qquad (\text{if } \mathsf{h} \in \mathcal{C} \cup \mathcal{D}) \end{array}$$

where $\rho$ denotes a binding-time environment. Roughly speaking, an expression $(B_e[[t]]\rho)$ returns S if $t$ contains no variable which is bound to D in $\rho$, and D otherwise.

$$
\begin{array}{l}
poly\_trans(\{\,\}) = \{\,\} \\
poly\_trans(\{R\} \cup \mathcal{R}) = poly\_trans(\mathcal{R}) \\
\qquad \cup \left\{ \begin{array}{ll}
\{f_{\overline{b_n}}(\overline{t_n}) = pt(r, bte(f(\overline{t_n}), \overline{b_n})) \mid \overline{b_n} \in BT^n\} & \text{if } R = (f(\overline{t_n}) = r) \\
\{\mathsf{apply}_{b_n}(\mathsf{partcall}(f_{\overline{b_{n-1}}}, k, \overline{x_{n-1}}), x_n) = \mathsf{partcall}(f_{\overline{b_n}}, k-1, \overline{x_n}) \mid \overline{b_n} \in BT^n\} & \\
\qquad \text{if } R = (\mathsf{apply}(\mathsf{partcall}(f, k, \overline{x_{n-1}}), x_n) = \mathsf{partcall}(f, k-1, \overline{x_n})) & \\
\{\mathsf{apply}_{b_n}(\mathsf{partcall}(f_{\overline{b_{n-1}}}, k, \overline{x_{n-1}}), x_n) = f_{\overline{b_n}}(\overline{x_n}) \mid \overline{b_n} \in BT^n\} & \\
\qquad \text{if } R = (\mathsf{apply}(\mathsf{partcall}(f, k, \overline{x_{n-1}}), x_n) = f(\overline{x_n})) &
\end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
pc(\{\,\}) = \{\,\} \\
pc(\{R\} \cup \mathcal{R}) = \left\{ \begin{array}{ll}
pc(\{f(t_1, \ldots, \mathsf{partcall}(g_{\overline{b_m}}, k, \overline{t_m}), \ldots, t_n) = r \mid \overline{b_m} \in BT^m\} \cup \mathcal{R}) & \\
\qquad \text{if } R = (f(\overline{t_n}) = r),\ t_i = \mathsf{partcall}(g, k, \overline{t_m}),\ i \in \{1, \ldots, n\} & \\
\{R\} \cup pc(\mathcal{R}) & \text{otherwise}
\end{array} \right.
\end{array}
$$

$$
pt(t, \rho) = \left\{ \begin{array}{ll}
t & \text{if } t \in \mathcal{V} \\
c(\overline{pt(t_n, \rho)}) & \text{if } t = c(\overline{t_n}),\ c \in \mathcal{C} \\
f_{\overline{b_n}}(\overline{pt(t_n, \rho)}) & \text{if } t = f(\overline{t_n}),\ f \in \mathcal{D},\ B_e[\![t_i]\!]\rho = b_i,\ i = 1, \ldots, n \\
\mathsf{partcall}(f_{\overline{b_n}}, k, \overline{pt(t_n, \rho)}) & \text{if } t = \mathsf{partcall}(f, k, \overline{t_n}),\ B_e[\![t_i]\!]\rho = b_i,\ i = 1, \ldots, n \\
\mathsf{apply}_{b_2}(pt(t_1, \rho), pt(t_2, \rho)) & \text{if } t = \mathsf{apply}(t_1, t_2),\ B_e[\![t_i]\!]\rho = b_i,\ i = 1, 2
\end{array} \right.
$$

**Fig. 1.** Polyvariant transformation: functions *poly_trans* and *pt*

Given a linear term $f(\overline{t_n})$ (usually the left-hand side of a rule), and a sequence of binding-times $\overline{b_n}$ for $f$, the associated binding-time environment, $bte(f(\overline{t_n}), \overline{b_n})$, is defined as follows:

$$bte(f(\overline{t_n}), \overline{b_n}) = \{x \mapsto b_1 \mid x \in \mathcal{V}ar(t_1)\} \cup \ldots \cup \{x \mapsto b_n \mid x \in \mathcal{V}ar(t_n)\}$$

**Definition 1 (polyvariant transformation).** *Let $\mathcal{R}$ be a program and $\overline{b_n}$ be a sequence of binding-times for* main$/n$*. The polyvariant transformation of $\mathcal{R}$ w.r.t. $\overline{b_n}$, denoted by $\mathcal{R}^{\overline{b_n}}_{poly}$, is computed as follows:*

$$
\begin{array}{l}
\mathcal{R}^{\overline{b_n}}_{poly} = \{\ \mathsf{main}(\overline{x_n}) = pt(r, bte(\mathsf{main}(\overline{x_n}), \overline{b_n})) \mid \mathsf{main}(\overline{x_n}) = r \in \mathcal{R}\ \} \\
\qquad \cup\ poly\_trans(pc(\mathcal{R} \setminus \{\mathsf{main}(\overline{x_n}) = r\}))
\end{array}
$$

*where the auxiliary functions poly_trans, pc and pt are defined in Fig. 1. Here, we denote by $BT^n$ the set of all possible sequences of $n$ binding-times.*

Intuitively, the polyvariant transformation proceeds as follows:

- First, the left-hand side of function main is not labeled since there is no call to main in the program. The right-hand side is transformed as any other user-defined function using *pt* (see below).

– For the program rules (i.e., rules defining functions different from apply), we first label the occurrences of partcall in the left-hand sides using auxiliary function $pc$.[6] Observe that the first case of the definition of $pc$ includes the transformed rule in the next recursive call since the left-hand side may contain several partcall arguments; in the second case, when no occurrence of partcall remains, the rule is deleted from the recursive call.

  Then, we replace the resulting rules by a number of copies labeled with all possible sequences of binding-times, whose right-hand sides are then transformed using function $pt$. Here, we could restrict the possible binding-times to those binding-times that are safe approximations of the arguments $\overline{t_m}$ of the partial call. However, we keep the current formulation for simplicity.
– Rules defining apply are transformed so that the binding-times of the partial function and the new argument are made explicit. Observe that we label the function symbol inside a partial call but not the partial call itself. Also, apply is just labeled with the binding-time of their second argument; the binding-time of the first argument is not needed since the binding-times labeling the function inside the corresponding partial call already contains more accurate information.
– Function $pt$ takes a term and a binding-time environment and proceeds as follows:
  - Variables and constructor symbols are left untouched.
  - Function calls are labeled with the binding-times of their arguments according to the current binding-time environment. Function symbols in partial calls are also labeled in the same way.
  - Applications and partial calls are labeled as in function $poly\_trans$.

Observe that labeling functions with all possible sequences of binding-times produces usually a significant increase of code size. Clearly, one could perform a pre-processing analysis to determine the *call patterns* $f(\overline{b'_m})$ that may occur from the initial call to $\mathsf{main}(\overline{b_n})$. This approach, however, will involve a similar complexity as constructing the higher-order call graph of [20]. Here, we preferred to trade time complexity for space complexity. Furthermore, many of these copies are dead code and will be easily removed in the partial evaluation stage (see Sect. 5).

*Example 4.* Consider the defunctionalized program $\mathcal{R}$ of Example 3:

$$\begin{array}{ll}
\mathsf{main}(x,y) = \mathsf{apply}(f(x),y) & f(Z) = \mathsf{partcall}(inc,1) \\
g(\mathsf{partcall}(inc,1),a) = inc(inc(a)) & f(Succ(n)) = \mathsf{partcall}(g,1,f(n)) \\
g(\mathsf{partcall}(g,1,x),a) = g(x,g(x,a)) & inc(n) = Succ(n) \\
\mathsf{apply}(\mathsf{partcall}(g,2),x) = \mathsf{partcall}(g,1,x) & \mathsf{apply}(\mathsf{partcall}(inc,1),x) = inc(x) \\
\mathsf{apply}(\mathsf{partcall}(g,1,x),y) = g(x,y) &
\end{array}$$

[6] For clarity, we assumed that all occurrences of partcall appear at the top level of arguments, i.e., either the argument $t_i$ has the form $\mathsf{partcall}(\ldots)$ or it contains no occurrences of partcall.

Given the initial binding-times SD, our polyvariant transformation produces the following program $\mathcal{R}^{\text{SD}}_{poly}$:[7]

$$\mathsf{main}(x, y) = \mathsf{apply}_{\text{D}}(f_{\text{S}}(x), y)$$

$$f_{\text{S}}(Z) = \mathsf{partcall}(inc, 1)$$
$$f_{\text{S}}(Succ(n)) = \mathsf{partcall}(g_{\text{S}}, 1, f_{\text{S}}(n))$$

$$inc_{\text{S}}(n) = Succ(n)$$
$$inc_{\text{D}}(n) = Succ(n)$$

$$g_{\text{SD}}(\mathsf{partcall}(inc, 1), a) = inc_{\text{D}}(inc_{\text{D}}(a))$$
$$g_{\text{SD}}(\mathsf{partcall}(g_{\text{S}}, 1, x), a) = g_{\text{SD}}(x, g_{\text{SD}}(x, a))$$

$$\mathsf{apply}_{\text{D}}(\mathsf{partcall}(inc, 1), x) = inc_{\text{D}}(x)$$
$$\mathsf{apply}_{\text{D}}(\mathsf{partcall}(g_{\text{S}}, 1, x), y) = g_{\text{SD}}(x, y)$$

The next section presents a summary of an experimental evaluation conducted with a prototype implementation of the partial evaluation.

## 5 The Transformation in Practice

In this section, we present a summary of our progress on the development of a partial evaluator that follows the ideas presented so far. The undertaken implementation follows these directions:

- The system accepts higher-order programs which are first transformed using the techniques of Sect. 3 (defunctionalization) and Sect. 4 (polyvariant transformation).
- Then, the standard size-change analysis of [6] (for first-order programs) is applied to the transformed program.
- Finally, we annotate the program using the output of the size-change analysis and apply the specialization phase of the existing offline partial evaluator [17,6]. We note that no propagation of binding-times is required here[8] since this information is already explicit in every function call thanks to the polyvariant transformation.

Table 1 shows the effectiveness of our transformation over the following examples: `ack`, the well known Ackermann's function, which is specialized for a given first argument; `bulyonkov`, a slight extension of the running example in [9]; `combinatorial`, a simple program including the computation of combinatorials; `changeargs`, another variation of the running example in [9]; `dfib`, a higher-order example that uses the well-known Fibonacci's function; `dmap`, a

[7] Actually, the transformation produces some more (useless) rules that we do not show for clarity. Note also that, according to our technique, the occurrence of *inc* in the expression $\mathsf{partcall}(inc, 1)$ should be labeled with an empty sequence of binding-times. However, for simplicity, we write just *inc*.

[8] In the original scheme, the binding-time of every function argument is required in order to identify *static* loops that can be safely unfolded.

**Table 1.** Benchmark results (run times, milliseconds)

| benchmark | original | specialized | | poly specialized | |
|---|---|---|---|---|---|
| | run time | run time | speedup | run time | speedup |
| ack | 1526 | 507 | 3.01 | 522 | 2.92 |
| bulyonkov | 559 | 727 | 0.77 | 402 | 1.39 |
| combinatorial | 991 | 887 | 1.12 | 612 | 1.62 |
| changeargs | 772 | 1157 | 0.67 | 478 | 1.62 |
| dfib (HO) | 326 | 294 | 1.11 | 95 | 3.43 |
| dmap (HO) | 905 | 427 | 2.12 | 885 | 1.02 |
| **Average** | **760** | **602** | **1.26** | **416** | **1.83** |

**Table 2.** Benchmark results (code size, bytes)

| benchmark | original | specialized | | poly specialized | |
|---|---|---|---|---|---|
| | size | size | variation | size | variation |
| ack | 951 | 3052 | 3.21 | 4168 | 4.38 |
| bulyonkov | 2250 | 3670 | 1.63 | 2440 | 1.08 |
| combinatorial | 2486 | 3546 | 1.43 | 6340 | 2.55 |
| changeargs | 3908 | 5335 | 1.37 | 5599 | 1.43 |
| dfib (HO) | 2911 | 4585 | 1.58 | 6204 | 2.12 |
| dmap (HO) | 2588 | 5236 | 2.02 | 3279 | 1.27 |
| **Average** | **2321** | **4147** | **1.79** | **4408** | **1.90** |

higher-order example with a function to map two functions to every element of a list.

For the first-order examples, we considered the previous offline partial evaluator of [17,6], the only difference being that in the last two columns the considered program is first transformed with the polyvariant transformation. As it can be seen, the polyvariant transformation improves the speedups in three out of four examples.

For the higher-order examples, since the previous offline partial evaluator did not accept higher-order programs, we compare the new offline partial evaluator with an online partial evaluator for Curry that accepts higher-order functions [1]. In this case, we get an improvement in one of the examples (`dfib`) and a slowdown in the other one (`dmap`). This result is not surprising since an online partial evaluator is usually able to propagate much more information than an offline partial evaluator. Nevertheless, the important remark here is that we are able to deal with programs that could not be dealt with the old version of the offline partial evaluator.

Averages are obtained from the geometric mean of the speedups.

A critical issue of our transformation is that it might produce a significant increase of code size. This is explored in Table 2. Here, although the size of residual programs produced with our approach is slightly bigger than the size of residual programs obtained with previous approaches, it is still reasonable. Actually, our benchmarks confirm that most of the (dead) code added in the

polyvariant transformation has been removed at specialization time. Nevertheless, producing intermediate programs with are too large might be problematic if memory is exhausted. Thus we are currently considering the definition of some program analysis that can be useful for avoiding the introduction of (potentially) dead code during the transformation process.

## 6 Related Work and Conclusions

Let us first review some related works. Defunctionalization was first introduced by Reynolds [18] (see also [10], where a number of applications are presented). Defunctionalization has already been used in the context of partial evaluation (see, e.g., [8]) as well as in the online approach to narrowing-driven partial evaluation [1]. The main novelty w.r.t. these approaches is that we introduced a more aggressive defunctionalization by instantiating functional variables with all possible partial calls. Although it may increase code size, the transformed program has more information explicit and both size-change analysis and specialization may often produce better results.

Size-change analysis has been recently extended to higher-order functional programs in [20]. In contrast to our approach, Sereni proposes a direct approach over higher-order programs that requires the construction of a complex call graph which might produce less efficient binding-time analyses. We have applied our technique to the example in [20] and we got the same accuracy (despite the use of defunctionalization). A deeper comparison is the subject of ongoing work.

Regarding the definition of transformational approaches to polyvariant BTA, we only found the work of [9]. In contrast to our approach, Bulyonkov duplicates the function arguments so that, for every argument of the original function, there is another argument with its binding-time. Furthermore, some additional code to compute the binding-times of the calls in the right-hand sides of the functions is added. Then, a first stage of partial evaluation is run with some concrete values for the binding-time arguments of some function. As a result, the specialized program may include different versions of the same function (for different combinations of binding-times). Then, partial evaluation is applied again using the actual values of the static arguments. Our approach replaces the first stage of transformation and partial evaluation by a simpler transformation based on duplicating code and labeling function symbols. No experimental comparison can be made since we are not aware of any implementation of Bulyonkov's approach.

Other approaches to polyvariant BTA of higher-order programs include Mogensen's work [16] for functional programs and Vanhoof's modular approach [22] for Mercury programs. In contrast to our approach, Mogensen presents a *direct* (i.e., not based on defunctionalization) approach for polyvariant BTA of higher-order functional programs.[9] Vanhoof's approach is also a direct approach to polyvariant BTA of higher-order Mercury programs. A nice aspect of [22] is that no closure analysis is required, since closures are encapsulated in the notion

[9] Actually, Mogensen's approach includes some steps that resemble a defunctionalization process but never adds a definition for an explicit application function.

of binding-time. The integration of some ideas from [22] in our setting could improve the accuracy of the method and reduce the increase of code size.

Other related approaches to improve the accuracy of termination analysis by labeling functions can be found in [19], which is based on a standard technique from logic programming [5]. Here, some program clauses are duplicated and labeled with different *modes*—the mode of an argument can be *input*, if it is known at call time, or *output*, if it is unknown—in order to have a well-moded program where every call to the same predicate has the same modes. This technique can be seen as a simpler version of our polyvariant transformation.

To summarize, in this work we have introduced a transformational approach to polyvariant BTA of higher-order functional programs. Our approach is based on two different transformations: an improved defunctionalization algorithm that makes as much higher-order information explicit as possible, together with a polyvariant transformation that improves the accuracy of the binding-time propagation. We have developed a prototype implementation of the complete partial evaluator, the first offline narrowing-driven partial evaluator that deals with higher-order programs and produces polyvariant specializations. Our experimental results are encouraging and point out that the new BTA is efficient and still sufficiently accurate.

As for future work, there are a number of interesting issues that we plan to investigate further. As mentioned above, [22] presents some ideas that could be adapted to our setting in order to improve the accuracy of the BTA and to avoid the code explosion due to the absence of a separate closure analysis in our transformation. Also, the use of more refined binding-time domains (including partially static information as in, e.g., [16,22]) may improve the accuracy of the specialization at a reasonable cost.

## Acknowledgements

We gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2008 for many useful comments and suggestions.

## References

1. Albert, E., Hanus, M., Vidal, G.: A Practical Partial Evaluation Scheme for Multi-Paradigm Declarative Languages. Journal of Functional and Logic Programming 2002(1) (2002)
2. Albert, E., Vidal, G.: The Narrowing-Driven Approach to Functional Logic Program Specialization. New Generation Computing 20(1), 3–26 (2002)
3. Antoy, S.: Definitional trees. In: Kirchner, H., Levi, G. (eds.) ALP 1992. LNCS, vol. 632, pp. 143–157. Springer, Heidelberg (1992)
4. Antoy, S., Tolmach, A.: Typed Higher-Order Narrowing without Higher-Order Strategies. In: Middeldorp, A. (ed.) FLOPS 1999. LNCS, vol. 1722, pp. 335–352. Springer, Heidelberg (1999)
5. Apt, K.R.: From Logic Programming to Prolog. Prentice-Hall, Englewood Cliffs (1997)

6. Arroyo, G., Ramos, J.G., Silva, J., Vidal, G.: Improving Offline Narrowing-Driven Partial Evaluation using Size-Change Graphs. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 60–76. Springer, Heidelberg (2007)
7. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press, Cambridge (1998)
8. Bondorf, A.: Self-Applicable Partial Evaluation. PhD thesis, DIKU, University of Copenhagen, Denmark, Revised version: DIKU Report 90/17 (1990)
9. Bulyonkov, M.A.: Extracting Polyvariant Binding Time Analysis from Polyvariant Specializer. In: Proc. of Partial Evaluation and Semantics-Based Program Manipulation, Copenhagen, Denmark, pp. 59–65. ACM, New York (1993)
10. Danvy, O., Nielsen, L.R.: Defunctionalization at Work. In: PPDP, pp. 162–174. ACM, New York (2001)
11. Hanus, M. (ed.): Curry: An Integrated Functional Logic Language, `http://www.informatik.uni-kiel.de/~mh/curry/`
12. González-Moreno, J.C.: A correctness proof for Warren's HO into FO translation. In: Proc. of 8th Italian Conf. on Logic Programming, GULP 1993, pp. 569–585 (1993)
13. Hanus, M., Prehofer, C.: Higher-Order Narrowing with Definitional Trees. Journal of Functional Programming 9(1), 33–75 (1999)
14. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Englewood Cliffs (1993)
15. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The Size-Change Principle for Program Termination. In: Proc. of POPL 2001, vol. 28, pp. 81–92 (2001)
16. Mogensen, T.Æ.: Binding Time Analysis for Polymorphically Typed Higher Order Languages. In: Díaz, J., Orejas, F. (eds.) TAPSOFT 1989 and CCIPL 1989. LNCS, vol. 352, pp. 298–312. Springer, Heidelberg (1989)
17. Ramos, J.G., Silva, J., Vidal, G.: Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In: Proc. of the 10th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP 2005), pp. 228–239. ACM Press, New York (2005)
18. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. Higher-Order and Symbolic Computation 11(4), 363–297 (1972); reprinted from the proceedings of the 25th ACM National Conference (1972)
19. Schneider-Kamp, P., Giesl, J., Serebrenik, A., Thiemann, R.: Automated Termination Proofs for Logic Programs by Term Rewriting. ACM Transactions on Computational Logic (to appear) (2008)
20. Sereni, D.: Termination Analysis and Call Graph Construction for Higher-Order Functional Programs. In: Proc. of the 12th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP 2007), pp. 71–84. ACM Press, New York (2007)
21. Slagle, J.R.: Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. Journal of the ACM 21(4), 622–642 (1974)
22. Vanhoof, W.: Binding-Time Analysis by Constraint Solving. A Modular and Higher-Order Approach for Mercury. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNCS, vol. 1955, pp. 399–416. Springer, Heidelberg (2000)

# Analysis of Linear Hybrid Systems in CLP*

Gourinath Banda and John P. Gallagher

Building 43.2, P.O. Box 260, Roskilde University, DK-4000 Denmark
{gnbanda,jpg}@ruc.dk

**Abstract.** In this paper we present a procedure for representing the semantics of linear hybrid automata (LHAs) as constraint logic programs (CLP); flexible and accurate analysis and verification of LHAs can then be performed using generic CLP analysis and transformation tools. LHAs provide an expressive notation for specifying real-time systems. The main contributions are (i) a technique for capturing the reachable states of the continuously changing state variables of the LHA as CLP constraints; (ii) a way of representing events in the LHA as constraints in CLP, along with a product construction on the CLP representation including synchronisation on shared events; (iii) a framework in which various kinds of reasoning about an LHA can be flexibly performed by combining standard CLP transformation and analysis techniques. We give experimental results to support the usefulness of the approach and argue that we contribute to the general field of using static analysis tools for verification.

## 1 Introduction

In this paper we pursue the general goal of applying program analysis tools to system verification problems, and in particular to verification of real-time control systems. The core of this approach is the representation of a given system as a program, so that the semantics of the system is captured by the semantics of the programming language. Our choice of programming language is constraint logic programming (CLP) due to its declarative character, its dual logical and procedural semantics, integration with decision procedures for arithmetic constraints and the directness with which non-deterministic transition systems can be represented. Generic CLP analysis tools and semantics-preserving transformation tools are applied to the CLP representation, thus yielding information about the original system. This work continues and extends previous work in applying CLP to verification of real-time systems, especially [20,13,26].

We first present a procedure for representing the semantics of linear hybrid automata (LHAs) as CLP programs. LHAs provide an expressive notation for specifying continuously changing real-time systems. The standard logical model of the CLP program corresponding to an LHA captures (among other things) the reachable states of the continuously changing state variables of the LHA; previous related work on CLP models of continuous systems [20,26] captured

* Work partly supported by the Danish Natural Science Research Council project *SAFT: Static Analysis Using Finite Tree Automata*.

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 55–70, 2009.

only the transitions between control locations of continuous systems but not all states within a location. The translation from LHAs to CLP is extended to handle events as constraints; following this the CLP program corresponding to the product of LHAs (with synchronisation on shared events) is automatically constructed. We show that flexible and accurate analysis and verification of LHAs can be performed by generic CLP analysis tools coupled to a polyhedron library [5]. We also show how various integrity conditions on an LHA can be checked by querying its CLP representation. Finally, a path automaton is derived along with the analysis; this can be used to query properties and to generate paths that lead to given states.

In Section 2 the CLP representation of transition systems in general is reviewed. In Section 3 we define LHAs and then give a procedure for translating an LHA to a CLP program. Section 4 shows how standard CLP analysis tools based on abstract interpretation can be applied. In Section 5 we report the results of experiments. Section 6 concludes with a discussion of the results and related research.

## 2 Transition Systems

State transition systems can be conveniently represented as logic programs. The requirements of the representation are that the (possibly infinite) set of reachable states can be enumerated, that the values of state variables can be discrete or continuous, that transitions can be deterministic or non-deterministic, that traces or paths can be represented and that we can both reason forward from initial states or backwards from target states.

Various different CLP programs can be generated, providing a flexible approach to reasoning about a given transition system using a single semantic framework, namely, minimal models of CLP programs. Given a program $P$, its least model is denoted $M[\![P]\!]$. In Section 6 we discuss the use of other semantics such as the greatest fixpoint semantics.

**CLP Representation of Transition Systems.** A transition system is a triple $\langle S, I, \Delta \rangle$ where $S$ is a set of *states*, $I \subseteq S$ is a set of *initial states* and $\Delta \subseteq S \times S$ is a transition relation. The set $S$ is usually of the form $V_1 \times \cdots \times V_n$ where $V_1, \ldots, V_n$ are sets of values of *state variables*. A *run* of the transition system is a sequence $s_0, s_1, s_2, \ldots$ where $\langle s_i, s_{i+1} \rangle \in \Delta$, $i \geq 0$. A *valid run* is a run where $s_0 \in I$. A *reachable state* is a state that appears in some valid run. A basic CLP representation is defined by a number of predicates, in particular `init(S)` and `transition(S1,S2)` along with either `rstate(S)` or `trace([Sk,...,S0])`. A program $P$ is formed as the union of a set of clauses defining the transition relation `transition(S1,S2)` and the initial states `init(S)` with the set of clauses in either Figure 1(i) or (ii). Figure 1(i) defines the set of reachable states; that is, $s$ is reachable iff `rstate(s)` $\in M[\![P]\!]$ where `s` is the representation of state $s$. Figure 1(ii) captures the set of valid run prefixes. $s_0, s_1, s_2, \ldots$ is a valid run of the transition system iff for every non-empty finite prefix $r_k$ of $s_0, s_1, s_2, \ldots s_k$,

```
rstate(S2) :-
    transition(S1,S2), rstate(S1).
rstate(S0) :- init(S0).
```
(i) Reachable states

```
trace([S2,S1|T]) :-
   transition(S1,S2), trace([S1|T]).
trace([S0]) :- init(S0).
```
(ii) Run prefixes

```
qstate(S1) :-
   transition(S1,S2), qstate(S2).
qstate(Sk) :- target(Sk).
```
(iii) Reaching states

```
dqstate(S1,Sk) :-
   transition(S1,S2), dqstate(S2,Sk).
dqstate(Sk,Sk) :- target(Sk).
```
(iv) State Dependencies

**Fig. 1.** Various representations of transition systems

`trace(rk)` $\in M[\![P]\!]$, where `rk` is the representation of the reverse of prefix $r_k$. Note that infinite runs are not captured using the least model semantics. The set of reachable states can be obtained directly from the set of valid run prefixes (since $s_k$ is reachable iff $s_0, \ldots, s_k$ is the finite prefix of some valid run).

Alternatively, reasoning backwards from some state can be modelled. Figure 1(iii) gives the backwards reasoning version of Figure 1(i). Given a program $P$ constructed according to Figure 1(iii), its least model captures the set of states $s$ such that there is a run from $s$ to a target (i.e. query) state. The predicate `target(Sk)` specifies the states from which backward reasoning starts. This is the style of representation used in [13], where it is pointed out that relation between forward and backward reasoning can be viewed as the special case of a query-answer transformation or so-called "magic-set" transformation [12]. Figure 1(iv) gives a further variation obtained by recording the dependencies on the target states; `dqstate(S,Sk)` $\in M[\![P]\!]$ iff a target state Sk is reachable from S. There are many other possible variants; system-specific behaviour is captured by the `transition`, `init` and `target` predicates while the definitions of `rstate`, `qstate` and so on capture various semantic views on the system within a single semantic framework. We can apply well-known semantics-preserving transformations such as unfolding and folding in order to gain precision and efficiency. For instance, the `transition` relation is usually unfolded away.

In Section 3 we show how to construct the `transition` relation and the `init` predicates for Linear Hybrid Automata. These definitions can then be combined with the clauses given in Figure 1.

## 3 Linear Hybrid Automata as CLP Programs

Embedded systems are predominantly employed in control applications, which are hybrid in the sense that the system whose behaviour is being controlled has continuous dynamics changing in dense time while the digital controller has discrete dynamics. Hence their analysis requires modelling both discrete and continuous variables and the associated behaviours. The theory of hybrid automata [24] provides an expressive graphical notation and formalism featuring both discrete and continuous variables. We consider only systems whose variables

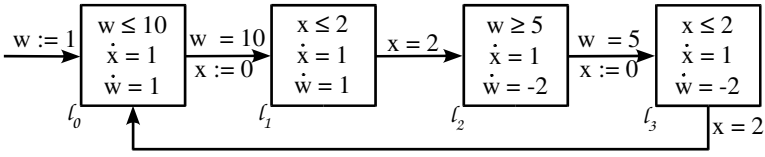



**Fig. 2.** A Water-level Monitor

change linearly with respect to time in this paper, captured by so-called Linear Hybrid Automata [2].

### 3.1 The Language of Linear Hybrid Automata

Following [24], we formally define a linear hybrid automaton (LHA) as a 6-tuple $\langle Loc, Trans, Var, Init, Inv, D\rangle$, with:

- A finite set $Loc$ of locations also called control nodes, corresponding to control modes of a controller/plant.
- A finite set $Var = \{x_1, x_2, \ldots, x_n\}$ of real valued variables, where $n$ is the number of variables in the system. The state of the automaton is a tuple $(l, X)$, where $X$ is the valuation vector in $\mathbb{R}^n$, giving the value of each variable. Associated with variables are two sets:
  - $\dot{Var} = \{\dot{x}_1, \ldots, \dot{x}_n\}$, where $\dot{x}_i$ represents the first derivative of variable $x_i$ w.r.t time;
  - $Var' = \{x'_1, \ldots, x'_n\}$, where $x'_i$ represents $x_i$ at the end of a transition.
- Three functions $Init$, $Inv$ and $D$ that assign to each location $l \in Loc$ three predicates respectively: $Init(l)$, $Inv(l)$ and $D(l)$. The free variables of $Init(l)$ and $Inv(l)$ range over $Var$, while those of $D(l)$ range over $Var \cup \dot{Var}$. An automaton can start in a particular location $l$ only if $Init(l)$ holds. So long as it stays in the location $l$, the system variables evolve as constrained by the predicate $D(l)$ not violating the invariant $Inv(l)$. The predicate $D(l)$ constrains the rate of change of system variables.
- A set of discrete transitions $Trans = \{\tau_1, \ldots, \tau_t\}$; $\tau_k = \langle k, l, \gamma_k, \alpha_k, l'\rangle$ is a transition (i) uniquely identified by integer $k$, $0 \leq k \leq t$; (ii) corresponding to a discrete jump from location $l$ to location $l'$; and (iii) guarded by a predicate $\gamma_k$ and with actions constrained by $\alpha_k$. The guard predicate $\gamma_k$ and action predicate $\alpha_k$ are both conjunctions of linear constraints whose free variables are from $Var$ and $Var \cup Var'$ respectively.

Figure 2 shows the LHA specification of a water-level monitor (taken from [21]).

### 3.2 LHA Semantics and Translation into CLP

**LHA Semantics as a Transition System.** The semantics of LHAs can be formally described as consisting of runs of a labelled transition system ($LHA_t$), which we sketch here (full details are omitted due to lack of space).

A *discrete transition* is defined by $(l, X) \rightarrow (l', X')$, where there exists a transition $\tau = \langle k, l, \gamma, \alpha, l' \rangle \in Trans$ identified by $k$; the guard predicate $\gamma(k, l, l')$ holds at the valuation $X$ in location $l$ and $k$ identifies the guarded transition; the associated action predicate $\alpha(k, l, (l', X), (l', X'))$ holds at valuation $X'$ with which the transition ends entering the new location $l'$ ($k$ identifies the transition that triggers the action). If there are events in the system, the event associated with the transition labels the relation. Events are explained later in this section.

A *delay transition* is defined as: $(l, X) \xrightarrow{\delta} (l^\delta, X^\delta)$ iff $l = l^\delta$, where $\delta \in \mathbb{R}_{\geq 0}$ is the duration of time passed *staying* in the location $l$, during which the predicate $Inv(l)$ continuously holds; $X$ and $X^\delta$ are the variable valuations in $l$ such that $D(l)$ and $Inv(l)$, the predicates on location $l$, hold. The predicate $D(l)$ constrains the variable derivatives $\dot{Var}$ such that $X^\delta = X + \delta * \dot{X}$.

Hence a delay transition of *zero time-duration*, during which the location and valuation remain unchanged is also a valid transition.

A *run* $\sigma = s_0 s_1 s_2 \cdots$ is an infinite sequence of *states* $(l, X) \in Loc \times \mathbb{R}^n$, where $l$ is the location and $X$ is the valuation. In a run $\sigma$, the transition from state $s_i$ to state $s_{i+1}$ are related by either a delay transition or a discrete transition. As the domain of time is dense, the number of states possible via delay transitions becomes infinite following the infinitely fine granularity of time. Hence the delay transitions and their derived states are abstracted by the duration of time ($\delta$) spent in a location. Thus a run $\sigma$ of an $LHA_t$ is defined as $\sigma = (l_0, X_0) \rightarrow_{(\gamma_0, \alpha_0)}^{\delta_0} (l_1, X_1) \rightarrow_{(\gamma_1, \alpha_1)}^{\delta_1} (l_2, X_2) \cdots$, where $\delta_j (j \geq 0)$ is the time spent in location $l_j$ from valuation $X_j$ until taking the discrete transition to location $l_{j+1}$, when the guard $\gamma_j$ holds. The new state $(l_{j+1}, X_{j+1})$ is entered with valuation $X_{j+1}$ as constrained by $\alpha_j$. Further $\tau_j = \langle j, l_j, \gamma_j, \alpha_j, l_{j+1} \rangle \in Trans$. Again during this time duration $\delta_j$, the defined invariant $inv(l_j)$ on $l_j$ continues to hold, and the invariant $inv(l_{j+1})$ holds at valuation $X_{j+1}$.

**LHA semantics in CLP.** Table 1 shows a scheme for translating an LHA specification into CLP clauses. The `transition` predicate defined in the table is then used in the transition system clauses given in Figure 1. A linear constraint such as $Init(l), Inv(l)$, etc. is represented as a CLP conjunction via `to_clp(.)`. The translation of LHAs is direct apart from the handling of the constraints on the derivatives on location $l$, namely $D(l)$ which is a conjunction of linear constraints on $\dot{Var}$. We add an explicit "time stamp" to a state, extending $Var, Var'$ with time variables $t, t'$ respectively giving $Var_t, Var'_t$. The constraint $D_t(l)$ is a conjunction of linear constraints on $Var_t \cup Var'_t$, obtained by replacing each occurrence of $\dot{x_j}$ in $D(l)$ by $(x'_j - x_j)/(t' - t)$ in $D_t(l)$, where $t', t$ represent the time stamps associated with $x'_j, x_j$ respectively.

**Event Semantics in CLP.** A system can be realised from two or more interacting LHAs. In such compound systems, parallel transitions from the individual LHAs rendezvous on events. Thus to model such systems the definition of an LHA is augmented with a *finite set of events* $\Sigma = \{evt_1, \ldots, evt_{ne}\}$. The resulting LHA then is a seven-tuple $\langle Loc, Trans, Var, Init, Inv, D, \Sigma \rangle$. Also associated

**Table 1.** Translation of LHAs to CLP

| LHA | *CLP* |
|---|---|
| location $l$<br>state variables $x_1, \ldots, x_n$<br>state with time $t$ and location $l$ | `L`<br>`X1,...,Xn`<br>`S = [L,X1,...,Xn,T]` |
| state time<br>state location<br>temporal order on states | `timeOf(S,T) :- lastElementOf(S,T).`<br>`locOf(S,L) :- S = [L\|_].`<br>`before(S,S1) :- timeOf(S,T), timeOf(S1,T1), T<T1.` |
| $Init(l)$<br>$Inv(l)$<br>$D(l)$ (using the derivative relation $D_t(l)$ explained in the text) | `init(S) :- locOf(S,L) ,to_clp(`$Init(l)$`).`<br>`inv(S) :- locOf(S,L),to_clp(`$Inv(l)$`).`<br>`d(S,S1):- locOf(S,L), timeOf(S,T), locOf(S1,L), timeOf(S1,T1), to_clp(`$D_t(l)$`).` |
| LHA transition $\langle k, l, \gamma_k, \alpha_k, l' \rangle$ | `gamma(K,L,S) :- locOf(S,L1), to_clp(`$\gamma_k$`).`<br>`alpha(K,L,S1,S2) :- locOf(S1,L1), locOf(S2,L1), to_clp(`$\alpha_k$`).` |
| delay transition<br><br>discrete transition | `transition(S0,S1) :- locOf(S0,L0), before(S0,S1), d(S0,S1),inv(L0,S1).`<br>`transition(S0,S2):- locOf(S0,L0), before(S0,S1), d(S0,S1), gamma(K,L0,S1),alpha(K,L0,S1,S2).` |

with events is a partial function $event : Trans \hookrightarrow \Sigma$, which labels (some) transitions with events.

In our framework we model event notification as constraints. To this end events are modelled as discrete state variables ranging over values {0,1}; these variables are initialised to 0; on an event-labelled transition the variable corresponding to the labelled event is set to value 1 to raise that event while the other event variables are reset to 0; on a transition not labelled with an event all event variables are reset to 0; the event variables remain constant within a location and thus at most one event variable can have value 1 at any time. In the CLP translation the state vector of an LHA with events is given by $S = [Loc, X_1, \ldots, X_n, Evt_1, \ldots, Evt_{ne}, T]$, where for j = 1 to $ne$ the variable $Evt_j$ represents $evt_j \in \Sigma$. The raising of event $evt_{re}$ is modelled as a constraint $\mathrm{E}_{re} = (\bigwedge_{i=1}^{ne} Evt_i = c_i)$ where the value $c_i$ equals 1 if $i = re$ or 0 otherwise. Similarly the constraint corresponding to *no event raised* is $\mathrm{E}_{none} = (\bigwedge_{i=1}^{ne} Evt_i = 0)$.

We modify the previous translation of the derivative constraints $D(l)$ to incorporate events, yielding the following definition of `d`/2.

$$\texttt{d}([l, X_1^0, \ldots, X_n^0, Evt_1^0, \ldots, Evt_n^0, T_0], [l, X_1^1, \ldots, X_n^1, Evt_1^1, \ldots, Evt_{ne}^1, T_1]) \leftarrow$$
$$LC_{d_{Evt}}, LC_{d_l}, X_1^1 = X_1^0 + d_{x_1} * (T_1 - T_0), \ldots, X_n^1 = X_n^0 + d_{x_n} * (T_1 - T_0),$$
$$Evt_1^1 = 0, \ldots, Evt_n^1 = 0.$$

The translation of predicate $\alpha$ to `alpha`/4 is modified to encode the event notification as follows.

$$\begin{aligned}\texttt{alpha}(T, L_0, [L_1, X_1, \ldots, X_n, Evt_1, \ldots, Evt_{ne}], \\ [L_1, X_1', \ldots, X_n', Evt_1', \ldots, Evt_{ne}']) \leftarrow \\ LAC_1, \ldots, LAC_{na},\ E_{xe}.\end{aligned}$$

where if $Evt_{re}$ is the label on the transition $E_{xe} = \mathrm{E}_{re}$ else $E_{xe} = \mathrm{E}_{none}$ when there is no event label (where $\mathrm{E}_{re}$ and $\mathrm{E}_{none}$ are as defined above). The translation of the $\gamma$ and $Inv$ constraints is unaffected by the addition of events apart from the extension of the state vector with the event variables.

### 3.3 Parallel Composition of Linear Hybrid Automata

The *discrete transitions* of two automata $LHA_1$ and $LHA_2$ with events $\Sigma_1$ and $\Sigma_2$ respectively, synchronise on an event $evt$ as following:

- if $evt \in \Sigma_1 \cap \Sigma_2$, then the discrete transitions $\tau_i \in Trans_1$ and $\tau_j \in Trans_2$ labelled with the event $evt$ must synchronize;
- if $evt \notin \Sigma_1 \cap \Sigma_2$ but $evt \in \Sigma_1$, then the discrete transition $\tau_i \in Trans_1$ can occur simultaneously with a zero duration delay transition of $LHA_2$, and similarly if $evt \in \Sigma_2$

Finally, a *delay transition* of $LHA_1$ with a duration $\delta$ must synchronize with a delay transition of $LHA_2$ of the same duration.

Synchronisation is enforced by constructing a product of the associated labelled transition systems ($LHA_t$). In our framework, the product of two labelled transition systems is realised as the *composition* $\boxtimes$ of the corresponding CLP programs, which corresponds closely to the LHA product construction as defined in [24]. More efficient encodings have been investigated [28,26]. $\boxtimes$ is defined as $CLP_1 \boxtimes CLP_2 = \{C_1 \boxtimes C_2 \mid C_1 \in CLP_1, C_2 \in CLP_2\}$ where $C_1 = p2(\overline{X}) \leftarrow c_1(\overline{X}, \overline{X}'), p1(\overline{X}')$ is a clause in $CLP_1$, $C_2 = q2(\overline{Y}) \leftarrow c_2(\overline{Y}, \overline{Y}'), q1(\overline{Y}')$ is a clause in $CLP_2$ and $C_1 \boxtimes C_2 = p2\_q2(\overline{X \cup Y}) \leftarrow c_1(\overline{X}, \overline{X}') \wedge c_2(\overline{Y}, \overline{Y}'), p1\_q1(\overline{X' \cup Y'})$. Here $p2\_q2$ and $p1\_q1$ are new predicates unique to the associated predicate pairs in the original programs. The notation $\overline{X}$ above, where $X$ is a set, denotes a tuple of the elements of $X$ in some fixed order (e.g. alphabetical order).

The operation $\boxtimes$ is quadratic in that the number of clauses in the resultant $CLP_1 \boxtimes CLP_2$ equals $\mid CLP_1 \mid \times \mid CLP_2 \mid$. However with shared events many of them can be eliminated since their constraints are not consistent due to the event constraints. If there is a constraint $E = 1$ on a shared event variable $E$ in some transition, then it will only form a consistent product clause with other clauses with $E = 1$. Following this composition we successfully built product systems of: (i) a task scheduler (ii) a train gate controller (LHA) (iii) a train gate controller (TSA/TA), from their constituent automata.

### 3.4 Integrity Constraints on LHAs

The semantics of LHA given in Section 3 places certain restrictions on runs, in particular that the relevant invariant is satisfied so long as the automaton

remains in a location. One approach to ensuring that the CLP program generates only valid runs is to build into the transitions all the necessary constraints. An alternative is to check statically that certain constraints on the CLP program are satisfied. This enables us to generate a simpler CLP model than otherwise, omitting some "runtime" checks. These integrity checks also represent natural sanity checks on the LHA and checking them can locate specification errors. The conditions are as follows. (i) The invariants are convex (with respect to the given rates of change of the state variables). (ii) The invariants are satisfied when a location is entered via a transition from another location or by initial conditions. (iii) The enabling constraint ($\gamma$) on a transition out of location either implies the invariant on that location, or becomes true as soon as the invariant ceases to be true (e.g. the invariant might be $x < 10$ and the transition constraint $x = 10$, where $x$ increases with time). This condition should be checked, because the language does permit *discontiguous* invariant and guard conditions that result in a situation where the invariant becomes invalid with the outgoing discrete transition guard not yet enabled.

We check these by running queries on predicates representing the negation of the integrity conditions, which ought to fail if the conditions are met. For example $\texttt{nonconvex}(L)$ is defined as $\texttt{nonConvex}(L) \leftarrow \texttt{locOf}(S_0, L), \texttt{inv}(L, S_0), \texttt{d}(S_0, S_2), \texttt{inv}(L, S_2), \texttt{d}(S_0, S_1), \texttt{before}(S_1, S_2), \texttt{negInv}(L, S_1)$. $\texttt{negInv}(L, S_1)$ is the negation of the invariant on location $L$ (the constraint language is closed under negation) and in general consists of several clauses since the negation of the invariant may be a disjunction.

We have noticed that condition (iii) above is violated in several LHAs in the literature. Typically, a transition that can fire "at any time" (perhaps triggered by an interrupt event) has $\gamma = \mathit{true}$. Hence this remains enabled even when the invariant is false. If a violation occurs we can repair it by simply conjoining the invariant on the location to the transition constraint $\gamma$; this is the implicit intention (enforced by the LHA semantics) and achieves the required behaviour.

## 4 Analysis of the CLP Representations

The concrete analysis problem is firstly to obtain an extensional, finite representation of the model; by extensional is meant a representation in which we can query and check model properties using simple constraint operations. In fact, we use CLP clauses in which the bodies consist only of constraints to represent the model (or an over-approximation of the model) of a CLP program. Thus checking of properties reduces in many cases to constraint solving [35].

**Computing a Model.** The usual immediate-consequences operator $T_P$ for logic programs is modified for CLP programs [25]. The sequence $T_P^i(\emptyset)$, $i = 0, 1, 2, \ldots$ is an increasing (w.r.t. subset ordering) sequence of programs whose clauses consist of *constrained facts*, clauses of form $p(\bar{X}) \leftarrow c(\bar{X})$ where $c(\bar{X})$ is a linear constraint. If the sequence stabilises with $T_P^i(\emptyset) = T_P^{i+1}(\emptyset)$ for some $i$, then $T_P^i(\emptyset)$ is the *concrete model* of $P$.

If the sequence does not stabilise (within some predefined resource limits) we are forced to compute an *abstract model*. There are various ways of doing this. Currently we use two abstractions: an abstraction defined in [13] and a classical abstract interpretation based on convex polyhedral hulls with widening and narrowing operators [11]. More complex and precise abstractions, such as those based on the powerset domain consisting of finite sets of convex polyhedra could be used [4]. The resulting abstract model can be represented as a set of constrained facts, as in the concrete model (since a convex polyhedron can be represented as a linear constraint).

**Checking Properties.** The concrete or abstract models can be used to check state safety properties. Suppose the constraint $c(\bar{X})$ defines some "bad" state. Then the query $\leftarrow c(\bar{X}), \texttt{rstate}(\bar{X})$ is evaluated on the model. If this query has no solutions, the safety property is verified. Note that if the query succeeds, with an answer constraint, this means that the bad state is possibly reachable. The answer constraints yield a description of the state variables in the unsafe states.

An alternative approach to checking safety properties is to define the unsafe states as target states and compute the set of *reaching states* w.r.t. those target states, namely those states from which there exists a run to a target state. We then query this set to see whether any of the initial states are among them. If not, then the safety property is satisfied. This is the approach used in [13] for example. As before, we obtain in general an answer constraint. Suppose we obtain the answer that $\texttt{qstate}(\bar{X}), c(\bar{X})$ is a reaching state that is also an initial state. Then we can use this information to strengthen the conditions on the initial states; by adding extra checks to the initial states to ensure that that the constraint $\neg c(\bar{X})$ holds, we can ensure satisfaction of the safety condition.

The CLP program constructed according to the schema summarised in Figure 1(iv) allows certain path properties to be captured. In the backwards version, the model of the program captures dependencies between the target state and reaching states. This model can be used to answer queries such as "Is there a run from a given state $s_1$ to a target state $s_2$?" or "Is it possible to reach state $s_1$ before state $s_2$?" The answers to the queries, as before, could yield constraints giving conditional answers, with constraints linking the values of states in the start and end states.

**Checking Path Properties.** One approach to checking properties of paths in the transition system is to use an explicit representation of the traces in the CLP program, using the scheme from Figure 1(ii) for example. However this can make the model of the program infinite even when the set of states is finite. Another approach is to build a *path automaton* while computing a model. A path automaton is a set of transitions of a tree grammar of the form $f_i(v) \rightarrow v'$. This means that the state $v'$ can be reached from state $v$ via the $i$th transition. During the construction of the model we record which transitions can be applied in a state, where the states $v_1, \ldots, v_k$ are identifiers for the constrained facts in the models (see above). With this automaton we can; (a) generate paths that reach some particular state; (b) check whether there is some "stuck" state which

can be reached but from which no other state is reachable; (c) check regular path properties using standard automata operations. Analysis techniques based on tree automata [16,17] are applied for such analyses.

## 5 Experiments

In this section, experiments applying the CLP analysis tools to LHA systems are described. The tool-chain is as follows: $\text{LHA}_{spec} \xrightarrow{parse} \text{CLP} \xrightarrow{PE} \text{CLP}_{trans} \xrightarrow{prod} \text{CLP}_{trans} \xrightarrow{Reach} \text{CLP}_{model} \xrightarrow{FTA} \text{CLP}_{path}$. Here $\text{LHA}_{spec}$ represents LHA specifications; CLP represents arbitrary CLP programs; $\text{CLP}_{trans}$ represents a subclass of CLP whose clauses are of the form $p_i(\bar{X}) \leftarrow c(\bar{X}, \bar{X}'), p_j(\bar{X}')$, where $\bar{X}, \bar{X}'$ are tuples of distinct variables, $c(\bar{X}, \bar{X}')$ is a conjunction of constraints over the variables and $p_i, p_j$ are non-constraint predicates (and $p_j$ is possibly absent); $\text{CLP}_{model}$ is a subclass of $\text{CLP}_{trans}$ where the clause bodies consist only of constraints; $\text{CLP}_{path}$ is a CLP program defining a finite tree automaton.

The $\xrightarrow{parse}$ step translates LHA specifications given in a simple source language which is simply a textual representation of the usual graphic specifications, into CLP according to the procedure defined in Section 3. In step $\xrightarrow{PE}$ the partial evaluator Logen [29] is applied in order to unfold the definitions of all the predicates except the state predicate `rstate` (or variations such as `qstate` or `dqstate`). Furthermore Logen *filters* are chosen to cause the single state argument to be replaced by a tuple of arguments for the state variables, with a separate state predicate for each location. The resulting programs are in the class $\text{CLP}_{trans}$. In step $\xrightarrow{prod}$ the product of LHAs in $\text{CLP}_{trans}$ form can be computed if necessary, yielding another $\text{CLP}_{trans}$ program. The step $\xrightarrow{Reach}$ uses an analysis tool to compute (an approximation of) the least model of a CLP program, represented as a program in the class $\text{CLP}_{model}$. This program can be used to check properties of the set of reachable states in the original LHA. Finally the step $\xrightarrow{FTA}$ derives a CLP program in the form of a finite automaton generating the set of possible paths in the preceding $\text{CLP}_{trans}$ program. This program can be used to check path properties of the original LHA. Steps $\xrightarrow{parse}$ and $\xrightarrow{PE}$ are standard and are not discussed in detail here. Step $\xrightarrow{prod}$ is also a straightforward implementation of the definition of product. We focus on the analysis phases.

*CLP Analysis Tools.* Our analysis tools are developed to analyse arbitrary CLP programs, in applications such as termination analysis and complexity analysis [7,31,10,18]. We have not developed any analysis tools specifically for the CLP programs resulting from LHA translation. This is an important principle since CLP is a target representation for many different source languages; thus the same set of CLP analysis tools is applicable regardless of the source language. We have previously used similar tools to analyse PIC microprocessor code [23,22]. We use an implementation in Ciao-Prolog that employs the Parma Polyhedra Library (PPL) [5] to handle the constraints. Note that this tool-set is a current snapshot; one of the key advantages of the approach is that improved CLP program

analyses can be incorporated as they become available. In particular we expect that analysis tools based on the powerset of convex polyhedra with widenings [4] will play an important role.

$T_P$: This computes the model of a program using a least fixpoint iteration. Well-known optimisations such as the *semi-naive* technique are incorporated. If it terminates (within some predefined period) the computed model is precise (provided the input program contains only linear constraints – the analyser makes some safe linear over-approximation of non-linear constraints).

DP99: This computes an over-approximation of the least model; the technique was proposed by Delzanno and Podelski in [13] and used in their experiments. Each predicate with argument $\bar{x}$ is approximated by a finite set of conjunctions of linear constraints over $\bar{x}$. The "widening" (which as they point out does not in fact guarantee termination) on successive approximations $F, F'$ returns each conjunction that is obtained from some conjunct $c_1 \wedge \ldots \wedge c_n \in F'$ by removing all conjuncts $c_i$ that are strictly entailed by some conjunct $d_j$ of some "compatible" constrained atom $d_1 \wedge \ldots \wedge d_m \in F$ where "compatible" means that $c_1 \wedge \ldots \wedge d_m$ is satisfiable. It does terminate in some cases where $T_P$ does not.

For cases that do not terminate within some resource bound using one of the above two tools, we use the CHA tool. This is an abstract interpreter computing an over-approximation of the least model, based on convex polyhedral approximations. It computes one polyhedron for each predicate. Termination is guaranteed by one of the known widenings [21,5]. CHA incorporates state-of-the-art refinements such as an optional narrowing phase, "widening up-to" [21] and delayed widening. The tool is available on-line (http://saft.ruc.dk/CHA/).

In Table 2 we summarise the results of computing a model or approximate model for a number of examples. As discussed earlier, querying this model to check safety properties or dependencies is computationally straightforward using a constraint solver. The number of locations in the automaton is $Q$ and number of discrete transitions is $\Delta$. The number of clauses in the translated CLP programs includes the clauses for the delay transitions. For the FTA size, the table reports the size for the most precise analysis that terminated. Timings are given in seconds and the symbol $\infty$ indicates failure to terminate within a time-out duration of 300 seconds.

*Description of the Examples.* The *Fischer Protocol*, *Water Level* and *Scheduler* are taken from [21]; version (E) of the Scheduler is constructed using the product construction synchronised by events while the other one is specified directly as a single automaton. The *Leaking Burner* is taken from [1]; the *Train Controller* is specified in [2] including a number of state variables such as the distance of the train and the angle of the gate, while [26] provides a simpler form as a timed automaton (TA) specifying only the events. The steam boiler problem is work in progress. The nine last examples are taken from [13]; these are specified as discrete automata (which can be modelled as special cases of LHAs)[1]. In all

[1] We gratefully acknowledge the use of the examples from [13] which are available for download and were translated into standard CLP programs for our experiments.

**Table 2.** Experimental Results

| Name | $Q$ | $\Delta$ | $\|CLP\|$ | $T_P$ (secs.) | DP99 (secs.) | CHA (secs.) | \|FTA\| |
|---|---|---|---|---|---|---|---|
| Fischer Protocol | 6 | 8 | 53 | 0.19 | 0.61 | 0.23 | 79 |
| Leaking Burner | 2 | 2 | 5 | $\infty$ | $\infty$ | 0.02 | 5 |
| Scheduler | 3 | 11 | 19 | 1.02 | 9.59 | 0.42 | 101 |
| Scheduler (E) | 3 | 11 | 15 | 0.94 | 40.16 | 0.63 | 93 |
| Steam Boiler | 10 | 23 | 166 | 0.50 | 0.74 | 0.88 | 146 |
| Switch | 2 | 2 | 15 | 0.03 | 0.08 | 0.04 | 33 |
| Train Controller | 3 | 8 | 26 | 0.15 | 1.44 | 0.15 | 57 |
| Train Train | 3 | 3 | 18 | 0.07 | 0.38 | 0.09 | 51 |
| Train Gate | 4 | 10 | 29 | 0.06 | 0.31 | 0.09 | 43 |
| Train Controller System | 14 | 20 | 143 | 4.54 | $\infty$ | 8.00 | 324 |
| Train Controller System (TA) | 14 | 20 | 169 | 1.01 | 36.68 | 2.17 | 85 |
| Water Level | 4 | 4 | 27 | 0.04 | 1.26 | 0.16 | 56 |
| Bakery 2 | 3 | 8 | 9 | 0.08 | 0.93 | 0.04 | 14 |
| Bakery 3 | 3 | 21 | 24 | 3.02 | 127.72 | 0.07 | 134 |
| Bakery 4 | 3 | 52 | 58 | 143.19 | $\infty$ | 0.34 | 1456 |
| Bbuffer 1 | 1 | 4 | 6 | 0.01 | 0.07 | 0.02 | 4 |
| Bbuffer 2 | 1 | 2 | 4 | 0.01 | 0.01 | 0.01 | 4 |
| MutAst | 7 | 20 | 21 | 0.27 | 0.60 | 0.09 | 30 |
| Network | 1 | 16 | 17 | $\infty$ | 0.12 | 0.30 | 13 |
| Ticket 2 | 3 | 6 | 7 | $\infty$ | 2.82 | 0.04 | 25 |
| Ubuffer | 3 | 6 | 9 | $\infty$ | 0.24 | 0.05 | 14 |

of these examples we check the same properties as in the original references, though for the examples from [13] we cover only the safety properties. (We discuss analysis of liveness properties in Section 6).

Many of these examples can be analysed with $T_P$, and thus without using abstraction since the input programs contain only linear constraints. Though the number of states is infinite due to continuously changing state variables, the reachable states can often be captured by a finite set of constraints. We believe that this observation is related to the existence of a finite number of *regions* in timed automata [3]. We were only forced to use the convex polyhedral analyser for the *Leaking Burner* problem and the DP99 abstraction for a few of the discrete examples. We were able to verify some properties that could not be verified with convex hull analysis, such as the bound on the $k_1$ variable, which is the number of lower priority tasks waiting and/or preempted in the *Scheduler* example [21]. The time to compute a concrete model is of course often less than that needed for a convex polyhedral abstraction. However, even when a system can be analysed without abstraction, state-space explosion could force abstractions to be introduced; the *Bakery* series of examples indicates the exponential growth in the time to compute the model as more processes are introduced. In our experience the abstraction introduced in [13] (DP99 in Table 2) is of limited usefulness. We could find no examples apart from the ones in [13] in which a system failed to terminate in $T_P$ but terminated with DP99. Nevertheless the ideas behind DP99 are valid; other approaches based on the

powerset of convex polyhedra with true widenings [4] will replace DP99 in future experiments.

## 6 Related Work and Conclusions

The idea of modelling transition systems of various kinds as CLP programs goes back many years. Our work contributes to two areas, CLP modelling and CLP proof techniques. On the one hand we add to the literature on CLP modelling techniques in showing that the continuous semantics of LHAs can be captured in CLP programs. On the other we add to the existing literature showing that effective reasoning can be carried out on CLP programs, and display a family of different reasoning styles (e.g. forward, backwards, state dependencies) that can be generated from a single system specification and handled with a single set of analysis tools.

Though comparing various formalisms is not the aim of our work, it is noteworthy that LHAs are *more expressive* than other formalisms such as Timed Automata (TA) [3] or other finite automata as discussed in [9]. Consequently, from the modelling perspective, our framework has two advantages over the Uppaal [6] model checker or any other TA model checkers. Firstly, we can directly handle LHAs having *multi-rate dynamics*[2] , whereas Uppaal mandates that an LHA specification be compiled down into a TA specification before being verified. Secondly, Uppaal restricts the clock variables to be compared only with natural numbers, i.e. guards such as $x > 1.1$, $10 * x > 11$ or $x > y$ are not permitted [27], while the CLP(Q) system permits us to handle such constraints. Such advantages in expressiveness might be balanced by extra complexity and possible non-termination; in the case of non-termination, we resort to abstraction. Furthermore, the state transition system $CLP_{trans}$ (see Section 5) can be directly input to Uppaal after relatively minor syntactic changes. Thus, our approach can also be regarded as potentially providing a flexible *LHA input interface* incorporating abstraction for TA model checkers (among others). We experimented with Uppaal, in this manner, to verify the Water-level control system using CLP as an intermediate representation.

Gupta and Pontelli [20] and Jaffar *et al.* [26] describe schemes for modelling timed (safety) automata (TSAs) as CLP programs. Our work has similarities to both of these, but we go beyond them in following closely the standard semantics for LHAs, giving us confidence that the full semantics has been captured by our CLP programs. In particular the set of all reachable states, not only those occurring at transitions between locations as in the cited works, is captured. Delzanno and Podelski [13] develop techniques for modelling discrete transition systems which in our approach are special cases of hybrid systems.

Another direction we could have taken is to develop a direct translation of LHA semantics into CLP, without the intermediate representation as a transition system. For example a "forward collecting semantics" for LHAs is given in [21],

[2] A timed automaton has variables called *clocks* that vary at a single rate i.e. $\dot{x} = 1, \dot{y} = 1$, while an LHA can have variables that vary at different rates i.e. $\dot{x} = 1, \dot{y} = 2$.

in the form of a recursive equation defining the set of reachable states for each location. It would be straightforward to represent this equation as a number of CLP clauses, whose least model was equivalent to the solution of the equation. A technique similar to our method of deriving the `d` predicate for the constraints on the derivatives could be used to model the operator $S \nearrow D$ in [21] representing the change of state $S$ with respect to derivative constraints $D$. The approach using transition systems is a little more cumbersome but gives the added flexibility of reasoning forwards or backwards, adding traces, dependencies and so on as described in Section 2. The clauses we obtain for the forward transition system (after partial evaluating the `transition` predicate) are essentially what would be obtained from a direct translation of the recursive equation in [21]. The toolchain in Section 5 differs only in the choice of driver in the *PE* step.

We focus on the application of standard analysis techniques using the bottom up least fixpoint semantics, but there are other approaches which are compatible with our representations. In [20] the CLP programs are run using the usual procedural semantics; this has obvious limitations as a proof technique. In [26] a method called "co-inductive" tabling is used. Path properties are expressed in [20] using CLP list programs; again this is adequate only for proofs requiring a finite CLP computation. [13] contains special procedures for proving a class of safety and liveness properties. Liveness properties require a greatest fixpoint computation (as used in [26] and [19]), which our toolset does not yet support. Our approach focusses on using standard CLP analysis techniques and abstractions based on over-approximations. However, the introduction of greatest fixpoint analysis engines into our framework is certainly possible and interesting.

A somewhat different but related CLP modelling and proof approach is followed in [8,30,32,14,15,33,34]. This is to encode a proof procedure for a modal logic such as CTL, $\mu$-calculus or related languages as a logic program, and then prove formulas in the language by running the interpreter (usually with tabling to ensure termination). The approach is of great interest but adapting it for abstraction of infinite state systems seems difficult since the proof procedures themselves are complex programs. The programs usually contain a predicate encoding the transitions of the system in which properties are to be proved, and thus could in principle be coupled to our translation. In this approach the full prover is run for each formula to be proved, whereas in ours an (abstract) model is computed once and then queried for difference properties. Our approach is somewhat more low-level as a proof technique but may offer more flexibility and scalability at least with current tools.

**Acknowledgements.** We thank the LOPSTR'2008 referees and Julio Peralta for helpful comments and suggestions.

## References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)

2. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.-H.: Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R.L., Ravn, A.P., Rischel, H., Nerode, A. (eds.) HS 1991 and HS 1992. LNCS, vol. 736, pp. 209–229. Springer, Heidelberg (1993)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
4. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. J. Software Tools for Technology Transfer 8(4-5), 449–466 (2006)
5. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly not closed convex polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 213–229. Springer, Heidelberg (2002)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
7. Benoy, F., King, A.: Inferring argument size relationships with CLP(R). In: Gallagher, J.P. (ed.) LOPSTR 1996. LNCS, vol. 1207, pp. 204–223. Springer, Heidelberg (1997)
8. Brzoska, C.: Temporal logic programming in dense time. In: ILPS, pp. 303–317. MIT Press, Cambridge (1995)
9. Carloni, L.P., Passerone, R., Pinto, A., Sangiovanni-Vincentelli, A.L.: Languages and tools for hybrid systems design. Found. Trends Electron. Des. Autom. 1(1/2), 1–193 (2006)
10. Codish, M., Taboch, C.: A semantic basic for the termination analysis of logic programs. The Journal of Logic Programming 41(1), 103–123 (1999)
11. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages, pp. 84–96 (1978)
12. Debray, S., Ramakrishnan, R.: Abstract Interpretation of Logic Programs Using Magic Transformations. Journal of Logic Programming 18, 149–176 (1994)
13. Delzanno, G., Podelski, A.: Model checking in CLP. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 223–239. Springer, Heidelberg (1999)
14. Du, X., Ramakrishnan, C.R., Smolka, S.A.: Real-time verification techniques for untimed systems. Electr. Notes Theor. Comput. Sci. 39(3) (2000)
15. Fioravanti, F., Pettorossi, A., Proietti, M.: Verifying CTL properties of infinite-state systems by specializing constraint logic programs. In: Leuschel, M., Podelski, A., Ramakrishnan, C., Ultes-Nitsche, U. (eds.) Proceedings of the Second International Workshop on Verification and Computational Logic (VCL 2001), pp. 85–96 (2001); tech. Report DSSE-TR-2001-3, University of Southampton
16. Gallagher, J.P., Henriksen, K.S.: Abstract domains based on regular types. In: Demoen, B., Lifschitz, V. (eds.) ICLP 2004. LNCS, vol. 3132, pp. 27–42. Springer, Heidelberg (2004)
17. Gallagher, J.P., Henriksen, K.S., Banda, G.: Techniques for scaling up analyses based on pre-interpretations. In: Gabbrielli, M., Gupta, G. (eds.) ICLP 2005. LNCS, vol. 3668, pp. 280–296. Springer, Heidelberg (2005)
18. Genaim, S., Codish, M.: Inferring termination conditions of logic programs by backwards analysis. In: Nieuwenhuis, R., Voronkov, A. (eds.) LPAR 2001. LNCS, vol. 2250, pp. 681–690. Springer, Heidelberg (2001)
19. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive logic programming and its applications. In: Dahl, V., Niemelä, I. (eds.) ICLP 2007. LNCS, vol. 4670, pp. 27–44. Springer, Heidelberg (2007)

20. Gupta, G., Pontelli, E.: A constraint-based approach for specification and verification of real-time systems. In: IEEE Real-Time Systems Symposium, pp. 230–239 (1997)
21. Halbwachs, N., Proy, Y.E., Raymound, P.: Verification of linear hybrid systems by means of convex approximations. In: LeCharlier, B. (ed.) SAS 1994. LNCS, vol. 864, pp. 223–237. Springer, Heidelberg (1994)
22. Henriksen, K.S., Banda, G., Gallagher, J.P.: Experiments with a convex polyhedral analysis tool for logic programs. In: Workshop on Logic Programming Environments, Porto (2007)
23. Henriksen, K.S., Gallagher, J.P.: Abstract interpretation of PIC programs through logic programming. In: Proceedings of SCAM 2006, Sixth IEEE International Workshop on Source Code Analysis and Manipulation (2006)
24. Henzinger, T.A.: The theory of hybrid automata. In: Clarke, E.M. (ed.) Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, pp. 278–292. IEEE Computer Society Press, Los Alamitos (1996)
25. Jaffar, J., Maher, M.: Constraint Logic Programming: A Survey. Journal of Logic Programming 19/20, 503–581 (1994)
26. Jaffar, J., Santosa, A.E., Voicu, R.: A CLP proof method for timed automata. In: Anderson, J., Sztipanovits, J. (eds.) The 25th IEEE International Real-Time Systems Symposium, pp. 175–186. IEEE Computer Society, Los Alamitos (2004)
27. Katoen, J.-P.: Concepts, algorithms, and tools for model checking. A lecture notes of the course "Mechanised Validation of Parallel Systems" for 1998/99 at Friedrich-Alexander Universitat, Erlangen-Nurnberg, p. 195 (1999)
28. Leuschel, M., Fontaine, M.: Probing the depths of CSP-M: A new fdr-compliant validation tool. In: Liu, S., Maibaum, T.S.E., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 278–297. Springer, Heidelberg (2008)
29. Leuschel, M., Jørgensen, J.: Efficient specialisation in Prolog using the hand-written compiler generator LOGEN. Elec. Notes Theor. Comp. Sci. 30(2) (1999)
30. Leuschel, M., Massart, T.: Infinite state model checking by abstract interpretation and program specialisation. In: Bossi, A. (ed.) LOPSTR 1999. LNCS, vol. 1817, pp. 63–82. Springer, Heidelberg (2000)
31. Mesnard, F.: Towards automatic control for CLP($\chi$) programs. In: Proietti, M. (ed.) LOPSTR 1995. LNCS, vol. 1048, pp. 106–119. Springer, Heidelberg (1996)
32. Nilsson, U., Lübcke, J.: Constraint logic programming for local and symbolic model-checking. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS, vol. 1861, pp. 384–398. Springer, Heidelberg (2000)
33. Pemmasani, G., Ramakrishnan, C.R., Ramakrishnan, I.V.: Efficient real-time model checking using tabled logic programming and constraints. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 100–114. Springer, Heidelberg (2002)
34. Peralta, J.C., Gallagher, J.P.: Convex hull abstractions in specialization of CLP programs. In: Leuschel, M.A. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 90–108. Springer, Heidelberg (2003)
35. Podelski, A.: Model checking as constraint solving. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 22–37. Springer, Heidelberg (2000)

# Automatic Generation of Test Inputs for Mercury

François Degrave[1], Tom Schrijvers[2], and Wim Vanhoof[1]

[1] Faculty of Computer Science, University of Namur, Belgium
{fde,wva}@info.fundp.ac.be
[2] Department of Computer Science, K.U. Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

**Abstract.** In this work, we consider the automatic generation of test inputs for Mercury programs. We use an abstract representation of a program that allows to reason about program executions as paths in a control-flow graph. Next, we define how such a path corresponds to a set of constraints whose solution defines input values for the predicate under test such that when the predicate is called with respect to these input values, the execution is guaranteed to follow the given path. The approach is similar to existing work for imperative languages, but has been considerably adapted to deal with the specificities of Mercury, such as symbolic data representation, predicate failure and non-determinism.

## 1 Introduction

It is a well-known fact that a substantial part of a software development budget (estimates range from 50% to 75% [1]) is spent in *corrective maintenance* – the act of correcting errors in the software under development. Arguably the most commonly applied strategy for finding errors and thus producing (more) reliable software is testing. Testing refers to the activity of running a software component with respect to a well-chosen set of inputs and comparing the outputs that are produced with the expected results in order to find errors. While the fact that the system under test passes successfully a large number of tests does not prove correctness of the software, it nevertheless increases confidence in its correctness and reliability [2].

In testing terminology, a *test case* for a software component refers to the combination of a single test input and the expected result whereas a *test suite* refers to a collection of individual test cases. Running a test suite is a process that can easily be automated by running the software component under test once for each test input and comparing the obtained result with the expected result as recorded in the testcase.

The hard part of the testing process is *constructing* a test suite, which comprises finding a suitable set of test inputs either based on the specification (*black-box* testing) or on the source code of program (*whitebox* or *structural* testing). In the latter approach, which we follow in this work, the objective is to create a set of test inputs that cover as much source code as possible according to some *coverage criterion*; some well-known examples being statement, branch and path coverage [3].

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 71–86, 2009.

Although some work exists on automatic generation of test inputs in functional and logic programming languages [4,5,6,7], most attempts to automate the generation of test inputs have concentrated on imperative [8,9,10] or object oriented programs [11,12]. In this work we consider the automatic generation of test inputs for programs written in the logic programming language Mercury [13]. Although being a logic programming language, Mercury is strongly moded and allows thus to generate a complete control flow graph similar to that of imperative programs which in turn makes it possible to adapt techniques for test input generation originally developed in an imperative setting. Such a control-flow based approach is appealing for the following reasons: (1) Given a path in the graph that represents a possible execution for a predicate, one can compute input values such that when the predicate is called with respect to those values, its execution will follow the derivation represented by the given path. These input values can easily be converted in a *test case* by the programmer: it suffices to add the expected output for the predicate under consideration. (2) Given a (set of) program point(s) within the graph, it is possible to compute a path through the graph that ressembles a computation covering that (set of) program point(s). By extension, this allows for algorithms to compute a set of execution paths (and thus test cases) guided by some coverage criterion [3].

The approach we present is similar in spirit to the constraint based approach of [8], although considerably adapted and extended to fit the particularities of a logic programming language. In particular, the fact that our notion of execution path captures failures and the sequence of answers returned by a nondeterministic predicate allows to automatically generate test inputs that test the implementation strategy of possibly failing and non-deterministic predicates. In addition, since *all* dataflow in a Mercury program can easily be represented by constraints on the involved variables implies that our technique is, in contrast to [8], not limited to numerical values. The current work is a considerably extended and revised version of the abstract presented at LOPSTR'06 [14].

## 2 Preliminaries

Mercury is a statically typed logic programming language [13]. Its type system is based on polymorphic many-sorted logic and essentially equivalent to the Mycroft-O'Keefe type system [15]. A type definition defines a possibly polymorphic type by giving the set of function symbols to which variables of that type may be bound as well as the type of the arguments of those functors [13]. Take for example the definition of the well known polymorphic type $list(T)$:

```
:- type list(T) ---> [] ; [T|list(T)].
```

According to this definition, if T is a type representing a given set of terms, values of type `list(T)` are either the empty list `[]` or a term $[t_1|t_2]$ where $t_1$ is of type $T$ and $t_2$ of type $list(T)$.

In addition to these so-called *algebraic types*, Mercury defines a number of primitive types that are builtin in the system. Among these are the *numeric types* `int` (integers) and `float` (floating point numbers). Mercury programs are

statically typed: the programmer declares the type of every argument of every predicate and from this information the compiler infers the type of every local variable and verifies that the program is well-typed.

In addition, the Mercury *mode system* describes how the instantiation of a variable changes over the execution of a goal. Each predicate argument is classified as either input (ground term before and after a call) or output (free variable at the time of the call that will be instantiated to a ground term). A predicate may have more than one mode, each mode representing a particular usage of the predicate. Each such mode is called a *procedure* in Mercury terminology. Each procedure has a declared (or inferred) *determinism* stating the number of solutions it can generate and whether it can fail. Determinisms supported by Mercury include `det` (a call to the procedure will succeed exactly once), `semidet` (a call will either succeed once or fail), `multi` (a call will generate one or more solutions), and `nondet` (a call can either fail or generate one or more solutions)[1]. Let us consider for example the definition of the well-known `append/3` and `member/2` predicates. We provide two mode declarations for each predicate, reflecting their most common usages:

```
:- pred append (list(T), list(T), list(T)).
:- mode append(in, in, out) is det.
:- mode append(out, out, in) is multi.
append([], Y, Y).
append([E|Es], Y ,[E|Zs]):- append(Es, Y, Zs).

:- pred member(T, list(T)).
:- mode member(in, in) is semidet.
:- mode member(out, in) is nondet.
member(X, [X|_]).
member(X, [Y|T]) :- not (X=Y), member(X, T).
```

For `append/3`, either the first two arguments are input and the third one is output in which case the call is deterministic (it will succeed exactly once), or the third argument is input and the first two are output in which case the call may generate multiple solutions. Note that no call to `append/3` in either of these modes can fail. For `member/2`, either both arguments are input and the call will either succeed once or fail, or only the second argument is input, in which case the call can fail, or generate one or more solutions.

## 3 Extracting Execution Paths

### 3.1 A Control Flow Graph for Mercury

For convenience, we consider that a Mercury program consists of a set of distinct procedures (a multi-moded predicate should have been transformed into different procedures). Each procedure should be well-typed and well-moded, and be in *superhomogeneous form*. A procedure in superhomogeneous form consists

[1] There exist other modes and determinisms but they are outside the scope of this paper; we refer to [13] for details

of a single clause (usually a disjunction) in which the arguments in the head of the clause and in procedure calls in the body are all distinct variables. Explicit unifications are generated for these variables in the body, and complex unifications are broken down into simple ones. Moreover, using mode information each unification is classified as either a test between two atomic values $X$==$Y$, an assignement $Z$:=$X$, deconstruction $X \Rightarrow f(Y_1, ..., Y_n)$ or construction $X \Leftarrow f(Y_1, ..., Y_n)$. See [13] for further details. We associate a distinct *label* to a number of program points of interest. These labels – which are written in subscripts and attached to the left and/or the right of a goal – are intended to identify the nodes of the program's control flow graph.

**Definition 1.** *Let $\Pi$ denote the set of procedures symbols, $\Sigma$ the set of function symbols and $\mathcal{V}$ and $\mathcal{L}$ respectively the set of variables and labels in a given program P. The syntax of a program in labelled superhomogenous form is defined as follows:*

$$
\begin{array}{lll}
LProc & ::= & p(X_1, \ldots, X_k) \ \texttt{:-}\ LConj. \\
LConj\ C & ::= & {}_lG_{l'} \mid {}_lG\texttt{,}C \\
LDisj\ D & ::= & C\texttt{;}C'' \mid D\texttt{;}C \\
LGoal\ G & ::= & A \mid D \mid \texttt{not}(C) \mid \texttt{if}\ C\ \texttt{then}\ C'\ \texttt{else}\ C'' \\
Atom\ A & ::= & X\texttt{==}Y \mid X \Rightarrow f(Y_1, ..., Y_n) \mid X \Leftarrow f(Y_1, ..., Y_n) \\
 & & \mid\ Z\texttt{:=}X \mid p(X_1, ..., X_n)
\end{array}
$$

*where $X, Y, Z$ and $X_i, Y_i (0 \leq i \leq n) \in \mathcal{V}, p/k \in \Pi, f \in \Sigma, l, l' \in \mathcal{L}$. All labels within a given program are assumed to be distinct.*

Note that according to the definition above, a label is placed between two successive conjuncts, as well as at the beginning and at the end of a conjunction and a disjunction.

*Example 1.* The `append(in,in,out)`, `member(in,in)` and `member(out,in)` procedures in labelled superhomogeneous form look as follows. Note that the only difference between the two procedures for `member` is the use of test, respectively an assignment at program point $l_3$.

$$
\begin{array}{l}
\texttt{append(X :: in, Y :: in, Z :: out)} : - \\
\quad {}_{l_1}({}_{l_2}\texttt{X} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}]_{,l_3}\ \texttt{append(E}_\texttt{s}\texttt{, Y, W)}_{,l_4}\ \texttt{Z} \Leftarrow [\texttt{E}|\texttt{W}]_{l_5}\ \texttt{;}\ {}_{l_6}\texttt{Z} = \texttt{Y}_{l_7})_{l_8}.
\end{array}
$$

$$
\begin{array}{l}
\texttt{member(X :: in, Y :: in)} : - \\
\quad {}_{l_1}\texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}]_{,l_2}\ ({}_{l_3}\texttt{X} == \texttt{E}_{l_4}\ \texttt{;}\ {}_{l_5}\texttt{member(X, E}_\texttt{s})_{l_6})_{l_7}.
\end{array}
$$

$$
\begin{array}{l}
\texttt{member(X :: out, Y :: in)} : - \\
\quad {}_{l_1}\texttt{Y} \Rightarrow [\texttt{E}|\texttt{E}_\texttt{s}]_{,l_2}\ ({}_{l_3}\texttt{X} := \texttt{E}_{l_4}\ \texttt{;}\ {}_{l_5}\texttt{member(X, E}_\texttt{s})_{l_6})_{l_7}.
\end{array}
$$

In [16], we have defined how one can build and use a control flow graph for Mercury. Given a Mercury program in labelled superhomogeneous form, its control graph can easily be constructed as follows. The nodes of the directed graph are the labels occuring in the program, together with two special labels: a *success label* $l_S$ and a *failure label* $l_F$, representing respectively success and failure of a (partial) derivation. The graph contains three types of arcs:

- Two nodes $l$ and $l'$ are linked with a *regular arc* if one of the following conditions holds:
  1. $l$ is the label preceding and $l'$ the label succeeding an atom;
  2. $l$ is the label preceding an atom that can possibly fail (i.e. deconstruction or equality test) and which is not part of the condition of a *if-then-else* construction and $l'$ is the failure label $l_F$;
  3. $l$ is the label preceding an atom that can possibly fail in the condition of a *if-then-else* construction and $l'$ is the first label of the goal in the *else* part of this construction;
  4. (a) $l$ is the label preceding a disjunction and $l'$ is the label preceding one of the disjuncts inside this disjunction; or
     (b) $l$ is the label succeeding one of the disjuncts inside a disjunction and $l'$ is the label succeeding the disjunction as a whole;
  5. $l$ is the label preceding a procedure call and $l'$ is the first label of this procedure's body goal;
  6. $l$ is the last label of the labelled program and $l'$ is the success label $l_S$.
- Two nodes $l$ and $l'$ are linked with a *return-after-success* or *return-after-failure* arc, denoted $(l, l')^{rs}$ respectively $(l, l')^{rf}$, if $l$ precedes a procedure call and if the execution should be resumed at $l'$ upon success, respectively failure, of the call.

Moreover, regular arcs are annotated by a natural number called *priority*. Each arc initiating a disjunct is annotated by the *position* of the disjunct in the disjunction when counted from right to left. Other arcs are annotated by zero.

*Example 2.* Figure 1. depicts two control flow graphs. The left one corresponds to a program defining the `member(in,in)` procedure, the right one defining the `member(out,in)` procedure, as defined in Example 1.

In both graphs, the arc $(l_1, l_2)$ represents success of the atom $Y \Rightarrow [E|E_s]$ whereas the arc $(l_1, l_F)$ represents failure of the atom. In the first case, the execution continues at $l_2$, in the latter it fails. The only difference between both graphs is the presence of the arc $(l_3, l_F)$ in the graph for `member(in,in)`; it represents the fact that the atom at $l_3$ (the test $X$==$E$) can fail whereas the assignement $X$:=$E$ in `member(out,in)` cannot. In order to avoid overloading the figures, we

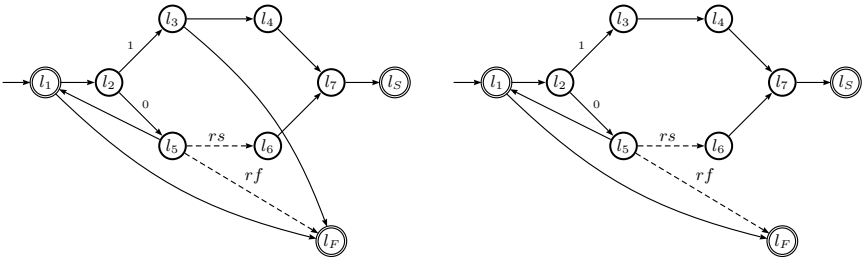



**Fig. 1.** `member(in,in)` and `member(out,in)`

depict priorities only when relevant, i.e. when they annotate an arc representing the entry into a disjunct. As such, a node from which leave several arcs bearing different priorities represents effectively a *choicepoint*. When walking through the graph in order to collect execution paths, the highest priority must be chosen first. This denotes the fact that, operationally, in a disjunction $(D_1; D_2)$ the disjunct $D_1$ is executed first, and $D_2$ is executed only if a backtracking occurs.

### 3.2 Deriving Execution Sequences

A program's control flow graph allows to reason about *all* possible executions of a given procedure. We first define the notion of a *complete execution segment* that represents a straightforward derivation from a call to a success (and thus the production of an answer) or a failure, in which an arbitrary disjunct is chosen at each encountered choicepoint. The definition is in two parts:

**Definition 2.** *An* execution segment *for a procedure $p$ is a sequence of labels with the first label being the label appearing at the very beginning of $p$'s body, the last label either the success label $l_S$ or the failure label $l_F$, where for each pair of consecutive labels $(l_i, l_{i+1})$ the following conditions hold:*

1. *If $l_i \neq l_S$ and $l_i \neq l_F$ then $l_i$ is connected to $l_{i+1}$ in the graph with a regular arc.*
2. *If $l_i = l_S$ then there exists $l_c$ $(c < i)$ such that $l_c$ and $l_{i+1}$ are connected in the graph with a* return-after-success *arc, and the sequence $\langle l_{c+1}, ..., l_i \rangle$ is itself an execution segment;*
3. *If $l_i = l_F$ then there exists $l_c$ $(c < i)$ such that $l_c$ and $l_{i+1}$ are connected in the graph with a* return-after-success *arc, the sequence $\langle l_{c+1}, ..., l_i \rangle$ is itself an execution segment* and *each pair of consecutive labels $(l_j, l_{j+1})$ with $c + 1 \leq j \leq i$ is connected in the graph with a regular arc of which priority equals zero;*

Definition 2 basically states that an execution segment is a path trough the graph in which a label $l_{i+1}$ follows a label $l_i$ if both labels are connected by a regular arc (condition (1)). If, however, $l_i$ represents the exit from a procedure call – either by success (condition (2)) or failure (condition (3)) – then the next label should be a valid resume point. Moreover, conditions 2 and 3 impose that each return has a corresponding call, and guarantee that the sequence of labels representing the execution through the callee is a valid execution segment as well. Condition 3 also denotes that the return after the *failure* of a call can be performed only if the corresponding call definitely failed, i.e. it is impossible to perform backtracking to a choicepoint created after the call that would make the latter succeed. In order to be useful, an execution segment must be *complete*, intuitively meaning that there should be no calls without a corresponding return, unless the derivation ends in failure and contains unexplored alternatives for backtracking.

**Definition 3.** *An execution segment $S$ for a procedure $p$ is* complete *if the following conditions hold:*

1. *If $S$ ends with $l_S$, then no suffix of $S$ is an execution segment for any procedure of the program;*
2. *If $S$ ends with $l_F$ then if $S$ has a suffix $S'$ which is an execution segment for a procedure of the program, then $S'$ contains at least one pair of consecutive labels $(l_j, l_{j+1})$ connected in the graph by a regular arc annotated by $n \geq 1$.*

In the above definition, condition 1 guarantees that, in a derivation leading to a success, every call has a corresponding return, while condition 2 imposes that, in a derivation leading to a failure, if there exists a call with no corresponding return, it must be possible to backtrack inside this call. Note that Definitions 2 and 3 only allow for *finite* (complete) execution segments.

*Example 3.* Let us consider `member(in,in)`, defined in Example 1. The sequence of labels $s = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ represents a complete execution segment in the control flow graph depicted on the left in Figure 1. It corresponds to the execution of a call in which the deconstruction of the list succeeds, the first disjunct is chosen at the choicepoint $l_2$, and the equality test between the first element and the call's first argument also succeeds, leading to the success of the predicate. In other words, it represents a call `member(X,Y)` in which the element $X$ appears at the first position in the list $Y$.

*Example 4.* Let us now consider the nondeterministic `member(out,in)` procedure, also defined in Example 1. The sequence of labels $s = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ represents a complete execution segment in the control flow graph depicted on the right in Figure 1. The execution segment $s$ represents the execution leading to the first solution of a call `member(X,Y)` in which the list $Y$ is not empty.

Of particular interest are the choices commited at each choicepoint encountered along a given complete execution segment. In the remaining we represent these choices by a sequence of integers, which are the priorities of the arcs chosen at each choicepoint.

**Definition 4.** *Let $s = \langle l_1, ..., l_n \rangle$ be a complete execution segment. The* sequence of choices *associated to $s$, noted $\mathcal{SC}(s)$, is defined as follows:*
$\mathcal{SC}(\langle l_1 \rangle) = \langle\rangle$
$\mathcal{SC}(\langle l_1, ..., l_n \rangle) = \mathcal{P}rior(l_1, l_2) \cdot \mathcal{SC}(\langle l_2, ..., l_n \rangle)$
*where $\cdot$ denotes sequence concatenation and $\mathcal{P}rior(l_i, l_{i+1}) = \langle nb \rangle$ if $l_i$ is a choicepoint and $l_i$ and $l_{i+1}$ are connected in the graph with a regular arc annotated by a number $nb$, or $\langle\rangle$ if $l_i$ is not a choicepoint.*

*Example 5.* Let us consider the complete execution segment $s = \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle$ for `member(in,in)`, defined in Example 3. The sequence of choices associated to this segment is $\mathcal{SC}(s) = \langle 1 \rangle$. On the other hand, for the complete execution segment $s' = \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle$, representing an execution in which the first argument of the call to member occurs in the second position of its second argument, we have $\mathcal{SC}(s') = \langle 0, 1 \rangle$.

A complete execution segment for a procedure $p$ represents a single derivation for a call to $p$ with respect to some (unknown) input values in which for each

encountered choicepoint an arbitrary choice is made. In order to model a real execution of the procedure, several such derivations need in general to be combined, in the right order. The order between two complete execution segments is determined by the sequence of choices that have been made. The sequence of choices being a sequence over natural numbers, we define the following operation:

**Definition 5.** *Let* $\langle i_1, ..., i_m \rangle$ *denote a sequence over* $\mathbb{N}$*, we define*

$$decr(\langle i_1, ..., i_m \rangle) = \begin{cases} \langle i_1, ..., (i_m - 1) \rangle & \text{if } i_m > 0 \\ decr(\langle i_1, ..., i_{m-1} \rangle) & \text{otherwise} \end{cases}$$

For a sequence of choices $s = \langle i_1, \ldots, i_n \rangle$, $decr(s)$ represents a new sequence of choices that is obtained from $s$ by deleting the rightmost zeros and decrementing the rightmost non-zero choice by one. Operationally, $decr(s)$ represents the stack after performing a backtrack operation.

**Definition 6.** *An* execution sequence *for a procedure p is defined as a sequence of complete execution segments* $\langle S_1, ..., S_n \rangle$ *for p having the following properties:*

1. *For all* $0 < i < n$, $decr(\mathcal{SC}(S_i))$ *is a prefix of* $\mathcal{SC}(S_{i+1})$*;*
2. *It is not possible to derive from the graph a complete execution segment* $S_k$ *for the procedure such that* $decr(\mathcal{SC}(S_k))$ *is a prefix of* $\mathcal{SC}(S_1)$*.*

Note that an execution sequence $\langle S_1, \ldots, S_n \rangle$ represents a derivation *tree* for a call to the predicate under consideration with respect to some (unknown) input values. Indeed, the first segment $S_1$ represents the first branch, i.e. the derivation in which for each encountered choicepoint the first alternative is chosen (the one having the highest priority in the graph). Likewise, an intermediate segment $S_{i+1}$ $(i \geq 1)$, represents the same derivation as $S_i$ except that at the last choicepoint having an unexplored alternative, the next alternative is chosen. Note that the derivation tree represented by $\langle S_1, \ldots, S_n \rangle$ is not necessarily complete. Indeed, the last segment $S_n$ might contain choicepoints having unexplored alternatives. However, by construction, there doesn't exist a complete execution segment representing an unexplored alternative between two consecutive segments $S_i$ and $S_{i+1}$.

While the definition allows in principle to consider infinite execution sequences, an execution sequence cannot contain an infinite segment, nor can it contain a segment representing a derivation in which one of the choicepoints has a previous alternative that would have led to an infinite derivation. It follows that an execution sequence represents a finite part of a real execution of the Mercury procedure under consideration (always with respect to the particular but unknown input values). The attentive reader will notice that if $\mathcal{SC}(S_n)$ is a sequence composed of all zeros, then the execution sequence represents a complete execution in which all answers for the call have been computed.

*Example 6.* Reconsider the nondeterministic `member(out,in)` procedure and the following complete execution segments:

$$\begin{aligned} S_1 &= \langle l_1, l_2, l_3, l_4, l_7, l_S \rangle, \\ S_2 &= \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle, \\ S_3 &= \langle l_1, l_2, l_5, l_1, l_2, l_5, l_1, l_F, l_F, l_F \rangle \rangle \end{aligned}$$

The reader can easily verify that $\mathcal{SC}(S_1) = \langle 1 \rangle$, $\mathcal{SC}(S_2) = \langle 0, 1 \rangle$, and $\mathcal{SC}(S_3) = \langle 0, 0, 1 \rangle$. Obviously, $decr(\mathcal{SC}(S_1)) = \langle 0 \rangle$ is a prefix of $\mathcal{SC}(S_2)$ and $decr(\mathcal{SC}(S_2)) = \langle 0, 0 \rangle$ is a prefix of $\mathcal{SC}(S_3)$. Moreover, there does not exist a complete execution segment $s$ such that $decr(s)$ is a prefix of $S_1$ and hence $\langle S_1, S_2, S_3 \rangle$ is an execution sequence for `member(out,in)`.

The execution sequence from Example 6 corresponds to the execution of a call $member(X, Y)$ in which a first solution is produced by assigning the first element of the list $Y$ to $X$ and returning from the call (expressed by the first segment of the path, ending in $l_S$). A second solution is produced by backtracking, choosing the disjunct corresponding to $l_5$, performing a recursive call, assigning the second element of the list to $X$, and performing the return (the second segment, also ending in $l_S$). The execution continues by backtracking and continuing at $l_5$ and performing a recursive call in which the deconstruction of the list argument fails. In other words, the execution sequence $e$ corresponds to a call to member in which the second argument is instantiated to a list containing exactly two elements.

In te remaining, we show how to compute input values from an execution sequence. Our approach consists of two phases. First, an execution sequence is translated into a set of constraints on the procedure's input (and output) arguments. Next, a solver written in CHR is used to generate arbitrary input values that satisfy the set of constraints. The solver contains type information from the program under test, but can be automatically generated from the program.

## 4 From Execution Sequences to Sets of Constraints

Since Mercury programs deal with both symbolic and numeric data, we consider two types of constraints: *symbolic constraints* which are either of the form $x = f(y_1, \ldots, y_n)$ or $x \not\Rightarrow f$[2] and *numerical constraints* which are of the form $x = y \oplus z$ (with $\oplus$ an arithmetic operator). Furthermore we consider constraints of the form $x = y$ and $x \neq y$ that can be either symbolic or numeric. Note that as a notational convenience, constraint variables are written in lowercase in order to distinguish them from the corresponding program variables.

In the remaining we assume that, in the control flow graph, edges originating from a label associated to an atom are annotated as follows: in case of a predicate call the edge is annotated by the call itself; in case of a unification it is annotated by the corresponding constraint, depending of the kind of atom and whether it succeeds or fails, as follows:

| source program | $(l, l')$ | $(l, l'')$ with $l' \neq l''$ |
|---|---|---|
| ${}_l X := Y_{l'}$ | $x = y$ | not applicable |
| ${}_l X == Y_{l'}$ | $x = y$ | $x \neq y$ |
| ${}_l X <= f(Y_1, \ldots, Y_n)_{l'}$ | $x = f(y_1, \ldots, y_n)$ | not applicable |
| ${}_l X => f(Y_1, \ldots, Y_n)_{l'}$ | $x = f(y_1, \ldots, y_n)$ | $x \not\Rightarrow f$ |
| ${}_l X := Y \oplus Z_{l'}$ | $x = y \oplus z$ | not applicable |

[2] The constraint $x \not\Rightarrow f$ denotes that the variable $x$ cannot be deconstructed into a term of which the functor is $f$. Formally, that means $\forall \overline{y} : \; x \neq f(\overline{y})$.

In order to collect the constraints associated to an execution segment, the basic idea is to walk the segment and collect the constraints associated to the corresponding edges. However, the constraints associated to each (sub)sequence of labels corresponding to the body of a call need to be appropriately renamed. Therefore, we keep a *sequence* of renamings during the constraint collection phase, initially containing a single renaming (possibly the identity renaming). Upon encountering an edge corresponding to a predicate call, a fresh variable renaming is constructed and added to the sequence. It is removed when the corresponding return edge is encountered. As such, this sequence of renamings can be seen as representing the call stack, containing one renaming for each call in a chain of (recursive) calls.

**Definition 7.** *Let $E$ denote the set of edges in a control flow graph and let $\langle l_1, \ldots, l_n \rangle$ be an execution segment for a procedure $p$. Given a sequence of renamings $\langle \sigma_1, \ldots, \sigma_k \rangle$, we define $\mathcal{U}(\langle l_1, \ldots, l_n \rangle, \langle \sigma_1, \ldots, \sigma_k \rangle)$ as the set of constraints $C$ defined as follows :*

1. *if $(l_1, l_2) \in E$ and $(l_1, l_v)^{rs} \notin E$ then let $c$ be the constraint associated to the edge $(l_1, l_2)$. We define $C = \{\sigma_1(c)\} \cup \mathcal{U}(\langle l_2, \ldots, l_n \rangle, \langle \sigma_1, \ldots, \sigma_k \rangle)$*
2. *if $(l_1, l_2) \in E$ and $(l_1, l_v)^{rs} \in E$ then let $p(X_1, \ldots, X_m)$ be the call associated to the edge $(l_1, l_2)$. If $head(p) = p(F_1, \ldots, F_m)$ then let $\gamma$ be a new renaming mapping $f_i$ to $x_i$ (for $1 \leq i \leq m$), and mapping every variable occuring free in $body(p)$ to a fresh variable. Then we define $C = \mathcal{U}(\langle l_2, \ldots, l_n \rangle, \langle \gamma, \sigma_1, \ldots, \sigma_k \rangle)$.*
3. *if $l_1 = l_S$ or $l_1 = l_F$, then we define $C = \mathcal{U}(\langle l_2, \ldots, l_n \rangle, \langle \sigma_2, \ldots, \sigma_k \rangle)$.*

*Furthermore, we define $\mathcal{U}(\langle \rangle, \langle \rangle) = \emptyset$.*

Note that the three cases in the definition above are mutually exclusive. The first case treats a success or failure edge associated to a unification. It collects the corresponding constraint, renamed using the *current* renaming (which is the first one in the sequence). The second case treats a success arc corresponding to a predicate call, by creating a fresh renaming $\gamma$ and collecting the constraints on the remaining part of the segment after adding $\gamma$ to the sequence of renamings. The third case, representing a return from a call, collects the remaining constraints after removing the current renaming from the sequence of renamings such that the remaining constraints are collected using the same renamings as those before the corresponding call.

*Example 7.* Let us reconsider the procedure `member(in,in)` and the execution segment $s' = \langle l_1, l_2, l_5, l_1, l_2, l_3, l_4, l_7, l_S, l_6, l_7, l_S \rangle$ given in Example 5. If we assume that *id* represents the identity renaming and that, when handling the recursive call at $l_5$, the constraint variables $e$ and $es$, corresponding to the local variables of `member`, are renamed into $e'$ and $es'$, we have

$$\mathcal{U}(s', \langle id \rangle) = \{y = [e|es], x \neq e, es = [e'|es'], x = e'\}.$$

As can be seen from Example 7, the set of constraints associated to an execution segment $s$ defines the *minimal* instantiation of the procedure's input variables so that the execution is guaranteed to proceed as specified by $s$. In case of Example 7 we have $y = [e, x|es] \wedge x \neq e$. Indeed, whatever (type correct) further instantiation we choose for the variables $x$, $e$ and $es$, as long as the above condition is satisfied, the execution of $member(x, y)$ is guaranteed to follow the execution segment $s$. A test input can thus be computed for a given execution segment by solving the associated set of constraints, further instantiating the free variables by arbitrary values, as long as the instantiation remains type correct.

To collect the constraints associated to an execution sequence, it suffices to collect the constraints associated to each individual execution segment using an appropriate initial renaming in order to avoid nameclashes.

**Definition 8.** *Let $\overline{S} = \langle s_1, \ldots, s_n \rangle$ denote an execution sequence for a procedure $p$. The* set of constraints *associated to $\overline{S}$, denoted $\mathcal{C}(\overline{S})$, is defined as*

$$\mathcal{C}(\langle s_1, \ldots, s_n \rangle) = \bigcup_{1 \leq i \leq n} \mathcal{U}(s_i, \sigma_i)$$

*where each $\sigma_i$ is a renaming mapping each non-input variable of $p$ to a fresh variable name.*

The initial renamings do not change the name of the procedure's input variables. Indeed, since each segment represents a different derivation for the *same* input values, *all* constraints on these values from the different segments must be satisfied.

*Example 8.* Let $\overline{S} = \langle S_1, S_2, S_3 \rangle$ be the execution sequence defined in Example 6 for the `member(out,in)` procedure defined in Section 2. Assuming that an initial renaming $\sigma_i$ simply adds the index $i$ to all concerned variables, and assuming that when handling the recursive call variables $e$ and $es$ are renamed into $e'$ and $es'$, one can easily verify that the set of constraints associated to $\overline{S}$ is as follows:

$$\begin{aligned}\mathcal{C}(\langle S_1, S_2, S_3 \rangle) &= \mathcal{U}(S_1, \sigma_1) \cup \mathcal{U}(S_2, \sigma 2) \cup \mathcal{U}(S_3, \sigma_3) \\ &= \{y = [e_1|es_1], x_1 = e_1\} \\ &\quad \cup \{y = [e_2|es_2], es_2 = [e'_2|es'_2], x_2 = e'_2\} \\ &\quad \cup \{y = [e_3|es_3], es_3 = [e'_3|es'_3], es'_3 \not\approx [\ |\ ]\}\end{aligned}$$

For a given execution sequence $\overline{S}$, $\mathcal{C}(\overline{S})$ defines the minimal instantiation of the procedure's input variables so that the execution is guaranteed to proceed as specified by $\mathcal{S}$. In Example 8 above, the set of constraints $\mathcal{C}(\langle S_1, S_2, S_3 \rangle)$ implies

$$y = [e_1, e'_2|es'_3] \wedge es'_3 \not\approx [\ |\ ] \wedge x_1 = e_1 \wedge x_2 = e'_2$$

and, indeed, whatever type correct instantiation we choose for the variables $e_1$ and $e'_2$, we will always have $es'_3$=[] and the execution of a call $member(\_, [E_1, E'_2])$ is guaranteed to proceed along the specified path.

Note that the obtained constraint set defines, for each segment ending in success, the minimal instantiation of the procedure's *output* arguments as well.

In Example 8, the sequence of output arguments is given by $\langle x_1, x_2 \rangle$. Hence, the computed results could be automatically converted not only into test inputs but into complete test cases. Of course, before such a computed test case can be recorded for further usage, the programmer should verify that the computed output corresponds with the *expected* output.

## 5 Constraint Solving

The constraints of a path are either *satisfiable* or *unsatisfiable*. The latter means that one or more labels in the path cannot be reached along the path (but may be reached along other paths). The latter means that solutions (one or more) exist, and that they will exercise the execution path. In order to establish the satisfiability, we take the usual constraint programming approach of interleaving *propagation* and *search*.

*Propagation* We reuse existing (CLP) constraints for most of our base constraints.

- $x = y$ and $x = f(\overline{y})$ are implemented as unification,
- $x \neq y$ is implemented as the standard Herbrand inequality constraint, known as `dif/2` in many Prolog systems, and
- $x = y \oplus z$ is implemented as the corresponding CLP(FD) constraint.

For $x \not\Rightarrow f$ we have our custom constraint propagation rules, implemented in CHR, based on the domain representation of CLP(FD). However, rather than maintaining a set of possible values for a variable, the domain of a variable is the set of possible function symbols. The initial domain are all the function symbols of the variable's type. For instance, the contraint `domain(X,{[]/0,[|]/2})` expresses that the possible functions symbol for variable `X` with type `list(T)` are `[]/0` and `[|]/2`, which is also its initial domain.

The following CHR rules further define the constraint propagators (and simplifiers) for the `domain/2` constraint:

```
domain(X,∅)       ==> fail.
domain(X,{F/A})   <=> functor(X,F,A).
domain(X,D)       <=> nonvar(X) | functor(X,F,A), F/A ∈ D.
domain(X,D1), domain(X,D2) <=> domain(X,D1 ∩ D2).
domain(X,D), X ↛ F/A       <=> domain(X,D \ {F/A}).
```

*Search Step* During search, we enumerate candidate values for the undetermined terms. From all undetermined terms, we choose one $x$ and create a branch in the search tree for each function symbol $f_i$ in its domain. In branch $i$, we add the constraint $x = f_i(\overline{y})$, where $\overline{y}$ are fresh undetermined terms. Subsequently, we exhaustively propagate again. Then either an (1) inconsistency is found, (2) all terms are determined or (3) some undetermined terms remain. In case (1) we must explore other branches, and in case (2) we have found a solution. In case (3) we simply repeat with another Search Step.

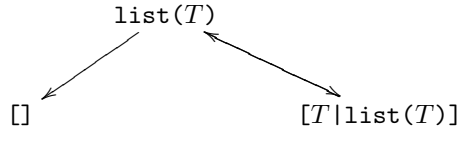
**Table 1.** Test input generation for `member(X::in,Y::in)` and `member(X::out,Y::in)`

| Test inputs | | Computed |
|---|---|---|
| `X::in` | `Y::in` | Result |
| 0 | [0] | Success |
| 0 | [1, 0] | Success |
| 0 | [1, 1, 0] | Success |
| 1 | [0, 0] | Failure |
| 1 | [0] | Failure |
| 0 | [] | Failure |

| Test input | Computed Result |
|---|---|
| `Y::in` | `X::out` |
| [0, 1, 2] | 0, 1, 2 |
| [0, 1] | 0, 1 |
| [0] | 0 |
| [] | – |

Our search algorithm visits the branches in depth-first order. Hence, we must make sure not to get caught in an infinitely diverging branch of the search tree, e.g. one that generates a list of unbounded length. For this purpose, we order a type's functions symbols according to the *type graph*. The nodes of a type graph consist of types and function symbols. There is an edge in the graph from each type to its function symbols, and from each function symbol to its argument types. We order the function symbols of a type according to the order in which they appear in a topologic ordering of the type graph.[3]

*Example 9.* Consider the list type `:- type list(T) --> [] ; [T|list(T)]`. Its type graph is:




A topological order is $\langle$`[]`, `list(`$T$`)`, `[`$T$`|list(`$T$`)]`$\rangle$, which yields the function symbol ordering $\langle$`[]/0`, `[|]/2`$\rangle$. Indeed, if we first try `[|]/2`, the search will diverge.

To conclude this section, note that our approach is the opposite of type inference, as we infer terms from types rather than types from terms. Among the vast amount of work on type inference we note [17] which employs a similar function symbol domain solver, for resolving ad-hoc overloading.

## 6 Implementation and Evaluation

The described approach for generating test inputs was implemented in Mercury. Our implementation first constructs a control flow graph for the program under test, and computes a set of execution sequences for the procedures in the program. To keep the set of execution sequences finite, we use a simple termination

[3] We assume that all types are well-founded, e.g. the type `:- type stream(T) --> cons(T,stream(T)).` is not a valid type.

**Table 2.** Test cases generation for `bubblesort(in,out)`

| Test input | Computed Result | Test input | Computed Result | Test input | Computed Result |
|---|---|---|---|---|---|
| List::in | Sorted::out | List::in | Sorted::out | List::in | Sorted::out |
| [] | [] | [2, 1, 0] | [0, 1, 2] | [1, 2, 1, 0] | [0, 1, 1, 2] |
| [0] | [0] | [0, 0, 0, 0] | [0, 0, 0, 0] | [1, 1, 0, 0] | [0, 0, 1, 1] |
| [0, 0] | [0, 0] | [0, 0, 1, 0] | [0, 0, 0, 1] | [2, 2, 1, 0] | [0, 1, 2, 2] |
| [1, 0] | [0, 1] | [0, 1, 1, 0] | [0, 0, 1, 1] | [1, 0, 0, 0] | [0, 0, 0, 1] |
| [0, 0, 0] | [0, 0, 0] | [1, 1, 1, 0] | [0, 1, 1, 1] | [2, 0, 1, 0] | [0, 0, 1, 2] |
| [0, 1, 0] | [0, 0, 1] | [1, 0, 1, 0] | [0, 0, 1, 1] | [2, 1, 1, 0] | [0, 1, 1, 2] |
| [1, 1, 0] | [0, 1, 1] | [0, 1, 0, 0] | [0, 0, 0, 1] | [2, 1, 0, 0] | [0, 0, 1, 2] |
| [1, 0, 0] | [0, 0, 1] | [0, 2, 1, 0] | [0, 0, 1, 2] | [3, 2, 1, 0] | [0, 1, 2, 3] |

**Table 3.** Test cases generation for different procedures

| Procedures | Determinism | Maximum call depth | Solutions requested | Number of test cases | Execution time (in $ms$) |
|---|---|---|---|---|---|
| Partition(in,in,out,out) | det | 6 | - | 126 | 890 |
| Append(in,in,out) | det | 6 | - | 6 | 40 |
| Append(out,out,in) | nondet | 6 | 10 | 10 | 70 |
| Doubleapp(in,in,in,out) | det | 3 | - | 6 | 650 |
| Doubleapp(out,out,out,in) | multi | 6 | 8 | 4 | 4670 |
| Member(in,in) | semidet | 5 | - | 12 | 700 |
| Member(out,in) | nondet | 5 | 5 | 6 | 1310 |
| Applast(in,in,out) | det | 3 | - | 14 | 40 |
| Match(in,in) | semidet | 3 | - | 6 | 960 |
| Matchappend(in,in) | semidet | 4 | - | 20 | 90 |
| MaxLength(in,out,out) | det | 5 | - | 10 | 600 |
| Revacctype(in,in,out) | det | 4 | - | 12 | 500 |
| Transpose(in,out) | det | 2 | - | 9 | 1370 |

scheme that consists in limiting the call depth as well as the the number of solutions in each execution sequence (in the case of non-deterministic procedures). Since our implementation is meant to be used as a proof of concept, performance of the tool has not been particularily stressed.

Table 1 gives the test inputs that are generated for the `member(in,in)` and `member(out,in)` procedures defined in Example 1 when the call depth is limited to 2 and the number of solutions in an execution sequence to 3. For `member(in,in)` we indicate for each generated test input whether this test input makes the procedure succeed or fail. For `member(out,in)` we give for each generated test input the corresponding output values.[4] As described in Section 4, it is up to the user to check whether the obtained result corresponds to the expected result when creating the test suite. The test inputs (and corresponding outputs) presented in Table 1 were generated in 20 ms , respectively 10 ms.

Table 2 contains the generated test inputs for a procedure implementing the bubble-sort algorithm. This well-know algorithm for list sorting uses two

[4] In the case of `member(out,in)`, we added manually the constraint `all_different/1` which guarantees all the elements of the list to be different.

recursive sub-procedures. Call depth was limited to 5, and for each test input we also give the computed output value. The test input generation took 1200 ms.

In Table 3, we present the behaviour of our implementation with different procedures, most of them have been chosen from the DPPD library [18]. For each of them, we indicate (1) the mode of the predicate, (2) its determinism, (3) the maximum call depth used, (4) the number of solutions requested (only in the case of non-deterministic and multi-deterministic procedures), (5) the number of test cases generated, and (6) the execution time of the test input generation, given in *ms*.

## 7 Conclusion and Future Work

In this work we have developed the necessary concepts and machinery for a control-flow based approach to the automatic generation of test inputs for structural testing of Mercury programs. Some parts of our approach, in particular the generation of execution sequences, have been deliberately left general, as it permits to develop algorithms for automatic test input generation that are parametrised with respect to a given coverage criterion or even a strategy for computing a series of "interesting" test inputs.

A side-effect of our approach is that not only test inputs are computed, but also the corresponding *outputs* (the fact that the predicate fails or succeeds and, in the latter case, what output values are produced). All the programmer has to do in order to construct a test suite is then to check whether the generated output corresponds to what is expected.

We have evaluated a prototype implementation that computes a finite set of execution sequences and the associated test inputs but makes no effort whatsoever to guarantee a certain degree of coverage. This is left as a topic for further research. Several improvements can be administered to our prototype in order to improve its performance. For example, one could integrate the constraint solving phase with the execution sequence generation in order to limit the number of generated sequences.

Other topics for further work include the development of a test evaluation framework, that would provide a mechanism for registering automatically generated (and possibly edited) test cases in a form that would allow a repeated evaluation of the test suite when changes in the source code occur (so-called regression testing).

## Acknowledgments

The authors would like to thank Baudouin Le Charlier for interesting and productive discussions on the subject of this paper. François Degrave is supported by F.R.I.A. - Fonds pour la formation à la Recherche dans l'Industrie et dans l'Agriculture. Tom Schrijvers is a post-doctoral researcher of the Fund for Scientific Research - Flanders.

## References


1. Glass, R.: Software runaways: lessons learned from massive software project failures. Prentice-Hall, Englewood Cliffs (1997)
2. Kaner, C., Falk, J., Nguyen, H.Q.: Testing computer software. John Wiley and Sons, Chichester (1993)
3. Zhu, H., Hall, P., May, J.: Software unit test coverage and adequacy. ACM Computing Surveys 29(4) (1997)
4. Fischer, S., Kuchen, H.: Systematic generation of glass-box test cases for functional logic programs. In: PPDP 2007: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming, pp. 63–74. ACM, New York (2007)
5. Claessen, K., Hughes, J.: Quickcheck: a lightweight tool for random testing of haskell programs. In: ICFP 2000: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp. 268–279. ACM, New York (2000)
6. Luo, G., von Bochmann, G., Sarikaya, B., Boyer, M.: Control-flow based testing of prolog programs. In: Proceedings of the Third International Symposium on Software Reliability Engineering, pp. 104–113 (1992)
7. Belli, B., Jack, O.: Implementation-based analysis and testing of prolog programs. In: ISSTA 1993: Proceedings of the 1993 ACM SIGSOFT international symposium on Software testing and analysis, pp. 70–80. ACM Press, New York (1993)
8. Sy, N.T., Deville, Y.: Automatic test data generation for programs with integer and float variables. In: Proceedings of ASE 2001 (2001)
9. Gotlieb, A., Botella, B., Rueher, M.: Automatic test data generation using constraint solving techniques. In: ISSTA 1998: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis, pp. 53–62. ACM Press, New York (1998)
10. Gupta, N., Mathur, A.P., Soffa, M.L.: Generating test data for branch coverage. In: Automated Software Engineering, pp. 219–228 (2000)
11. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java pathfinder. SIGSOFT Softw. Eng. Notes 29(4), 97–107 (2004)
12. Müller, R.A., Lembeck, C., Kuchen, H.: A symbolic java virtual machine for test case generation. In: IASTED International Conference on Software Engineering, part of the 22nd Multi-Conference on Applied Informatics, Innsbruck, Austria, February 17-19, pp. 365–371 (2004)
13. Somogyi, Z., Henderson, H., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. Journal of Logic Programming 29(4) (1997)
14. Mweze, N., Vanhoof, W.: Automatic generation of test inputs for Mercury programs (extended abstract). In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407. Springer, Heidelberg (2007)
15. Mycroft, A., O'Keefe, R.: A polymorphic type system for Prolog. Artificial Intelligence 23, 295–307 (1984)
16. Degrave, F., Vanhoof, W.: A control flow graph for Mercury. In: Proceedings of CICLOPS 2007 (2007)
17. Demoen, B., de la Banda, M.G., Stuckey, P.: Type constraint solving for parametric and ad-hoc polymorphism. In: Edwards, J. (ed.) The 22nd Australian Computer Science Conference, pp. 217–228. Springer, Heidelberg (1999)
18. Leuschel, M.: The dppd library of benchmarks, `http://www.ecs.soton.ac.uk/ mal/systems/dppd.html`

# Analytical Inductive Functional Programming*

Emanuel Kitzelmann

University of Bamberg, 96045 Bamberg, Germany
emanuel.kitzelmann@uni-bamberg.de
http://www.uni-bamberg.de/kogsys/members/kitzelmann/

**Abstract.** We describe a new method to induce functional programs from small sets of non-recursive equations representing a subset of their input-output behaviour. Classical attempts to construct functional Lisp programs from input/output-examples are *analytical*, i.e., a Lisp program belonging to a strongly restricted program class is algorithmically derived from examples. More recent approaches enumerate candidate programs and only *test* them against the examples until a program which correctly computes the examples is found. Theoretically, large program classes can be induced generate-and-test based, yet this approach suffers from combinatorial explosion. We propose a combination of search and analytical techniques. The method described in this paper is search based in order to avoid strong a-priori restrictions as imposed by the classical analytical approach. Yet candidate programs are computed based on analytical techniques from the examples instead of being generated independently from the examples. A prototypical implementation shows first that programs are inducible which are not in scope of classical purely analytical techniques and second that the induction times are shorter than in recent generate-and-test based methods.

## 1 Introduction

Synthesis of recursive declarative programs from input/output-examples (I/O-examples) is an active area of research since the seventies, though it has always been only a niche within other research fields such as machine learning, inductive logic programming (ILP) or functional programming.

The basic approach to inductive inference is to simply enumerate concepts—programs in this case—of a defined class until one is found which is consistent with the examples. When new examples appear and some of them contradict the current program then enumeration continues until a program consistent with the extended set of examples is found. Gold [1] stated this inductive inference model precisely and reported first results. Due to combinatorial explosion, this general enumerative approach is too expensive for practical use.

Summers [2] developed a completely different method to induce functional Lisp programs. He showed, first, that under certain conditions traces—expressions

* Research was supported by the German Research Community (DFG), grant SCHM 1239/6-1.

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 87–102, 2009.

computing a single input or a finite set of inputs correctly—can be computed from I/O-examples without search and, second, that for certain classes of LISP programs even a recursive program can be constructed without search in a program space from traces. It suffices to search for repetitive patterns in the traces from which a recursive function definition can be derived. Several variants and extensions of Summers method have been proposed. An early overview is given in [3], a recent extension is described in [4]. We call this approach *analytical*. However, due to strong constraints imposed to the forms of I/O-examples and inducible programs in order to avoid search, only relatively simple functions can be induced.

Recently, some papers on the induction of functional programs have been published in the functional programming community [5,6]. They all take a generate-and-test based approach. A further enumerative functional system is ADATE [7]. Though these systems use type information, some of them higher-order functions, and further more or less sophisticated techniques for pruning the search space, they strongly suffer from combinatorial explosion. Again only comparatively small problems can be solved with acceptable resources and in reasonable time.

Also ILP [8] has originated some methods intended for, or respectively capable of, inducing recursive programs on inductive datatypes [9,10,11,12], though ILP in general has a focus on classification and concept learning. In ILP one distinguishes top-down systems starting with an overly general theory and stepwise specialising it, e.g., FOIL [10], and bottom-up systems which search in the converse direction, e.g., GOLEM [9]. Typically, top-down systems are generate-and-test based whereas bottom-up systems contain analytical elements such as least generalisations [13]. Most often, clauses are constructed one after the other, called the *covering approach*, and single clauses are constructed by a greedy search. This might be adequate for classifier induction but has its drawbacks for inducing algorithmic concepts including recursion. Moreover, preference biases such as the *foil-gain* and also the most often used generality model *$\theta$-subsumption* are inadequate for the induction of recursive programs. This led to the development of ILP methods specialised for program induction, partly based on *inverse implication* instead of $\theta$-subsumption, as surveyed in [12]. Especially the idea of *sub-unification* [14] is similar to the analytical approach of inducing functional programs. The problems are similar to those of the functional approaches described above.

In this paper we suggest a particular combination of the analytical and the enumerative approach in order to put their relative strengths into effect and to repress their relative weaknesses. The general idea is to base the induction on a search in order to avoid strong a-priori restrictions on I/O-examples and inducible programs. But in contrast to the generate-and-test approach we construct successor programs during search by using analytical techniques similar to those proposed by Summers, instead of generating them independently from the examples. We represent functional programs as sets of recursive first-order equations. Adopting machine learning terminology, we call such generated equation sets *hypotheses*. The effect of constructing hypotheses w.r.t. example equations is that only hypotheses entailing the example equations are enumerated. In

contrast to greedy search methods, the search is still complete—only programs known to be incorrect w.r.t. the I/O-examples are ruled out. Besides narrowing the search space, analytical hypotheses construction has the advantage that the constructed hypotheses need not to be evaluated on the I/O-examples since they are correct by construction.

The rest of the paper is organised as follows: In the following section we shortly review Summers method to derive recursive function definitions from recurrence relations found in a trace. In Section 3 we introduce some basic terminology. In Section 4 we present a strong generalisation of Summers theorem. In Section 5 we describe an algorithm based on the generalised synthesis theorem. Section 6 shows results for some sample problems, and in Section 7, we conclude.

## 2 Summers Synthesis Theorem

Summers showed how linear-recursive LISP-programs can be derived from a finite set of I/O-examples $\{e_1, \ldots, e_k\}$ without search. The method consists of two steps: First, a trace in form of a McCarthy-Conditional with predicates and program fragments for each I/O-pair

$$F(x) = (p_1(x) \to f_1(x), \ldots, p_{k-1}(x) \to f_{k-1}(x), T \to f_k(x))$$

is constructed. Second, recurrences between predicates $p_i(x)$ and program fragments $f_i(x)$ are identified which are used to construct a recursive program.

*Example 1.* Consider the following I/O-examples for computing the initial sequence of a list:
$\langle (A), () \rangle, \langle (A, B), (A) \rangle, \langle (A, B, C), (A, B) \rangle, \langle (A, B, C, D), (A, B, C) \rangle$. From these, the following trace will be derived:

$$\begin{aligned} F(x) = (&atom(cdr(x)) \to nil \\ &atom(cddr(x)) \to cons(car(x), nil) \\ &atom(cdddr(x)) \to cons(car(x), cons(cadr(x), nil)) \\ &T \to cons(car(x), cons(cadr(x), cons(caddr(x), nil)))) \end{aligned}$$

The following recurrences are identified:

$$\begin{aligned} p_{i+1}(x) &= p_i(cdr(x)) \quad \text{for } i = 1, 2 \\ f_{i+1}(x) &= cons(car(x), f_i(cdr(x))) \quad \text{for } i = 1, 2, 3 \end{aligned}$$

The recurrences are inductively generalised to $i \in \mathbb{N}_+$ and the following recursive LISP-program results:

$$\begin{aligned} F(x) = (&atom(cdr(x)) \to nil \\ &T \to F'(x)) \end{aligned}$$

$$\begin{aligned} F'(x) = (&atom(cddr(x)) \to cons(car(x), nil) \\ &T \to cons(car(x), F'(cdr(x))) \end{aligned}$$

The first step, construction of traces, is based on strong constraints: First, traces are composed from a small fixed set of operation symbols, namely *cons*, *car*, *cdr* to uniquely compose and decompose S-expressions, the McCarthy-Conditional, and the predicate *atom* to check atomicity of S-expressions; second, each atom occurring in an example output must also occur in the corresponding example input; third, each atom may not occur more than once in an input; and fourth, the example inputs must be totally ordered according to a particular order over S-expressions. These restrictions, especially the first and the last one, are far too strong for practical use. In practice, programs make use of functions predefined in some library. This is not possible due to the first restriction. Moreover, comparing values regarding equality or order is not possible due to *atom* as the only allowed predicate. The last restriction forces the induced programs to be linear recursive. All classical methods surveyed in [3] as well as the recent powerful extension of Summers described in [4] suffer from these restrictions.

Our method described in Section 5 is not based on this first restrictive step of the classical methods and therefore we do not describe it here.

Now suppose that fragments and predicates for examples $\{e_1, \ldots, e_k\}$ can be expressed as recurrence relations with $n$ being a fixed number $> 0$.

$$\begin{aligned} &f_1(x), \ldots, f_n(x), f_{i+n}(x) = a(f_i(b(x)), x), \\ &p_1(x), \ldots, p_n(x), p_{i+n}(x) = p_i(b(x)) \qquad \text{for } 1 < i \leq k - n - 1 \end{aligned} \tag{1}$$

If $k - 1 > 2n$ then we *inductively infer* that the recurrence relations hold for all $i > 1$. Given such a generalised recurrence relation, more and more predicates and program fragments can be derived yielding a chain of approximations to the function specified by the generalised recurrence relations. The function itself is defined to be the supremum of that chain.

**Theorem 1 (Basic Synthesis).** *The function specified by the generalisation of recurrence relations as defined in* (1) *is computed by the recursive function*

$$F(x) = (p_1(x) \to f_1(x), \ldots, p_n(x) \to f_n(x), T \to a(F(b(x)), x))$$

The theorem is proved in [2]. It is easily extended to cases where the smallest index for the repetitive patterns is greater than 1 as in example 1 and for different basic functions $b$ for fragments and predicates.

*Remark.* Note that the recurrence detection method finds differences *between* fragments and predicates of respectively two I/O-examples. Therefore, the provided examples need to be complete in the sense that if one I/O-pair with an input $i$ of a particular complexity is given then all I/O-pairs with inputs of the intended domain smaller than $i$ according to the order over S-expressions also must be given. For example, if a function for computing lists of arbitrary length including the empty list shall be induced and one provides an example for an input list of length 4 then also examples for input lists of length 3, 2, 1, and 0 must be given. In [3], also methods to find recurrences *within* a program fragment of an I/O-pair are surveyed. For this methods theoretically one I/O-pair suffices to induce a recursive function. Yet such methods are again based on enumerative search in program space.

## 3 Equations and Term Rewriting

We shortly introduce concepts on terms and term rewriting here as it is described, e.g., in [15].

We represent I/O-examples and functional programs as sets of equations (pairs of terms) over a first-order signature $\Sigma$. The variables of a term $t$ are denoted by $Var(t)$. Signatures are many-sorted, i.e., terms are typed.

Each equation defining (amongst other equations) a function $F$ has a left-hand side (lhs) of the form $F(p_1, \ldots, p_n)$ where neither $F$ nor the name of any other of the defined functions occur in the $p_i$. Thus, the symbols in $\Sigma$ are divided into two disjoint subsets $\mathcal{D}$ of *defined function symbols* and $\mathcal{C}$ of (type) *constructors*. In the following we assume that all considered sets of equations have this form.

Terms without defined function symbols are called *constructor terms*. The constructor terms $p_i$ in the lhss of the equations for a defined function $F$ may contain variables and are called *pattern*. This corresponds to the concept of pattern matching in functional programming languages and is our only form of case distinction. Each variable in the right-hand side (rhs) of an equation must occur in the lhs, i.e., in the pattern. We say that rhs variables must be *bound* (by the lhs).

In order to evaluate a function defined by equations we read the equations as *simplification (or rewrite) rules* from left to right as known from functional programming, i.e., as a *term rewriting system (TRS)*. TRSs whose lhss have defined function symbols as roots and constructor terms as arguments, i.e., whose lhss have the described pattern-matching form, are called *constructor (term rewriting) systems (CSs)*. We use the terms *equation* and *rule* as well as *equation set* and *CS* interchangeably throughout the paper, depending on the context. Evaluating an $n$-ary function $F$ for an input $i_1, \ldots, i_n$ consists of repeatedly rewriting the term $F(i_1, \ldots, i_n)$ w.r.t. the rewrite relation implied by the CS until the term is in *normal form*, i.e., cannot be further rewritten. A sequence of (in)finitely many rewrite steps $t_0 \to_R t_1 \to_R \cdots$ is called *derivation*. If a derivation starts with term $t$ and results in a normal form $s$, then $s$ is called normal form of $t$, written $t \stackrel{!}{\to} s$. We say that $t$ normalises to $s$. In order to define a *function* on a domain (a set of ground terms) by a CS, no two derivations starting with the same ground term may lead to different normal forms, i.e., normal forms must be unique. A sufficient condition for this is that no two lhss of a CS unify; this is a sufficient condition for a CS to be *confluent*. A CS is *terminating* if each possible derivation reaches a normal form in finitely many rewrite steps. A sufficient condition for termination is that the arguments/inputs of recursive calls strictly decrease within each derivation and w.r.t. a well founded order.

A *substitution* $\sigma$ is a mapping from variables to terms and is uniquely extended to a mapping from terms to terms which is also denoted by $\sigma$ and written in postfix notation; $t\sigma$ is the result of applying $\sigma$ to all variables in term $t$. If $s = t\sigma$, then $t$ is called *generalisation* of $s$ and we say that $t$ *subsumes* $s$ and that $s$ *matches* $t$ by $\sigma$. Given two terms $t_1, t_2$ and a substitution $\sigma$ such that $t_1\sigma = t_2\sigma$, then we say that $t_1, t_2$ *unify*. Given a set of terms, $S = \{s, s', s'', \ldots\}$, then there exists a term $t$ which subsumes all terms in $S$ and which is itself

subsumed by each other term subsuming all terms in $S$. The term $t$ is called *least general generalisation (lgg)* of the terms in $S$ [13]. We also need lggs of *equations*. Therefor, consider an equation as a term with the equal sign as root symbol and the lhs and rhs as first and second subterm respectively. Then the lgg of a set of equations is the equation represented by the lgg of the terms representing the set of equations.

Each "node" in a term $t$ has a unique position $u$ denoted by $t|_u$.

## 4 A Generalised Synthesis Theorem

We call equations representing I/O-examples *example equations*. They have constructor terms as rhss, i.e., they are not allowed to contain function calls. Although example equations may contain variables, we call their lhss *(example) inputs* and their rhss *(example) outputs*.

**Theorem 2.** *Let $E$ be a set of example equations, $F$ a defined function symbol occurring in $E$, $p$ a pattern for $F$, and $E_{F,p} \subseteq E$ the example equations for $F$ whose lhss match $F(p)$ with substitutions $\sigma$.*

*If it exist a context $C$, a defined function symbol $F'$ (possibly $F' = F$) occurring in $E$, and a further defined function $G$ such that for every $F(i) = o \in E_{F,p}$ exist an equation $F'(i') = o' \in E$ and a substitution $\tau$ with*

$$o = C\sigma[o'\tau] \quad \textit{and} \quad G(i) \xrightarrow{!} i'\tau$$

*then*

$$(E \setminus E_{F,p}) \cup \{F(p) = C[F'(G(p))]\} \models E$$

*Proof.* Obviously it suffices to proof that $E_{F,p}$ is modelled. Let $F(i) = o$ be an equation in $E_{F,p}$ and $\sigma$ the substitution such that $F(i) = F(p)\sigma$. Then

$$F(i) = C[F'(G(p))]\sigma = C\sigma[F'(i'\tau)] = C\sigma[o'\tau] = o$$

□

The theorem states conditions under which example equations whose inputs match a lhs $F(p)$ can be replaced by one single equation with this lhs and a rhs containing recursive calls or calls to other defined functions $F'$. The theorem generalises Summers' theorem and its extensions (cp. [3]) in several aspects: The example inputs need not be totally ordered, differences between example outputs are not restricted to the function to be induced but may include other user-defined functions, so called *background functions*. The number of recursive calls or calls of other defined functions is not limited.

As in Summers recurrence detection method, (recursive) function calls are hypothesised by finding recurrent patterns *between* equations and not within equations. Therefore, the requirement for complete sets of example equations applies to our generalisation as well.

Note that the theorem is restricted to unary functions $F$. This is easily extended to functions $F$ with arity $> 1$.

## 5 The Igor2-Algorithm

In the following we write vectors $t_1, \ldots, t_n$ of terms abbreviated as $\boldsymbol{t}$.

Given a set of example equations $E$ for any number of defined function symbols and background equations $B$ describing the input-output behaviour of further user-defined functions on a suitable subset of their domains with constructor rhss each, the IGOR2 algorithm returns a set of equations $P$ constituting a confluent constructor system which is *correct* in the following sense:

$$F(\boldsymbol{i}) \xrightarrow{!}_{P \cup B} o \quad \text{for all } F(\boldsymbol{i}) = o \in E \tag{2}$$

The constructor system constituted by the induced equations together with the background equations rewrites each example input to its example output. The functions to be induced are called *target functions*.

### 5.1 Signature and Form of Induced Equations

Let $\Sigma$ be the signature of the example- and the background equations respectively. Then the signature of an induced program is $\Sigma \cup \mathcal{D}_\mathrm{A}$. $\mathcal{D}_\mathrm{A}$ is a set of defined function symbols not contained in the example- and background equations and not declared in their signatures. These defined function symbols are names of auxiliary, possibly recursive, functions dynamically introduced by IGOR2. This corresponds to predicate invention in inductive logic programming.

Auxiliary functions are restricted in two aspects: First, the type of an auxiliary function is identical with the type of the function calling it. That is, auxiliary functions have the same types as the target functions. IGOR2 cannot automatically infer auxiliary parameters as, e.g., the accumulator parameter in the efficient implementation of *Reverse*. Second, auxiliary function symbols cannot occur at the root of the rhs of another function calling it. Such restrictions are called *language bias*.

### 5.2 Overview over the Algorithm

Obviously, there are infinitely many correct solutions $P$ in the sense of (2), one of them $E$ itself. In order to select one or at least a finite subset of the possible solutions at all and particularly to select "good" solutions, IGOR2—like almost all inductive inference methods—is committed to a *preference bias*. IGOR2 prefers solutions $P$ whose patterns partition the example inputs in fewer subsets. The search for solutions is complete, i.e., solutions inducing the *least number* of subsets are found. Spoken in programming terminology: A functional program, fitting the language bias described above, with the least number of case distinctions correctly computing all specified examples is returned. This assures that the recursive structure in the examples as well as the computability through predefined functions is best possible covered in the induced program.

*Example 2.* From the type declarations

$$\begin{array}{ll} nil & : \rightarrow List \\ cons & : Elem\ List \rightarrow List \\ Reverse & : List \rightarrow List \\ Last & : List \rightarrow Elem \end{array}$$

the example equations[1]

$$\begin{array}{lll} 1. & Reverse([\,]) & = [\,] \\ 2. & Reverse([X]) & = [X] \\ 3. & Reverse([X,Y]) & = [Y,X] \\ 4. & Reverse([X,Y,Z]) & = [Z,Y,X] \\ 5. & Reverse([X,Y,Z,V]) & = [V,Z,Y,X] \end{array}$$

and the background equations

$$\begin{array}{ll} Last([X]) & = X \\ Last([X,Y]) & = Y \\ Last([X,Y,Z]) & = Z \\ Last([X,Y,Z,V]) & = V \end{array}$$

IGOR2 induces the following equations for *Reverse* and an auxiliary function *Init*[2]:

$$\begin{array}{ll} Reverse([\,]) & = [\,] \\ Reverse([X|Xs]) & = [Last([X|Xs])|Reverse(Init([X|Xs]))] \\ Init([X]) & = [\,] \\ Init([X_1,X_2|Xs]) & = [X_1|Init([X_2|Xs])] \end{array}$$

The induction of a terminating, confluent, correct CS is organised as a best first search, see Algorithm 1.

During search, a hypothesis is a set of equations entailing the example equations and constituting a terminating and confluent CS *but potentially with unbound variables in the rhss*. We call such equations and hypotheses containing them *unfinished* equations and hypotheses. A goal state is reached, if at least one of the best—according to the preference bias described above—hypotheses is finished, i.e., does not contain unfinished equations. Such a finished hypothesis is terminating and confluent by construction and since its equations entail the example equations, it is also correct.

W.r.t. the described preference bias and in order to get a complete hypothesis w.r.t. the examples, the initial hypothesis is a CS with one rule per target function such that its pattern subsumes all example inputs. In most cases (e.g., for all recursive functions) one rule is not enough and the rhss will remain unfinished. Then for one of the unfinished rules successors will be computed which

[1] We use a syntax for lists as known from PROLOG.

[2] If no background equations are provided, then IGOR2 also induces *Last* as auxiliary function.

**Algorithm 1.** The general IGOR2 algorithm

```
h ← the initial hypothesis (one rule per target function)
H ← {h}
while h unfinished do
    r ← an unfinished rule of h
    S ← all successor rule sets of r
    foreach h ∈ H with r ∈ h do
        remove h from H
        foreach successor set s ∈ S do
            add h with r replaced by s to H
    h ← a hypothesis from H with the minimal number of case distinctions
return H
```

leads to one or more hypotheses. Now repeatedly unfinished rules of currently best hypotheses are replaced until a currently best hypothesis is finished. Since one and the same rule may be member of different hypotheses, the successor rules originate successors of *all* hypotheses containing this rule. Hence, in each induction step several hypotheses are processed.

### 5.3 Initial Rules

Given a set of example equations for one target function, the initial rule is constructed by antiunifying [13] all example equations. This leads to the lgg of the example equations. In particular, the pattern of the initial rule is the most specific term subsuming all example inputs. Considering only lggs of example inputs as patterns narrows the search space. It does not constrain completeness of the search as shown by the following theorem.

**Theorem 3.** *Let $R$ be a CS with non-unifying but possibly not most specific patterns which is correct regarding a set of example equations. Then there exists a CS $R'$ such that $R'$ contains exactly one lhs $F(\boldsymbol{p'})$ for each lhs $F(\boldsymbol{p})$ in $R$, each $\boldsymbol{p'}$ being the lgg of all example inputs matching $\boldsymbol{p}$, and $R$ and $R'$ compute the same normal form for each example lhs.*

*Proof.* It suffices to show (i) that if a pattern $\boldsymbol{p}$ of a rule $r$ is replaced by the lgg of the example inputs matching $\boldsymbol{p}$, then also the rhs of $r$ can be replaced such that the rewrite relation remains the same for the example lhss matching $F(\boldsymbol{p})$, and (ii) that if the rhs of $r$ contains a call to a defined function then each instance of this call regarding the example inputs matching $\boldsymbol{p}$ is also an example lhs (subsumed by $F(\boldsymbol{p})$ or another lhs and, hence, also subsumed by lhss constituting lggs of example lhss). Proving these two conditions suffices because (i) assures equal rewrite relations for the example lhss and (ii) assures that each resulting term of one rewrite step which is not in normal form regarding $R$ is also not in normal form regarding $R'$.

The second condition is assured if the example equations are complete. To show the first condition let $F(\boldsymbol{p})$ be a lhs in $R$ such that $\boldsymbol{p}$ is *not* the lgg of the example inputs matching it. Then there exists a position $u$ with $\boldsymbol{p}|_u = X, X \in Var(\boldsymbol{p})$ and $\boldsymbol{p'}|_u = s \neq X$ if $\boldsymbol{p'}$ is the lgg of the example inputs matching $\boldsymbol{p}$.

First assume that $X$ does not occur in the rhs of the rule $r$ with pattern $\boldsymbol{p}$, then replacing $X$ by $s$ in $\boldsymbol{p}$ does not change the rewrite relation of $r$ for the example lhss because still all example lhss are subsumed and are rewritten to the same term as before. Now assume that $X$ occurs in the rhs of $r$. Then the rewrite relation of $r$ for the example lhss remains the same if $X$ is replaced by $s$ in $\boldsymbol{p}$ as well as in the rhs. □

### 5.4 Processing Unfinished Rules

This section describes the three methods for replacing unfinished rules by successor rules. All three methods are applied to a chosen unfinished rule in parallel. The first method, *splitting rules by pattern refinement*, replaces an unfinished rule with pattern $\boldsymbol{p}$ by at least two new rules with more specific patterns in order to establish a case distinction. The second method, *introducing subfunctions*, generates new induction problems, i.e., new example equations, for those subterms of an unfinished rhs containing unbound variables. The third method, *introducing function calls*, implements the principle described in Sec. 4 in order to introduce recursive calls or calls to other defined functions. Other defined functions can be further target functions or background knowledge functions. Finding the arguments of such a function call is considered as new induction problem. The last two methods implement the capability to automatically find auxiliary subfunctions. We denote the set of example equations whose inputs match the lhs $F(\boldsymbol{p})$ of an unfinished rule by $E_{F,\boldsymbol{p}}$.

**Splitting Rules by Pattern Refinement.** The first method for generating successors of an unfinished rule is to replace its pattern $\boldsymbol{p}$ by a set of more specific patterns, such that the new patterns induce a partition of the example inputs in $E_{F,\boldsymbol{p}}$. This results in a *set* of new rules replacing the original rule and—from a programming point of view—establishes a case distinction. It has to be assured that no two of the new patterns unify. This is done as follows: First a position $u$ is chosen at which a variable stands in $\boldsymbol{p}$ and constructors stands in the subsumed example inputs. Since $\boldsymbol{p}$ is the lgg of the inputs, at least two inputs have different constructor symbols at position $u$. Then respectively all example inputs with the *same* constructor at position $u$ are taken into the same subset. This leads to a partition of the example equations. Finally, for each subset of equations the lgg is computed leading to a set of initial rules with refined patterns.

Possibly, different positions of variables in pattern $\boldsymbol{p}$ lead to different partitions. Then all partitions are generated. Since specialised patterns subsume fewer inputs, the number of unbound variables in the initial rhss (non-strictly) decreases with each refinement step. Eventually, if no correct hypothesis with fewer case distinctions exists, each example input is subsumed by itself such that the example equations are simply reproduced.

*Example 3.* Reconsider the example equations for *Reverse* in Example 2. The pattern of the initial rule is simply a variable $Q$, since the example inputs have no common root symbol. Hence, the unique position at which the pattern contains

a variable and the example inputs different constructors is the root position. The first example input consists of only the constant [ ] at the root position. All remaining example inputs have the constructor *cons* at that position. I.e., two subsets are induced, one containing the first example equation, the other containing all remaining example equations. The lggs of the example lhss of these two subsets are $Reverse([\,])$ and $Reverse([Q|Qs])$ resp. which are the lhss of the two successor rules.

**Introducing Subfunctions.** The second method to generate successor equations can be applied, if all example equations $F(\boldsymbol{i}) = o \in E_{F,\boldsymbol{p}}$ have the same constructor symbol $c$ as root of their rhss $o$. Let $c$ be of arity $m$ then the rhs of the unfinished rule is replaced by the term $c(S_1(\boldsymbol{p}), \ldots, S_m(\boldsymbol{p}))$ where each $S_i$ denotes a new defined (sub)function. This finishes the rule since all variables from the new rhs are contained in the lhs. Examples for the new subfunctions are abduced from the examples of the current function as follows: If the $o|_j$, $j \in \{1, \ldots, m\}$ denote the $j$th subterms of the example rhss $o$, then the equations $S_j(\boldsymbol{i}) = o|_j$ are the example equations of the new subfunction $S_j$. Thus, correct rules for $S_j$ compute the $j$th subterms of the rhss $o$ such that the term $c(S_1(\boldsymbol{p}), \ldots, S_m(\boldsymbol{p}))$ normalises to the rhss $o$.

*Example 4.* Reconsider the *Reverse* examples except the first one as they have been put into one subset in Example 3. The initial equation for this subset is:

$$Reverse([Q|Qs]) = [Q_2|Qs_2] \tag{3}$$

It is unfinished due two the two unbound variables in the rhs. Now the two unfinished subterms (consisting of exactly the two variables) are taken as new subproblems. This leads to two new example sets for two new help functions $S_1$ and $S_2$: $S_1([X]) = X,\ S_1([X,Y]) = Y,\ \ldots, S_2([X]) = [\,],\ S_2([X,Y]) = [X],\ \ldots$. The successor equation set for the unfinished equation contains three equations determined as follows: The original unfinished equation (3) is replaced by the finished equation $Reverse([Q|Qs]) = [S_1([Q|Qs] \mid S_2[Q|Qs]]$ and from the new example sets for $S_1$ and $S_2$ initial equations are derived.

**Introducing Function Calls.** The last method to generate successor sets for an unfinished rule with pattern $\boldsymbol{p}$ for a target function $F$ is to replace its rhs by a call to a defined function $F'$, i.e. by a term $F'(G_1(\boldsymbol{p}), \ldots, G_k(\boldsymbol{p}))$ (if $F'$ is of arity $k$). Each $G_j$, $j \in \{1, \ldots, k\}$ denotes a new introduced defined (sub)function. This finishes the rule, since now the rhs does not longer contain variables not contained in the lhs. In order to get a rule leading to a *correct* hypothesis, for each example equation $F(\boldsymbol{i}) = o$ of function $F$ whose input $\boldsymbol{i}$ matches $\boldsymbol{p}$ with substitution $\sigma$ must hold: $F'(G_1(\boldsymbol{p}), \ldots, G_k(\boldsymbol{p}))\sigma \xrightarrow{!} o$. This holds if for each example output $o$ an example equation $F'(\boldsymbol{i'}) = o'$ of function $F'$ exists such that $o = o'\tau$ for a substitution $\tau$ and $G_j(\boldsymbol{p})\sigma \xrightarrow{!} i'_j\tau$ for each $G_j$ and $i'_j$. Thus, if we find such example equations of $F'$, then we abduce example equations $G_j(\boldsymbol{i}) = i'_j\tau$ for the new subfunctions $G_j$ and induce them from these examples.

Provided, the final hypothesis is correct for $F'$ and all $G_j$ then it is also correct for $F$. In order to assure termination of the final hypothesis it must hold $\boldsymbol{i'} < \boldsymbol{i}$ according to any reduction order $<$ if the function call is recursive.[3]

*Example 5.* Consider the examples for the help function $S_2$ from Example 4 and the corresponding unfinished initial equation:

$$S_2([Q|Qs] = Qs_2 \tag{4}$$

The example outputs, $[\,], [X], \ldots$ of $S_2$ match the example outputs for *Reverse*. That is, the unfinished rhs $Qs_2$ can be replaced by a (recursive) call to the *Reverse*-function. The argument of the call must map the inputs $[X], [X, Y], \ldots$ of $S_2$ to the corresponding inputs $[\,], [X], \ldots$ of *Reverse*, i.e., a new help function $G$ with example equations $G([X]) = [\,],\ G([X, Y] = [X],\ \ldots$ will be introduced. The successor equation set for the unfinished equation (4) contains the finished equation $S_2([Q|Qs] = \mathit{Reverse}(G([Q|Qs]))$ and the initial equation for $G$.

Note that in further steps the example outputs for $S_1$ from Example 4 would match the example outputs of the predefined *Last*-function (Example 2) and $G$ from Example 5 becomes the invented *Init*-function of Example 2.

## 6 Experiments

We have implemented a prototype of IGOR2 in the reflective term-rewriting based programming language Maude [16]. The implementation includes an extension compared to the approach described in the previous section. Different variables within a pattern can be tested for equality. This establishes—besides pattern refinement—a second form of case distinction.

In Tab. 1 we have listed experimental results for sample problems. The first column lists the names for the induced target functions, the second the names of additionally specified background functions, the third the number of given examples (for target functions), the fourth the number of automatically introduced recursive subfunctions, the fifth the maximal number of calls to defined functions within one rule, and the sixth the times in seconds consumed by the induction. Note that the example equations contain variables if possible (except for the *Add*-function). The experiments were performed on a Pentium 4 with Linux and the Maude 2.3 interpreter.

All induced programs compute the intended functions with more or less "natural" definitions. *Length*, *Last*, *Reverse*, and *Member* are the well known functions on lists. *Reverse* has been specified twice, first with *Snoc* as background knowledge which inserts an element at the end of a list and second without background knowledge. The second case (see Example 2 for given data and computed solution), is an example for the capability of automatic subfunction introduction and nested calls of defined functions. *Odd* is a predicate and true for odd natural

[3] Assuring decreasing arguments is more complex if mutual recursion is allowed.

**Table 1.** Some inferred functions

| target functions | bk funs | #expl | #sub | #calls | times |
|---|---|---|---|---|---|
| *Length* | / | 3 | 0 | 1 | .012 |
| *Last* | / | 3 | 0 | 1 | .012 |
| *Odd* | / | 4 | 0 | 1 | .012 |
| *ShiftL* | / | 4 | 0 | 1 | .024 |
| *Reverse* | *Snoc* | 4 | 0 | 2 | .024 |
| *Even*, *Odd* | / | 3, 3 | 0 | 1 | .028 |
| *Mirror* | / | 4 | 0 | 2 | .036 |
| *Take* | / | 6 | 0 | 1 | .076 |
| *ShiftR* | / | 4 | 2 | 2 | .092 |
| *DelZeros* | / | 7 | 0 | 1 | .160 |
| *InsertionSort* | *Insert* | 5 | 0 | 2 | .160 |
| *PlayTennis* | / | 14 | 0 | 0 | .260 |
| *Add* | / | 9 | 0 | 1 | .264 |
| *Member* | / | 13 | 0 | 1 | .523 |
| *Reverse* | / | 4 | 2 | 3 | .790 |
| *QuickSort* | *Append*, $P_1$, $P_2$ | 6 | 0 | 5 | 63.271 |

numbers, false otherwise. The solution contains two base cases (one for 0, one for 1) and in the recursive case, the number is reduced by 2. In the case where both *Even* and *Odd* are specified as target functions, both functions of the solution contain one base case for 0 and a call to the other function reduced by 1 as the recursive case. I.e. the solution contains a mutual recursion. *ShiftL* shifts a list one position to the left and the first element becomes the last element of the result list, *ShiftR* does the opposite, i.e., shifts a list to the right such that the last element becomes the first one. The induced solution for *ShiftL* contains only the *ShiftL*-function itself and simply "bubbles" the first element position by position through the list, whereas the solution for *ShiftR* contains two automatically introduced subfunctions, namely again *Last* and *Init*, and conses the last element to the input list without the last element. *Mirror* mirrors a binary tree. *Take* keeps the first $n$ elements of a list and "deletes" the remaining elements. This is an example for a function with two parameters where both parameters are reduced within the recursive call. *DelZeros* deletes all zeros from a list of natural numbers. The solution contains two recursive equations. One for the case that the first element is a zero, the second one for all other cases. *InsertionSort* and *QuickSort* respectively are the well known sort algorithms. The five respectively six well chosen examples as well as the additional examples to specify the background functions are the absolute minimum to generate correct solutions. The solution for *InsertionSort* has been generated within a time that is not (much) higher as for the other problems, but when we gave a few more examples, the time to generate a solution explodes. The reason is, that all outputs of lists of the same length are equal such that many possibilities of matching the outputs in order to find recursive calls exist. The number of possible matches increases

exponentially with more examples. The comparatively very high induction time for *QuickSort* results from the many examples needed to specify the background functions and from the complex calling relation between the target function and the background functions. $P_1$ and $P_2$ are the functions computing the lists of smaller numbers and greater numbers respectively compared to the first element in the input list. For *Add* we have a similar problem. First of all, we have specified *Add* by ground equations such that more examples were needed as for a non-ground specification. Also for *Add* holds, that there are systematically equal outputs, since, e.g., $Add(2,2), Add(1,3)$ etc. are equal and due to commutativity. Finally, *PlayTennis* is an attribute vector concept learning example from Mitchell's machine learning text book [17]. The 14 training instances consist of four attributes. The five non-recursive rules learned by our approach are equivalent with the decision tree learned by ID3 which is shown on page 53 in the book. This is an example for the fact, that learning decision trees is a subproblem of inducing functional programs.

To get an idea of which problems cannot be solved consider again *QuickSort* and *InsertionSort*. If we would not have provided *Insert* and *Append* as background functions respectively then the algorithms could not have been induced. The reason is that *Insert* and *Append* occur at the root of the expressions for the recursive cases respectively. But, as mentioned in Sect. 5.1, such functions cannot be invented automatically.

In [18] we compare the performance of IGOR2 with the systems GOLEM and MAGICHASKELLER on a set of sample problems. IGOR2 finds correct solutions for more problems than the other systems and the induction times are better than those needed by MAGICHASKELLER.

## 7 Conclusion

We described a new method, called IGOR2, to induce functional programs represented as confluent and terminating constructor systems. IGOR2 is inspired by classical and recent analytical approaches to the fast induction of functional programs. One goal was to overcome the drawback that "pure" analytical approaches do not facilitate the use of background knowledge and have strongly restricted hypothesis languages and on the other side to keep the analytical approach as far as possible in order to be able to induce more complex functions in a reasonable amount of time. This has been done by applying a search in a more comprehensive hypothesis space but where the successor functions are data-driven and not generate-and-test based, such that the number of successors is more restricted and the hypothesis spaced is searched in a controlled manner. The search is complete and only favours hypotheses inducing fewer partitions.

Compared to classical "pure" analytical systems, IGOR2 is a substantial extension since the class of inducible programs is much larger. E.g., all of the sample programs from [4, page 448] can be induced by IGOR2 but only a fraction of the problems in Section 6 can be induced by the system described in [4]. Compared to ILP systems capable of inducing recursive functions and recent enumerative

functional methods like GOLEM, FOIL, and MAGICHASKELLER IGOR2 mostly performs better regarding inducability of programs and/or induction times.

We see several ways to improve IGOR2, partly by methods existing in one or the other current system. Some of the enumerative functional systems apply methods to detect semantical equivalent programs in order to prune the search space. These methods could also be integrated into IGOR2. MAGICHASKELLER is one of the few systems using higher-order functions like *Map*. Many implicitly recursive functions can be expressed without explicit recursion by using such paramorphisms leading to more compact programs which are faster to induce. Currently we further generalise our synthesis theorem in order to analytically find calls of such paramorphisms provided as background knowledge. Finally we look at possible heuristics to make the complete search more efficient.

The IGOR2 implementation as well as sample problem specifications can be obtained at `http://www.cogsys.wiai.uni-bamberg.de/effalip/download.html`.

## References


1. Gold, E.M.: Language identification in the limit. Information and Control 10(5), 447–474 (1967)
2. Summers, P.D.: A methodology for LISP program construction from examples. Journal of the ACM 24(1), 161–175 (1977)
3. Smith, D.R.: The synthesis of LISP programs from examples: A survey. In: Biermann, A.W., Guiho, G., Kodratoff, Y. (eds.) Automatic Program Construction Techniques, pp. 307–324. Macmillan, Basingstoke (1984)
4. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. Journal of Machine Learning Research 7, 429–454 (2006)
5. Katayama, S.: Systematic search for lambda expressions. In: van Eekelen, M.C.J.D. (ed.) Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Intellect, vol. 6, pp. 111–126 (2007)
6. Koopman, P.W.M., Plasmeijer, R.: Systematic synthesis of functions. In: Nilsson, H. (ed.) Revised Selected Papers from the Seventh Symposium on Trends in Functional Programming, TFP 2006, Intellect, vol. 7, pp. 35–54 (2007)
7. Olsson, J.R.: Inductive functional programming using incremental program transformation. Artificial Intelligence 74(1), 55–83 (1995)
8. Muggleton, S.H., Raedt, L.D.: Inductive logic programming: Theory and methods. Journal of Logic Programming 19-20, 629–679 (1994)
9. Muggleton, S.H., Feng, C.: Efficient induction of logic programs. In: Proceedings of the First Conference on Algorithmic Learning Theory, Tokyo, Japan, Ohmsha, pp. 368–381 (1990)
10. Quinlan, J.R., Cameron-Jones, R.M.: FOIL: A midterm report. In: Brazdil, P.B. (ed.) ECML 1993. LNCS, vol. 667, pp. 3–20. Springer, Heidelberg (1993)
11. Bergadano, F., Gunetti, D.: Inductive Logic Programming: From Machine Learning to Software Engineering. MIT Press, Cambridge (1995)
12. Flener, P., Yilmaz, S.: Inductive synthesis of recursive logic programs: Achievements and prospects. The Journal of Logic Programming 41(2-3), 141–195 (1999)
13. Plotkin, G.D.: A note on inductive generalization. Machine Intelligence 5, 153–163 (1970)

14. Lapointe, S., Matwin, S.: Sub-unification: a tool for efficient induction of recursive programs. In: ML 1992: Proceedings of the Ninth International Workshop on Machine Learning, pp. 273–281. Morgan Kaufmann Publishers Inc., San Francisco (1992)
15. Terese: Term Rewriting Systems. Cambridge Tracts in Theoretical Computer Science, vol. 55. Cambridge University Press, Cambridge (2003)
16. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: The maude 2.0 system. In: Nieuwenhuis, R. (ed.) RTA 2003. LNCS, vol. 2706, pp. 76–87. Springer, Heidelberg (2003)
17. Mitchell, T.M.: Machine Learning. McGraw-Hill Higher Education, New York (1997)
18. Hofmann, M., Kitzelmann, E., Schmid, U.: Analysis and evaluation of inductive programming systems in a higher-order framework. In: Dengel, A.R., Berns, K., Breuel, T.M., Bomarius, F., Roth-Berghofer, T.R. (eds.) KI 2008. LNCS, vol. 5243, pp. 78–86. Springer, Heidelberg (2008)

# The MEB and CEB Static Analysis for CSP Specifications*

Michael Leuschel[1], Marisa Llorens[2], Javier Oliver[2], Josep Silva[2], and Salvador Tamarit[2]

[1] Institut für Informatik, Heinrich-Heine-Universität Düsseldorf, Universitätsstrasse 1, D-40225 Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de
[2] Technical University of Valencia, Camino de Vera S/N, E-46022 Valencia, Spain
{mllorens,fjoliver,jsilva,stamarit}@dsic.upv.es

**Abstract.** This work presents a static analysis technique based on program slicing for CSP specifications. Given a particular event in a CSP specification, our technique allows us to know what parts of the specification must necessarily be executed before this event, and what parts of the specification could be executed before it in some execution. Our technique is based on a new data structure which extends the *Synchronized Control Flow Graph* (SCFG). We show that this new data structure improves the SCFG by taking into account the context in which processes are called and, thus, makes the slicing process more precise.

**Keywords:** Concurrent Programming, CSP, Program Slicing.

## 1 Introduction

The *Communicating Sequential Processes* (CSP) [6] language allows us to specify complex systems with multiple interacting processes. The study and transformation of such systems often implies different analyses (e.g., deadlock analysis [10], reliability analysis [7], refinement checking [15], etc.).

In this work we introduce a static analysis technique for CSP which is based on a well-known program comprehension technique called *program slicing* [17].

Program slicing is a method for decomposing programs by analyzing their data and control flow. Roughly speaking, a *program slice* consists of those parts of a program which are (potentially) related with the values computed at some program point and/or variable, referred to as a *slicing criterion*. Program slices are usually computed from a *Program Dependence Graph* (PDG) [4] that makes explicit both the data and control dependences for each operation in a program.

* This work has been partially supported by the EU (FEDER and FP7 research project 214158 DEPLOY), by DAAD (PPP D/06/12830); by the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and *Acción Integrada* HA2006-0008; by the Generalitat Valenciana under grant GVPRE/2008/001; and by the *Programa de Apoyo a la Investigación y Desarrollo* de la *Universidad Politécnica de Valencia*.

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 103–118, 2009.

Program dependences can be traversed backwards or forwards (from the slicing criterion), which is known as *backward* or *forward* slicing, respectively. Additionally, slices can be *dynamic* or *static*, depending on whether a concrete program's input is provided or not. A survey on slicing can be found, e.g., in [16].

Our technique allows us to extract the part of a CSP specification which is related to a given event (referred to as the slicing criterion) in the specification. This technique can be very useful to debug, understand, maintain and reuse specifications; but also as a preprocessing stage of other analyses and/or transformations in order to reduce the complexity of the CSP specification.

In particular, given an event in a specification, our technique allows us to extract those parts of the specification which must be executed before the specified event (thus they are an implicit precondition); and those parts of the specification which could, and could not, be executed before it.

*Example 1.* Consider the following specification[1] with three processes (`STUDENT`, `PARENT` and `COLLEGE`) executed in parallel and synchronized on common events. Process `STUDENT` represents the three-year academic courses of a student; process `PARENT` represents the parent of the student who gives her a present when she passes a course; and process `COLLEGE` represents the college who gives a prize to those students which finish without any fail.

```
MAIN = (STUDENT || PARENT) || COLLEGE
STUDENT = year1 → (pass → YEAR2 □ fail → STUDENT)
YEAR2 = year2 → (pass → YEAR3 □ fail → YEAR2)
YEAR3 = year3 → (pass → graduate → STOP □ fail → YEAR3)
PARENT = pass → present → PARENT
COLLEGE = fail → STOP □ pass → C1
C1 = fail → STOP □ pass → C2
C2 = fail → STOP □ pass → prize → STOP
```

In this specification, we are interested in determining what parts of the specification must be executed before the student fails in the second year, hence, we mark event `fail` of process `YEAR2` (thus the slicing criterion is (`YEAR2`, `fail`)). Our slicing technique automatically extracts the slice composed by the black parts. We can additionally be interested in knowing which parts could be executed before the same event. In this case, our technique adds to the slice the underscored parts because they could be executed (in some executions) before the marked event. Note that, in both cases, the slice produced could be made executable by replacing the removed parts by "`STOP`" or by "→ `STOP`" if the removed expression has a prefix.

[1] We refer those readers non familiarized with CSP syntax to Section 2 where we provide a brief introduction to CSP.

It should be clear that computing the minimum slice of an arbitrary CSP specification is a non-decidable problem. Consider for instance the following CSP specification:

```
MAIN = P ⊓ Q
P = X ; Q
Q = a → STOP
X = Infinite Process
```

together with the slicing criterion (Q, a). Determining whether $X$ does not belong to the slice implies determining that $X$ is an infinite process which is known to be an undecidable problem [17].

The main contributions of this work are the following:

- We define two new static analyses for CSP and propose algorithms for their implementation. Despite their clear usefulness we have not found these static analyses in the literature.
- We define the context-sensitive synchronized control flow graph and show its advantages over its predecessors. This is a new data structure able to represent all computations taking into account the context of process calls; and it is particularly interesting for slicing languages with explicit synchronization.
- We have implemented our technique in a prototype integrated in ProB [11,1,12]. Preliminary results are very encouraging.

The rest of the paper is organized as follows. In Section 2 we overview the CSP language and introduce some notation. In Section 3 we show that previous data structures used in program slicing are inaccurate in our context, and we introduce the *Context-sensitive Synchronized Control Flow Graph* (CSCFG) as a solution and discuss its advantages over its predecessors. Our slicing technique is presented in Section 4 where we introduce two algorithms to slice CSP specifications from their CSCFGs. Next, we discuss some related works in Section 5. In Section 6 we present our implementation, and, finally, Section 7 concludes.

## 2 Communicating Sequential Processes

In order to keep the paper self-contained, in the following we recall the syntax of our CSP specifications. We also introduce here some notation that will be used along the paper.

Figure 1 summarizes the syntax constructions used in our CSP specifications. A specification is a finite collection of definitions. The left-hand side of each definition is the name of a different process, which is defined in the right-hand side by means of an expression that can be a call to another process or a combination of the following operators:

| | | | |
|---|---|---|---|
| $S$ | $::= D_1 \ldots D_m$ | (entire specification) | *Domains* |
| $D$ | $::= P = \pi$ | (definition of a process) | $P, Q, R \ldots$ (processes) |
| $\pi$ | $::= Q$ | (process call) | $a, b, c \ldots$ (events) |
| | $\mid\ a \rightarrow \pi$ | (prefixing) | |
| | $\mid\ \pi_1 \sqcap \pi_2$ | (internal choice) | |
| | $\mid\ \pi_1 \Box \pi_2$ | (external choice) | |
| | $\mid\ \pi_1 \mid\mid\mid \pi_2$ | (interleaving) | |
| | $\mid\ \pi_1 \mid\mid_{\{\overline{a_n}\}} \pi_2$ | (synchronized parallelism) | where $\overline{a_n} = a_1, \ldots, a_n$ |
| | $\mid\ \pi_1 \ ; \ \pi_2$ | (sequential composition) | |
| | $\mid\ SKIP$ | (skip) | |
| | $\mid\ STOP$ | (stop) | |

**Fig. 1.** Syntax of CSP specifications

**Prefixing.** It specifies that event $a$ must happen before expression $\pi$.

**Internal choice.** The system chooses (e.g., nondeterministically) to execute one of the two expressions.

**External choice.** It is identic to internal choice but the choice comes from outside the system (e.g., the user).

**Interleaving.** Both expressions are executed in parallel and independently.

**Synchronized parallelism.** Both expressions are executed in parallel with a set of synchronized events. In absence of synchronizations both expressions can execute in any order. Whenever a synchronized event $a_i, 1 \leq i \leq n$, happens in one of the expressions it must also happen in the other at the same time. Whenever the set of synchronized events is not specified, it is assumed that expressions are synchronized in all common events.

**Sequential composition.** It specifies a sequence of two processes. When the first finishes, the second starts.

**Skip.** It finishes the current process. It allows us to continue the next sequential process.

**Stop.** It finishes the current process; but it does not allow the next sequential process to continue.

Note, to simplify the presentation we do not yet treat process parameters, nor input and output of data values on channels.

We define the following notation for a given CSP specification $\mathcal{S}$: $\mathcal{P}roc(\mathcal{S})$ and $\mathcal{E}vent(\mathcal{S})$ are, respectively, the sets of all (possible repeated) process calls and events appearing in $\mathcal{S}$. Similarly, $\mathcal{P}roc(P)$ and $\mathcal{E}vent(P)$ with $P \in \mathcal{S}$, are, respectively, the sets of all (possible repeated) processes and events in $P$. In addition, we define $choices(A)$, where $A$ is a set of processes, events and operators; as the subset of operators that are either an internal choice or an external choice.

We use $a_1 \rightarrow^* a_n$ to denote a feasible (sub)execution which leads from $a_1$ to $a_n$; and we say that $b \in (a_1 \rightarrow^* a_n)$ iff $b = a_i, 1 \leq i \leq n$. In the following, unless we state the contrary, we will assume that programs start the computation from a distinguished process `MAIN`.

We need to define the notion of *specification position* which, roughly speaking, is a label that identifies a part of the specification. Formally,

**Definition 1 (position, specification position).**
*Positions are represented by a sequence of natural numbers, where $\Lambda$ denotes the empty sequence (i.e., the root position). They are used to address subexpressions of an expression viewed as a tree:*

$$\pi|_\Lambda = \pi \qquad \text{for all process } \pi$$
$$(\pi_1 \ op \ \pi_2)|_{1.w} = \pi_1|_w \quad \text{for all operator } op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\}$$
$$(\pi_1 \ op \ \pi_2)|_{2.w} = \pi_2|_w \quad \text{for all operator } op \in \{\rightarrow, \sqcap, \Box, |||, ||, ;\}$$

*Given a specification $\mathcal{S}$, we let $\mathcal{P}os(\mathcal{S})$ denote the set of all specification positions in $\mathcal{S}$. A specification position is a pair $(P, w) \in \mathcal{P}os(\mathcal{S})$ that addresses the subexpression $\pi|_w$ in the right-hand side of the definition, $P = \pi$, of process $P$ in $\mathcal{S}$.*

*Example 2.* In the following specification each term has been labelled (in grey color) with its associated specification position so that all labels are unique.

$$\texttt{MAIN} = (\texttt{P}_{(Main,1.1)} ||_{\{b\}(Main,1)} \texttt{Q}_{(Main,1.2)}) ;_{(Main,\Lambda)} (\texttt{P}_{(Main,2.1)} ||_{\{a\}(Main,2)} \texttt{R}_{(Main,2.2)})$$

$$\texttt{P} = \texttt{a}_{(P,1)} \rightarrow_{(P,\Lambda)} \texttt{b}_{(P,2.1)} \rightarrow_{(P,2)} \texttt{SKIP}_{(P,2.2)}$$

$$\texttt{Q} = \texttt{b}_{(Q,1)} \rightarrow_{(Q,\Lambda)} \texttt{c}_{(Q,2.1)} \rightarrow_{(Q,2)} \texttt{SKIP}_{(Q,2.2)}$$

$$\texttt{R} = \texttt{d}_{(R,1)} \rightarrow_{(R,\Lambda)} \texttt{a}_{(R,2.1)} \rightarrow_{(R,2)} \texttt{SKIP}_{(R,2.2)}$$

We often use indistinguishably an expression and its specification position (e.g., $(Q, c)$ and $(Q, 2.1)$) when it is clear from the context.

## 3 Context-Sensitive Synchronized Control Flow Graphs

As it is usual in static analysis, we need a data structure able to finitely represent the (often infinite) computations of our specifications. Unfortunately, we cannot use the standard *Control Flow Graph* (CFG) [16], neither the *Interprocedural Control Flow Graph* (ICFG) [5] because they cannot represent multiple threads and, thus, they can only be used with sequential programs. In fact, for CSP specifications, being able to represent multiple threads is a necessary but not a sufficient condition. For instance, the *threaded Control Flow Graph* (tCFG) [8,9] can represent multiple threads through the use of the so called "*start thread*" and "*end thread*" nodes; but it does not handle synchronizations between threads. Callahan and Sublok introduced a data structure [2], the *Synchronized Control Flow Graph* (SCFG), which explicitly represents synchronizations between threads with a special edge for synchronization flows.

For convenience, the following definition adapts the original definition of SCFG for CSP; and, at the same time, it slightly extends it with the "*start thread*" and "*end thread*" notation from tCFGs.

**Definition 2.** *(Synchronized Control Flow Graph) Given a CSP specification $\mathcal{S}$, its* Synchronized Control Flow Graph *is a directed graph, $SCFG = (N, E_c, E_s)$ where nodes $N = \mathcal{P}os(\mathcal{S}) \cup \mathcal{S}tart(\mathcal{S})$; and $\mathcal{S}tart(\mathcal{S}) = \{\text{“}start\ P\text{”}, \text{“}end\ P\text{”} | P \in \mathcal{P}roc(\mathcal{S})\}$. Edges are divided into two groups,* control-flow edges *$(E_c)$ and* synchronization edges *$(E_s)$. $E_s$ is a set of two-way edges (denoted with $\leftrightarrow$) representing the possible synchronization of two (event) nodes.*[2] *$E_c$ is a set of one-way edges (denoted with $\mapsto$) such that, given two nodes $n, n' \in N$, $n \mapsto n' \in E_c$ iff one of the following is true:*

- $n = P \wedge n' = \text{“}start\ P\text{”}$ with $P \in \mathcal{P}roc(\mathcal{S})$
- $n = \text{“}start\ P\text{”} \wedge n' = first((P, \Lambda))$ with $P \in \mathcal{P}roc(\mathcal{S})$
- $n \in \{\sqcap, \Box, |||, ||\} \wedge n' \in \{first(n.1), first(n.2)\}$
- $n \in \{\rightarrow, ;\} \wedge n' = first(n.2)$
- $n = n'.1 \wedge n' = \rightarrow$
- $n \in last(n'.1) \wedge n' = \ ;$
- $n \in last((P, \Lambda)) \wedge n' = \text{“}end\ P\text{”}$ with $P \in \mathcal{P}roc(\mathcal{S})$

*where $first(n)$ is the initial node of the subprocess denoted by $n$:*

$$first(n) = \begin{cases} n.1 & \text{if } n = \rightarrow \\ first(n.1) & \text{if } n = \ ; \\ n & \text{otherwise} \end{cases}$$

*and where $last(n)$ is the set of possible termination points of the subprocess denoted by $n$:*

$$last(n) = \begin{cases} \{n\} & \text{if } n = SKIP \\ \emptyset & \text{if } n = STOP \vee \\ & (n \in \{|||, ||\} \wedge (last(n.1) = \emptyset \vee last(n.2) = \emptyset)) \\ last(n.2) & \text{if } n \in \{\rightarrow, ;\} \\ last(n.1) \cup last(n.2) & \text{if } n \in \{\sqcap, \Box\} \vee \\ & (n \in \{|||, ||\} \wedge last(n.1) \neq \emptyset \wedge last(n.2) \neq \emptyset) \\ \{\text{“}end\ P\text{”}\} & \text{if } n = P \end{cases}$$

The SCFG can be used for slicing CSP specifications as it is described in the following example.

*Example 3.* Consider the specification of Example 2 and its associated SCFG shown in Fig. 2 (left); for the sake of clarity we show the term represented by each specification position. If we select the node labelled `c` and traverse the SCFG backwards in order to identify the nodes on which `c` depends, we get the whole graph except nodes `end MAIN`, `end R` and `SKIP` at process `R`.

This example is twofold: on the one hand, it shows that the SCFG could be used for static slicing of CSP specifications. On the other hand, it shows that it is still too imprecise as to be used in practice. The cause of this imprecision is that the

[2] This will be computed using the technique from [14], but is left open in this definition.

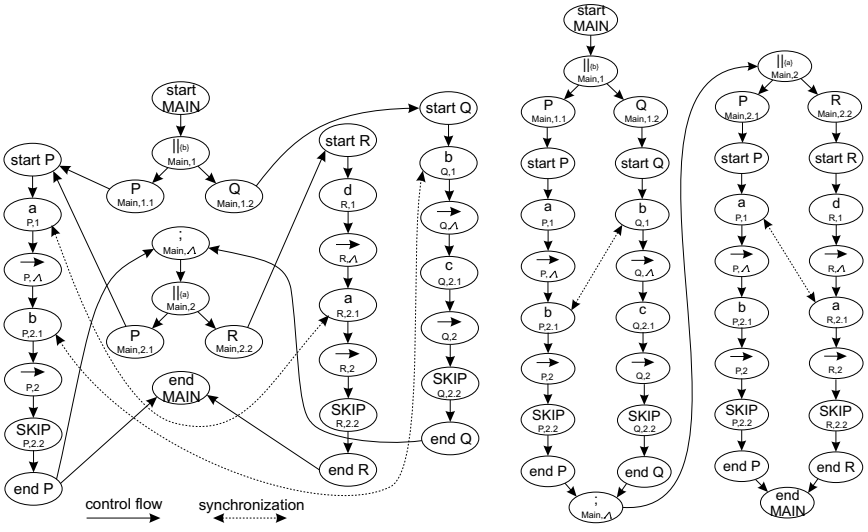



**Fig. 2.** SCFG (left) and CSCFG (right) of the program in Example 2

SCFG is context-insensitive because it connects all the calls to the same process with a unique set of nodes. This causes the SCFG to mix different executions of a process with possible different synchronizations, and, thus, slicing lacks precision. For instance, in Example 2 process `P` is called twice in different contexts. It is first executed in parallel with `Q` producing the synchronization of their `b` events. Then, it is executed in parallel with `R` producing the synchronization of their `a` events. This makes the complete process `R` be part of the slice, which is suboptimal because process `R` is always executed after `Q`.

To the best of our knowledge, there do not exist other graph representations which face this problem. In the rest of this section, we propose a new version of the SCFG, the context-sensitive synchronized control flow graph (CSCFG) which is context-sensitive because it takes into account the different contexts on which a process can be executed.

Contrarily to the SCFG, inside a CSCFG the same specification position can appear multiple times. Hence, we now use labelled graphs, with nodes labelled by specification positions. Therefore, we use $l(n)$ to refer to the label of node $n$. We also need to define the context of a node in the graph.

**Definition 3.** *(Context) A* path *between two nodes $n_1$ and $m$ is a sequence $l(n_1), \ldots, l(n_k)$ such that $n_k \mapsto m$ and for all $0 < i < k$ we have $n_i \mapsto n_{i+1}$ . The path is* loop-free *if for all $i \neq j$ we have $n_i \neq n_j$.*

*Given a labelled graph $G = (N, E_c)$ and a node $n \in N$, the context of $n$, $\mathcal{C}on(n) = \{m \mid l(m) =$ "start $P$", $P \in \mathcal{P}roc(\mathcal{S})$ and exists a loop-free path $\pi = m \mapsto^* n$ with "end $P$" $\notin \pi\}$.*

Intuitively speaking, the context of a node represents the set of processes in which a particular node is being executed. If we focus on a node $n \in \mathcal{P}roc(\mathcal{S})$ we can use the context to identify loops; i.e., we have a loop whenever "*start* $P$" $\in \mathcal{C}on(P)$.

**Definition 4.** *(Context-Sensitive Synchronized Control Flow Graph) Given a CSP specification $\mathcal{S}$, a* Context-Sensitive Synchronized Control Flow Graph *$CSCFG = (N, E_c, E_l, E_s)$ is a SCFG graph, except in two aspects:*

1. *There is a special set of* loop edges *($E_l$) denoted with $\rightsquigarrow$. $(n_1 \rightsquigarrow n_2) \in E_l$ iff $l(n_1) = P \in \mathcal{P}roc(\mathcal{S})$, $l(n_2) =$ "start $P$" and $n_2 \in \mathcal{C}on(n_1)$, and*
2. *There is no longer a one to one correspondence between nodes and labels. Every node in $\mathcal{S}tart(\mathcal{S})$ has one and only one incoming edge in $E_c$. Every process call node has one and only one outgoing edge which belongs to either $E_c$ or $E_l$.*

The CSCFG can be constructed in a way similar to the SCFG. Starting from `Main`, each process call is connected to a subtree which contains the right-hand side of the called process. However, in the CSCFG, each subtree is a different subgraph except if a loop is detected.

For slicing purposes, the CSCFG is interesting because we can use the edges to determine if a node must be executed or not before another node, thanks to the following properties:

- if $n \mapsto n' \in E_c$ then $n$ must be executed before $n'$ in all executions.
- if $n \rightsquigarrow n' \in E_l$ then $n'$ must be executed before $n$ in all executions.
- if $n \leftrightarrow n' \in E_s$ then, in all executions, if $n$ is executed there must be some $n''$ which is executed at the same time than $n$ with $n \leftrightarrow n'' \in E_s$.

The key difference between the SCFG and the CSCFG is that the latter unfolds every process call node except those that belong to a loop. This is very convenient for slicing because every process call which is executed in a different context is unfolded, thus, slicing does not mix computations. Moreover, it allows to deal with recursion and, at the same time, it prevents from infinite unfolding of process calls; because loop edges prevent from infinite unfolding. One important characteristic of the CSCFG is that loops are unfolded once, and thus all the specification positions inside the loops are in the graph and can be collected by slicing algorithms. For slicing purposes, this representation also ensures that every possibly executed part of the specification belongs to the CSCFG because only loops (i.e., repeated nodes) are missing.

The CSCFG provides a different representation for each context in which a procedure call is made. This can be seen in Fig. 2 (right) where process `P` appears twice to account for the two contexts in which it is called. In particular, in the CSCFG we have a fresh node to represent each different process call, and two nodes point to the same process if and only if they are the same call (they are labelled with the same specification position) and they belong to the same loop. This property ensures that the CSCFG is finite.

**Lemma 1.** *(Finiteness) Given a specification $\mathcal{S}$, its associated CSCFG is finite.*

*Proof.* We show first that there do not exist infinite unfolding in a CSCFG. Firstly, the same start process node only appears twice in the same control loop-free path if it belongs to a process which is called from different process calls

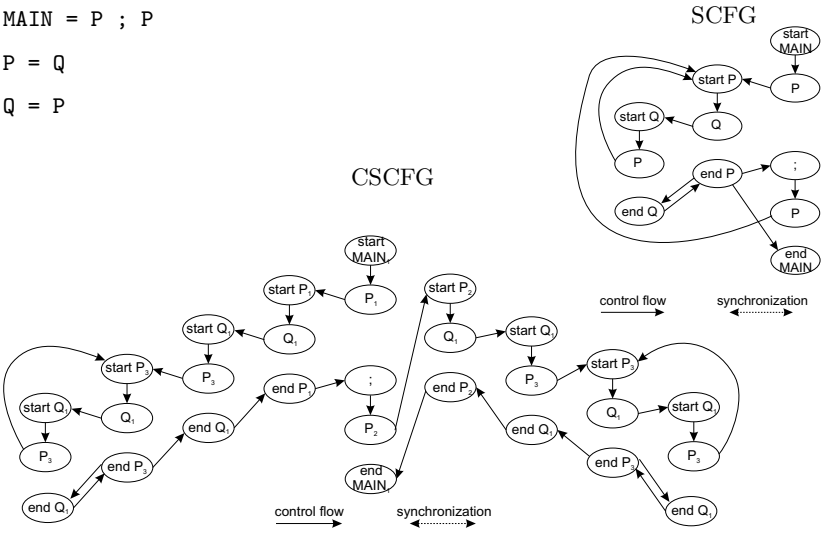



**Fig. 3.** SCFG and CSCFG representing an infinite computation

(i.e., with different specification positions) as it is ensured by the first condition of Definition 4. Therefore, the number of repeated nodes in the same control loop-free path is limited by the number of different process calls appearing in the program. However, the number of terms in the specification is finite and thus there is a finite number of different process calls. Moreover, every process call has only one outgoing edge as it is ensured by the second condition of Definition 4. Therefore, the number of paths is finite and the size of every path of the CSCFG is limited.

*Example 4.* The specification in Fig. 3 makes clear the difference between the SCFG and the CSCFG. While the SCFG only uses one representation for the process `P` (there is only one `start P`), the CSCFG uses four different representations because `P` could be executed in four different contexts. Note that due to the infinite loops, some parts of the graph are not reachable from `start MAIN`; i.e., there is not possible control flow to `end MAIN`. However, it does not hold in the SCFG.

## 4 Static Slicing of CSP Specifications

We want to perform two kinds of analyses. Given an event or a process in the specification, we want, on the one hand, to determine what parts of the specification MUST be executed before (MEB) it; and, on the other hand, we want to determine what parts of the specification COULD be executed before (CEB) it.

We can now formally define our notion of slicing criterion.

**Definition 5.** *(Slicing Criterion) Given a specification $\mathcal{S}$, a slicing criterion is a specification position $(P, w) \in \mathcal{P}roc(\mathcal{S}) \cup \mathcal{E}vent(\mathcal{S})$.*

Clearly, the slicing criterion points to a set of nodes in the CSCFG, because the same event or process can happen in different contexts and, thus, it is represented in the CSCFG with different nodes. As an example, consider the slicing criterion $(P, a)$ for the specification in Example 2, and observe in its CSCFG in Fig. 2 (right) that two different nodes are pointed out by the slicing criterion.

This means that a slicing criterion $\mathcal{C} = (P, w)$ is used to produce a slice with respect to *all* possible executions of $w$. We use function $nodes(\mathcal{C})$ to refer to all the nodes in the CSCFG pointed out by the slicing criterion $\mathcal{C}$.

Given a slicing criterion $(P, w)$, we use the CSCFG to calculate MEB. In principle, one could think that a simple backwards traversal of the graph from $nodes(\mathcal{C})$ would produce a correct slice. Nevertheless, this would produce a rather imprecise slice because this would include pieces of code which cannot be executed but they refer to the slicing criterion (e.g., dead code). The union of paths from `MAIN` to $nodes(\mathcal{C})$ is neither a solution because it would be too imprecise by including in the slice parts of code which are executed before the slicing criterion only in some executions. For instance, in the process `(b→a→STOP)□(c→a→STOP),c` belongs to one of the paths to `a`, but it must be executed before `a` or not depending on the choice. The intersection of paths is neither a solution as it can be seen in the process `a→((b→SKIP)||(c→SKIP));d` where `b` must be executed before `d`, but it does not belong to all the paths from `MAIN` to `d`.

Before we introduce an algorithm to compute MEB we need to formally define the notion of MEB slice.

**Definition 6.** *(MEB Slice) Given a specification $\mathcal{S}$ with an associated CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$, and a slicing criterion $\mathcal{C}$ for $\mathcal{S}$; a MEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$ is a subgraph of $\mathcal{G}$, $MEB(\mathcal{S}, \mathcal{C}) = (N', E'_c, E'_l, E'_s)$ with $N' \subseteq N$, $E'_c \subseteq E_c$, $E'_l \subseteq E_l$ and $E'_s \subseteq E_s$, where $N' = \{n | n \in N$ and $\forall\ X = (\texttt{MAIN} \rightarrow^* m), m \in nodes(\mathcal{C}) . n \in X\}$, $E'_c = \{(n, m) | n \mapsto m \in E_c$ and $n, m \in N'\}$, $E'_l = \{(n, m) | n \rightsquigarrow m \in E_l$ and $n, m \in N'\}$ and $E'_s = \{(n, m) | n \leftrightarrow m \in E_s$ and $n, m \in N'\}$.*

Algorithm 1 can be used to compute the MEB analysis. It basically computes for each node in $nodes(\mathcal{C})$ a set containing the part of the specification which must be executed before it. Then, it returns MEB as the intersection of all these sets. Each set is computed with an iterative process that takes a node and (i) it follows backwards all the control-flow edges. (ii) Those nodes that could not be executed before it are added to a black list (i.e., they are discarded because they belong to a non-executed choice). And (iii) synchronizations are followed in order to reach new nodes that must be executed before it.

The algorithm always terminates. We can ensure this due to the invariant $pending \cap Meb = \varnothing$ which is always true at the end of the loop (8). Then, because $Meb$ increases in every iteration (5) and the size of $N$ is finite, *pending* will eventually become empty and the loop will terminate.

**Algorithm 1.** Computing the MEB set

**Input:** A CSCFG $(N, E_c, E_l, E_s)$ of a specification $\mathcal{S}$ and a slicing criterion $\mathcal{C}$
**Output:** A CSCFG's subgraph

Function $buildMeb(n)$ :=
(1) $pending := \{n' \mid (n' \mapsto o) \in E_c\}$ where $o \in \{n\} \cup \{o' \mid o' \leftrightarrow n\}$
(2) $Meb := pending \cup \{o \mid o \in N$ and $\texttt{MAIN} \mapsto^* o \mapsto^* m,\ m \in pending\}$
(3) $blacklist := \{p \mid p \in N \backslash Meb$ and $o \mapsto^* p$, with $o \in choices(Meb)\}$
(4) $pending := \{q \mid q \in N \backslash (blacklist \cup Meb)$
and $q \leftrightarrow r \vee (q \rightsquigarrow r$ and $r \not\mapsto^* n)$ with $r \in Meb\}$
(5) while $\exists\ m \in pending$ do
(6) $\quad Meb := Meb \cup \{m\} \cup \{o \mid o \in N$ and $\texttt{MAIN} \mapsto^* o \mapsto^* m\}$
(7) $\quad sync := \{q \mid q \in N \backslash (blacklist \cup Meb)$
and $q \leftrightarrow r \vee (q \rightsquigarrow r$ and $r \not\mapsto^* n)$ with $r \in Meb\}$
(8) $\quad pending := (pending \backslash Meb) \cup sync$
(9) return $Meb$

**Return:** $MEB(\mathcal{S},\mathcal{C}) = (N' = \bigcap_{n \in nodes(\mathcal{C})} buildMeb(n),$
$E'_c = \{(n,m) | n \mapsto m \in E_c$ and $n, m \in N'\},$
$E'_l = \{(n,m) | n \rightsquigarrow m \in E_l$ and $n, m \in N'\},$
$E'_s = \{(n,m) | n \leftrightarrow m \in E_s$ and $n, m \in N'\}).$

The CEB analysis computes the set of nodes in the CSCFG that could be executed before a given node $n$. This means that all those nodes that must be executed before $n$ are included, but also those nodes that are executed before $n$ in some executions, and they are not in other executions (e.g., due to non synchronized parallelism). Formally,

**Definition 7.** *(CEB Slice) Given a specification $\mathcal{S}$ with an associated CSCFG $\mathcal{G} = (N, E_c, E_l, E_s)$, and a slicing criterion $\mathcal{C}$ for $\mathcal{S}$; a CEB slice of $\mathcal{S}$ with respect to $\mathcal{C}$ is a subgraph of $\mathcal{G}$, $CEB(\mathcal{S},\mathcal{C}) = (N', E'_c, E'_l, E'_s)$ with $N' \subseteq N$, $E'_c \subseteq E_c$, $E'_l \subseteq E_l$ and $E'_s \subseteq E_s$, where $N' = \{n | n \in N$ and $\exists\ \texttt{MAIN} \rightarrow^* n \rightarrow^* m$ with $m \in nodes(\mathcal{C})\}$, $E'_c = \{(n,m) | n \mapsto m \in E_c$ and $n, m \in N'\}$, $E'_l = \{(n,m) | n \rightsquigarrow m \in E_l$ and $n, m \in N'\}$, $E'_s = \{(n,m) | n \leftrightarrow m \in E_s$ and $n, m \in N'\}$.*

Therefore, $MEB(\mathcal{S},\mathcal{C}) \subseteq CEB(\mathcal{S},\mathcal{C})$. The graph $CEB(\mathcal{S},\mathcal{C})$ can be computed with Algorithm 2 which, roughly, traverses the CSCFG forwards following all the paths that could be executed in parallel to nodes in $MEB(\mathcal{S},\mathcal{C})$.

The algorithms presented can extract a slice from any specification formed with the syntax of Fig 1. However, note that only two operators have a special treatment in the algorithms: choices (because they introduce alternative computations) and synchronized parallelisms (because they introduce synchronizations). Other operators such as prefixing, interleaving or sequential composition can be treated similarly.

**Algorithm 2.** Computing the CEB set

**Input:** A CSCFG $(N, E_c, E_l, E_s)$ of a specification $\mathcal{S}$ and a slicing criterion $\mathcal{C}$
**Output:** A CSCFG's subgraph
**Initialization:**

$$Ceb := \{m \mid m \in N_1 \text{ and } MEB(\mathcal{S},\mathcal{C}) = (N_1, E_{c1}, E_{l1}, E_{s1})\}$$
$$loopnodes := \{n \mid n_1 \mapsto^+ n \mapsto^* n_2 \rightsquigarrow n_3 \text{ and } (n \leftrightarrow n') \notin E_s \text{ with } n' \notin Ceb,\ n_1 \in choices(Ceb) \text{ and } n_3 \in Ceb\}$$
$$Ceb := Ceb \cup loopnodes$$
$$pending := \{m \mid m \notin (Ceb \cup nodes(\mathcal{C})) \text{ and } (m' \mapsto m) \in E_c, \text{ with } m' \in Ceb \backslash choices(Ceb)\}$$

Repeat
(1) if $\exists\ m \in pending \mid (m \leftrightarrow m') \notin E_s$ or $((m \leftrightarrow m') \in E_s$ and $m' \in Ceb)$
(2) then $pending := pending \backslash \{m\}$
(3) $Ceb := Ceb \cup \{m\}$
(4) $pending := pending \cup \{m'' \mid (m \mapsto m'') \in E_c$ and $m'' \notin Ceb\}$
(5) else if $\exists\ m \in pending$ and $\forall\ (m \leftrightarrow m') \in E_s\ .\ m' \in pending$
(6) then $candidate := \{m' \mid (m \leftrightarrow m') \in E_s\} \cup \{m\}$
(7) $Ceb := Ceb \cup candidate$
(8) $pending := (pending \backslash Ceb) \cup \{n \mid n \notin Ceb$ and $o \mapsto n$, with $o \in candidate\}$

Until a fix point is reached

**Return:** $CEB(\mathcal{S},\mathcal{C}) = (N' = Ceb,$
$E'_c = \{(n,m) \mid n \mapsto m \in E_c \text{ and } n, m \in N'\},$
$E'_l = \{(n,m) \mid n \rightsquigarrow m \in E_l \text{ and } n, m \in N'\},$
$E'_s = \{(n,m) \mid n \leftrightarrow m \in E_s \text{ and } n, m \in N'\}).$

## 5 Related Work

Program slicing has been already applied to concurrent programs of different programming paradigms, see e.g. [19,18]. As a result, different graph representations have arisen to represent synchronizations. The first proposal by Cheng [3] was later improved by Krinke [8,9] and Nanda [13]. All these approaches are based on the so called *threaded control flow graph* and the *threaded program dependence graph*. Unfortunately, their approaches are not appropriate for slicing CSP, because their work is based on a different kind of synchronization. They use the following concept of *interference* to represent program synchronizations.

**Definition 8.** *A node $S1$ is interference dependent on a node $S2$ if $S2$ defines a variable $v$, $S1$ uses the variable $v$ and $S1$ and $S2$ execute in parallel.*

In CSP, in contrast, a synchronization happens between two processes if the synchronized event is executed at the same time by both processes. In addition, both processes cannot proceed in their executions until they have synchronized. This is the key point that underpin our MEB and CEB analyses.

In addition, the purpose of our slicing technique is essentially different from previous work: while other approaches try to answer the question "*what parts of the program can influence the value of this variable at this point?*", our technique tries to answer the question "*what parts of the program must be executed before this point? and what parts of the program can be executed before this point?*". Therefore, our slicing criterion is different, but also the data structure we use for slicing is different. In contrast to previous work, we do not use a PDG like graph, and use instead a CFG like graph, because we focus on control flow rather than control and data dependence.

## 6 Implementation

We have implemented the MEB and CEB analyses and the algorithm to build the CSCFG for ProB. ProB [11] is an animator for the B-Method which also supports other languages such as CSP [1,12]. ProB has been implemented in Prolog and it is publicly available at `http://www.stups.uni-duesseldorf.de/ProB`.

The implementation of our slicer has three main modules. The first module implements the construction of the CSCFG. Nodes and control and loop edges are built following the algorithm of Definition 4. For synchronization edges we use an algorithm based on the approach by Naumovich et al. [14]. For efficiency reasons, the implementation of the CSCFG does some simplifications which reduces the size of the graph. For instance, "*start*" and "*end*" nodes are not present in the graph. Another simplification to reduce the size of the graph is the graph compactation which joins together all the nodes defining a sequential path (i.e., without nodes with more than one output edge). For instance, the graph of Fig. 4 is the compacted version of the CSCFG in Fig. 2. Here, e.g., node `6` accounts for the sequence of nodes `P` $\rightarrow$ `start P`. The compacted version is a very convenient representation because the reduced data structure speeds up the graph traversal process.

The second module performs the MEB and CEB analyses by implementing Algorithm 1 and Algorithm 2. Finally, the third module allows the tool to communicate with the ProB interface in order to get the slicing criterion and show the highlighted code.

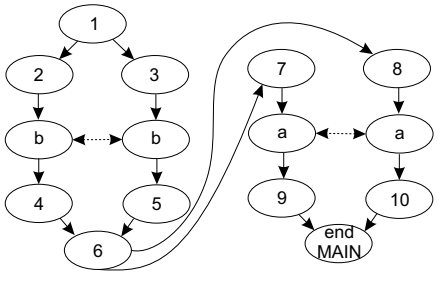



**Fig. 4.** Compacted version of the CSCFG in Fig. 2

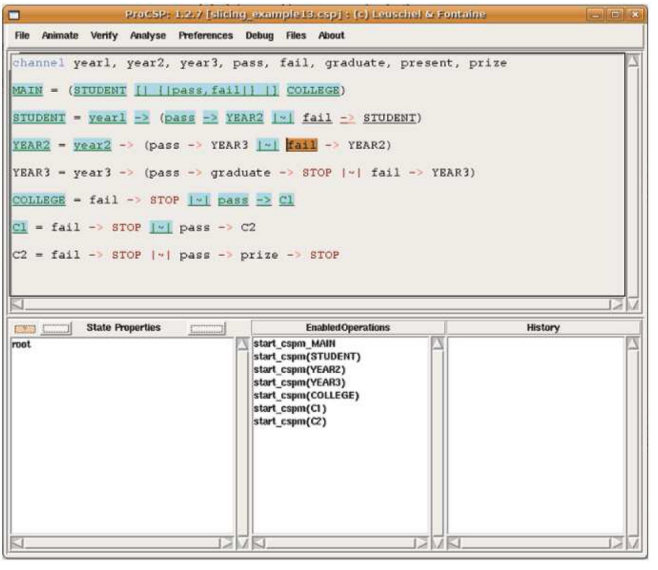



**Fig. 5.** Slice of a CSP specification in ProB

We have integrated our tool into the graphical user interface of ProB. This allows us to use features such as text coloring in order to highlight the final slices. In particular, once the user has loaded a CSP specification, she can select (with the mouse) the event or process call she is interested in. Obviously, this simple action is enough to define a slicing criterion because the tool can automatically determine the process and the specification position of interest. Then, the tool internally generates the CSCFG of the specification and uses the MEB and CEB algorithms to construct the slices. The result is shown to the user by highlighting the part of the specification that must be executed before the specified event, and underscoring the part of the specification that could be executed before this event. Figure 5 shows a screenshot of the tool showing a slice of a CSP specification.

## 7 Conclusions

This work defines two new static analysis that can be applied to languages with explicit synchronizations such as CSP. In particular, we introduce a method to slice CSP specifications, in such a way that, given a CSP specification and a slicing criterion we produce a slice such that (i) it is a subset of the specification (i.e., it is produced by deleting some parts of the original specification); (ii) it contains all the parts of the specification which must be executed (in any execution) before the slicing criterion (MEB analysis); and (iii) we can also produce an augmented slice which also contains those parts of the specification that could be executed before the slicing criterion (CEB analysis).

We have presented two algorithms to compute the MEB and CEB analyses which are based on a new data structure, the CSCFG, that has shown to be more precise than the previously used SCFG. The advantage of the CSCFG is that it cares about contexts, and it is able to distinguish between different contexts in which a process is called.

We have built a prototype which implements all the data structures and algorithms defined in the paper; and we have integrated it into the system ProB. Preliminary experiments has demonstrated the usefulness of the technique.

## References


1. Butler, M., Leuschel, M.: Combining CSP and B for specification and property verification. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)
2. Callahan, D., Sublok, J.: Static analysis of low-level synchronization. In: Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and Distributed Debugging (PADD 1988), New York, NY, USA, pp. 100–111 (1988)
3. Cheng, J.: Slicing concurrent programs - a graph-theoretical approach. In: Fritzson, P.A. (ed.) AADEBUG 1993. LNCS, vol. 749, pp. 223–240. Springer, Heidelberg (1993)
4. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems 9(3), 319–349 (1987)
5. Harrold, M.J., Rothermel, G., Sinha, S.: Computation of interprocedural control dependence. In: International Symposium on Software Testing and Analysis, pp. 11–20 (1998)
6. Hoare, C.A.R.: Communicating sequential processes. Communications ACM 26(1), 100–106 (1983)
7. Kavi, K.M., Sheldon, F.T., Shirazi, B.: Reliability analysis of CSP specifications using petri nets and markov processes. In: Proceedings 28th Annual Hawaii International Conference on System Sciences. Software Technology, January 3-6, Wailea, HI, vol. 2, pp. 516–524 (1995)
8. Krinke, J.: Static slicing of threaded programs. In: Workshop on Program Analysis For Software Tools and Engineering, pp. 35–42 (1998)
9. Krinke, J.: Context-sensitive slicing of concurrent programs. ACM SIGSOFT Software Engineering Notes 28(5) (2003)
10. Ladkin, P., Simons, B.: Static deadlock analysis for csp-type communications. In: Responsive Computer Systems, ch. 5. Kluwer Academic Publishers, Dordrecht (1995)
11. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Journal of Software Tools for Technology Transfer 10(2), 185–203 (2008)
12. Leuschel, M., Fontaine, M.: Probing the depths of CSP-M: A new FDR-compliant validation tool. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 278–297. Springer, Heidelberg (2008)
13. Nanda, M.G., Ramesh, S.: Slicing concurrent programs. In: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis, New York, NY, USA, pp. 180–190 (2000)

14. Naumovich, G., Avrunin, G.S.: A conservative data flow algorithm for detecting all pairs of statements that happen in parallel. SIGSOFT Software Engineering Notes 23(6), 24–34 (1998)
15. Roscoe, A.W., Gardiner, P.H.B., Goldsmith, M., Hulance, J.R., Jackson, D.M., Scattergood, J.B.: Hierarchical compression for model-checking CSP or how to check $10^{20}$ dining philosophers for deadlock. In: Proceedings of the First International Workshop Tools and Algorithms for Construction and Analysis of Systems, pp. 133–152 (1995)
16. Tip, F.: A survey of program slicing techniques. Journal of Programming Languages 3, 121–189 (1995)
17. Weiser, M.D.: Program slicing. IEEE Transactions on Software Engineering 10(4), 352–357 (1984)
18. Zhao, J., Cheng, J., Ushijima, K.: Slicing concurrent logic programs. In: Proceedings of the 2nd Fuji International Workshop on Functional and Logic Programming, pp. 143–162 (1997)
19. Zhao, J.: Slicing aspect-oriented software. In: Proceedings of the 10th IEEE International Workshop on Programming Comprehension, pp. 251–260 (2002)

# Fast Offline Partial Evaluation of Large Logic Programs*

Michael Leuschel[1] and Germán Vidal[2]

[1] Institut für Informatik, Universität Düsseldorf, D-40225, Düsseldorf, Germany
leuschel@cs.uni-duesseldorf.de
[2] DSIC, Technical University of Valencia, E-46022, Valencia, Spain
gvidal@dsic.upv.es

**Abstract.** In this paper, we present a fast binding-time analysis (BTA) by integrating a size-change analysis, which is independent of a selection rule, into a classical BTA for offline partial evaluation of logic programs. In contrast to previous approaches, the new BTA is conceptually simpler and considerably faster, scaling to medium-sized or even large examples and, moreover, it ensures both the so called local and global termination. We also show that through the use of selective hints, we can achieve both good specialisation results and a fast BTA and specialisation process.

## 1 Introduction

There are two main approaches to *partial evaluation* [8], a well-known technique for program specialisation. *Online* partial evaluators basically include an augmented interpreter that tries to evaluate the program constructs as much as possible—using the partially known input data—while still ensuring the termination of the process. *Offline* partial evaluators, on the other hand, require a *binding-time analysis* (BTA) to be run before specialisation, which annotates the source code to be specialised. Roughly speaking, the BTA annotates the various calls in the source program as either unfold (executed by the partial evaluator) or memo (executed at run time, i.e., memoized), and annotates the arguments of the calls themselves as static (known at specialisation time) or dynamic (only definitely known at run time).

In the context of logic programming, a BTA should ensure that the annotations of the arguments are correct, in the sense that an argument marked as static will be ground in all possible specialisations. It should also ensure that the specialiser will always terminate. This can be broadly classified into local and global termination [9]. The *local* termination of the process implies that no atom is infinitely unfolded. The *global* termination ensures that only a finite number of atoms are unfolded. In previous work, Craig et al [4] have presented a fully automatic BTA for logic programs, whose output can be used for the

* This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and DAAD PPP D/06/12830 together with *Acción Integrada* HA2006-0008.

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 119–134, 2009.

offline partial evaluator LOGEN [11]. Unfortunately, this BTA still suffers from some serious practical limitations:

- The current implementation does not guarantee global termination.
- The technique and its implementation are quite complicated, consisting of a combination of various other analyses: model-based binding-time inference, binary clause generation, inter-argument size relation analysis with polyhedra. To make matters more complicated, these analyses also run on different Prolog systems. As a consequence, the current implementation is quite fragile and hard to maintain.
- In addition to the implementation complexity, the technique is also very slow and does not scale to medium-sized examples.

Recently, Vidal [18] has introduced a quasi-termination analysis for logic programs that is independent of the selection rule. This approach is less precise than other termination analyses that take into account a particular selection strategy but, as a counterpart, is also faster and well suited for partial evaluation (where flexible selection strategies are often mandatory, see, e.g., [1,9]).

In this paper, we introduce a new BTA for logic programs with the following advantages:

- it is conceptually simpler and considerably faster, scaling to medium-sized or even large examples;
- the technique does ensure both local and global termination;
- the technique can make use of user-provided hints to obtain better specialisation.

The main source of improvement comes from using a size-change analysis for termination purposes rather than a termination analysis based on the *abstract binary unfoldings* [3] as in [4]. Basically, the main difference between both termination analyses is that the binary unfoldings consider a particular selection strategy (i.e., Prolog's leftmost selection strategy). As a consequence, every time the annotation of an atom changes from unfold to memo during the BTA of [4], the termination analysis should be redone from scratch in order to take into account that this atom would not be unfolded (thus avoiding the propagation of some bindings). On the other hand, the size-change analysis is independent of a particular selection strategy. As a consequence, it is less precise, since no variable bindings are propagated between the body atoms of a clause, but it should be run only *once*. In general the termination analysis is the most expensive component of a BTA.

We have implemented the new approach, and we will show on experimental results that the new technique is indeed much faster and much more scalable. On some examples, the accuracy is still sub-optimal, and we present various ways to improve this. Still, this is the first BTA for logic programs that can be applied to larger programs, and as such is an important step forward.

The paper is organized as follows. Section 2 introduces a fully automatic BTA which is parameterized by the output of a termination analysis, which is then presented in Sect. 3. Section 4 presents a summary of experimental results

which demonstrate the usefulness of our approach. Finally, Sect. 5 concludes and presents some directions for future work.

## 2 A Fully Automatic Binding-Time Analysis

In the remainder we assume basic knowledge of the fundamentals of logic programming [2,17]. In this section, we present a fully automatic BTA for (definite) logic programs. Our algorithm is parametric w.r.t.

- a domain of binding-times and
- a function for propagating binding-times through the atoms of clause bodies.

For instance, one can consider a simple domain with the basic binding-times static (definitely known at partial evaluation time) and dynamic (possibly unknown at partial evaluation time). More refined binding-time domains could also include, e.g., the following elements:

- nonvar: the argument is not a variable at partial evaluation time, i.e., the top-level function symbol is known;
- list_nonvar: the argument is definitely bound to a finite list, whose elements are not variables;
- list: the argument is definitely bound to a finite list of possibly unknown arguments at partial evaluation time.

In the LOGEN system [11], an *offline* partial evaluator for Prolog, the user can also define their own binding-times [4] (called *binding-types* in this context), and one can use the pre-defined list-constructor to define additional types such as list(dynamic) to denote a list of known length with dynamic elements, or list(nonvar) to denote a list of known length with non-variable elements.

For any binding-time domain $(\mathcal{D}, \sqsubseteq)$, we let $b_1 \sqsubseteq b_2$ denote that $b_1$ is *less dynamic* than $b_2$; also, we denote the least upper bound of binding-times $b_1$ and $b_2$ by $b_1 \sqcup b_2$ and the greatest lower bound by $b_1 \sqcap b_2$. For instance, if $\mathcal{D} = \{\mathsf{static}, \mathsf{dynamic}\}$ and $\mathsf{static} \sqsubseteq \mathsf{dynamic}$, we have

$$\mathsf{static} \sqcup \mathsf{static} = \mathsf{static}$$
$$\mathsf{static} \sqcup \mathsf{dynamic} = \mathsf{dynamic} \sqcup \mathsf{static} = \mathsf{dynamic} \sqcup \mathsf{dynamic} = \mathsf{dynamic}$$
$$\mathsf{static} \sqcap \mathsf{static} = \mathsf{static} \sqcap \mathsf{dynamic} = \mathsf{dynamic} \sqcap \mathsf{static} = \mathsf{static}$$
$$\mathsf{dynamic} \sqcap \mathsf{dynamic} = \mathsf{dynamic}$$

Given a set of binding-times $W$, we define $\sqcup W$ as follows (in a similar way we define $\sqcap W$):

$$\sqcup W \quad = \quad \begin{cases} \mathsf{static} & \text{if } W = \emptyset \\ b & \text{if } W = \{b\} \\ b_1 \sqcup (b_2 \sqcup (\ldots \sqcup b_{n-1}) \sqcup b_n) & \text{if } W = \{b_1, \ldots, b_n\},\ n > 0 \end{cases}$$

In the following, a *pattern* is defined as an expression of the form $p(b_1, \ldots, b_n)$ where $p/n$ is a predicate symbol and $b_1, \ldots, b_n$ are binding-times. Given a binding-time domain, we consider an associated domain of *abstract substitutions* that map variables to binding-times. We introduce the following auxiliary functions that deal with patterns and abstract substitutions:

- Given a pattern $p(b_1, \ldots, b_n)$, the function $\bot(p(b_1, \ldots, b_n))$ returns a new pattern $p(\mathsf{static}, \ldots, \mathsf{static})$ where all arguments are static.
- Given an atom $A$ and a pattern $pat$ with $pred(A) = pred(pat)$,[1] the partial function $asub(A, pat)$ returns an abstract substitution for the variables of $A$ according to pattern $pat$. For instance, for a simple binding-time domain $\mathcal{D} = \{\mathsf{static}, \mathsf{dynamic}\}$, function $asub$ is defined as follows:[2]

$$asub(p(t_1, \ldots, t_n), p(b_1, \ldots, b_n)) = \{x/b \mid x \in \mathcal{V}ar(p(t_1, \ldots, t_n)) \wedge b = \sqcap\{b_i \mid x \in \mathcal{V}ar(t_i)\}\}$$

  Roughly speaking, a variable that appears only in one argument $t_i$ will be mapped to $b_i$; if the same variable appears in several arguments, then it is mapped to the greatest lower bound of the corresponding binding-times of these arguments. For instance, we have

$$asub(p(X, X), p(\mathsf{static}, \mathsf{dynamic})) = \{X/\mathsf{static}\}$$

  Observe that the greatest lower bound is used to compute the less dynamic binding-time of a variable when it is bound to different values.
- Given an atom $A = p(t_1, \ldots, t_n)$ and an abstract substitution $\sigma$, function $pat(A, \sigma)$ returns a pattern in which the binding-time of every argument is determined by the abstract substitution. For the binding-time domain $\mathcal{D} = \{\mathsf{static}, \mathsf{dynamic}\}$, this function can be formally defined as follows:

$$pat(A, \sigma) = p(b_1, \ldots, b_n) \text{ where } b_i = \sqcup\{x\sigma \mid x \in \mathcal{V}ar(t_i)\},\ i = 1, \ldots, n$$

  E.g., we have $pat(p(X, X), \{X/\mathsf{static}\}) = p(\mathsf{static}, \mathsf{static})$ and $pat(q(f(X, Y)), \{X/\mathsf{static}, Y/\mathsf{dynamic}\}) = q(\mathsf{dynamic})$.

Now, we present our BTA algorithm in a stepwise manner.

### 2.1 Call and Success Pattern Analysis

The first step towards a BTA is basically a simple call and success pattern analysis parameterized by the considered binding-time domain $(\mathcal{D}, \sqsubseteq)$. The algorithm is shown in Fig. 1. Here, we keep a *memo* table $\mu$ with the call and success patterns already found in the analysis, i.e., if $\mu(cpat) = spat$ then we have a call pattern $cpat$ with associated success pattern $spat$; initially, all success patterns have static arguments. In order to add new entries to the memo table, we use the following function $addentry$:

$$addentry(pattern) = \textbf{if } pattern \notin dom(\mu) \textbf{ then } \mu(pattern) := \bot(pattern) \textbf{ fi}$$

where the notation $\mu(pattern) := \bot(pattern)$ is used to denote an update of $\mu$.

Basically, the algorithm takes a logic program $P$ and an entry pattern $epat$ and, after initializing the memo table, enters a loop until the memo table reaches a fixed point. Every iteration of the main loop proceeds as follows:

[1] Auxiliary function *pred* returns the predicate name and arity of an atom or pattern.
[2] $\mathcal{V}ar(s)$ denotes the set of variables in the term or atom $s$.

1. **Input:** a program $P$ and an entry pattern *epat*
2. **Initialisation:** $\mu := \emptyset$; $addentry(epat)$
3. **repeat**
   **for** all $cpat \in dom(\mu)$:
   for all clauses $H \leftarrow B_1, \ldots, B_n \in P$, $n \geqslant 0$, with $pred(H) = pred(cpat)$:
   (a) $\sigma_0 := asub(H, cpat)$
   (b) **for** i = 1 to n:
   $\sigma_i := get(B_i, \sigma_{i-1})$
   (c) $\mu(cpat) := \mu(cpat) \sqcup pat(H, \sigma_n)$
   **until** $\mu$ doesn't change

**Fig. 1.** Call and success pattern analysis

- for every call pattern *cpat* with a matching clause $H \leftarrow B_1, \ldots, B_n$, we first compute the entry abstract substitution $asub(H, cpat)$;
- then, we use function *get* to propagate binding-times through the atoms of the body, thus updating correspondingly the memo table with the call and (initial) success patterns for every atom;
- finally, we update in the memo table the success pattern associated to the call pattern *cpat* using the exit abstract substitution of the clause.

Function *get* is used to propagate binding-times through the atoms of the bodies as follows:

$$get(B, \sigma) \;=\; addentry(pat(B, \sigma));\ \textbf{return}\ asub(B, \mu(pat(B, \sigma)))$$

In the next section, we present a BTA that slightly extends this algorithm.

### 2.2 A BTA Ensuring Local and Global Termination

In contrast to the call and success pattern analysis of Fig. 1, a BTA should annotate every call with either unfold or memo so that

- all atoms marked as unfold can be unfolded as much as possible (as indicated by the annotations) while still guaranteeing the local termination, and
- global termination is guaranteed by generalising the dynamic arguments whenever a new atom is added to the set of (to be) partially evaluated atoms; also, all arguments marked as static must indeed be ground.

Figure 2 shows a BTA that slightly extends the call and success pattern analysis of Fig. 1 (differences appear in a box). The main changes are as follows:

- We consider that each call $B$ is uniquely identified by a program point $ppoint(B)$. The algorithm keeps track of the considered program points using the set *memo* (initially empty).
- The function for propagating binding-times, now called $get'$, takes an additional parameter: the program point of the considered atom; function $get'$ is defined as follows:

1. **Input:** a program $P$ and an entry pattern *epat*
2. **Initialisation:** $\mu := \emptyset$; $addentry(epat)$; $\boxed{memo := \emptyset}$
3. **repeat**
   **for** all $cpat \in dom(\mu)$:
   for all clauses $H \leftarrow B_1, \ldots, B_n \in P$, $n \geqslant 0$, with $pred(H) = pred(cpat)$:
   (a) $\sigma_0 := asub(H, cpat)$
   (b) **for** i = 1 to n:
   $\sigma_i := \boxed{get'(B_i, \sigma_{i-1}, ppoint(B_i))}$
   (c) $\mu(cpat) := \mu(cpat) \sqcup pat(H, \sigma_n)$
   **until** $\mu$ doesn't change

**Fig. 2.** A BTA ensuring local and global termination

$$
\begin{aligned}
get'(B, \sigma, pp) = \; & \textbf{if } unfold(pat(B,\sigma)) \wedge pp \notin memo \\
& \quad \textbf{then return } get(B, \sigma) \\
& \quad \textbf{else } addentry(gen(pat(B,\sigma))); memo := memo \cup \{pp\} \\
& \qquad \textbf{return } \sigma \textbf{ fi} \\
get(B, \sigma) \quad = \; & addentry(pat(B,\sigma)); \textbf{ return } asub(B, \mu(pat(B,\sigma)))
\end{aligned}
$$

(note that function *get* is the same as the one used in Fig. 1).

- Auxiliary functions *gen* and *unfold* are defined as follows:
  - Given a pattern *pat*, function $unfold(pat)$ returns true if *pat* is safe for unfolding, i.e., if it guarantees *local* termination.
  - Given a pattern *pat*, function $gen(pat)$ returns a generalization of *pat* (i.e., $pat \sqsubseteq gen(pat)$) that ensures *global* termination.

  Precise definitions for *gen* and *unfold* will be presented in the next section.

## 3 Size-Change Termination Analysis

In this section, we first present the basis of the size-change analysis of [18], which is used to check the (quasi-)termination of a program. Then, we introduce appropriate definitions for functions *unfold* (local termination) and *gen* (global termination) which are based on the output of the size-change analysis.

We denote by $calls_P^{\mathcal{R}}(Q_0)$ the set of calls in the computations of a goal $Q_0$ within a logic program $P$ and a computation rule $\mathcal{R}$. We say that a query $Q$ is *strongly terminating* w.r.t. a program $P$ if every SLD derivation for $Q$ with $P$ is finite. The query $Q$ is *strongly quasi-terminating* if, for every computation rule $\mathcal{R}$, the set $call_P^{\mathcal{R}}(Q)$ contains finitely many nonvariant atoms. A program $P$ is strongly (quasi-)terminating w.r.t. a set of queries $\mathcal{Q}$ if every $Q \in \mathcal{Q}$ is strongly (quasi-)terminating w.r.t. $P$. For conciseness, in the remainder of this paper, we write "(quasi-)termination" to refer to "strong (quasi-)termination."

Size-change analysis is based on constructing graphs that represent the decrease of the arguments of a predicate from one call to another. For this purpose, some ordering on terms is required. In [18], reduction orders $(\succsim, \succ)$ *induced* from symbolic norms $||\cdot||$ are used:

**Definition 1 (symbolic norm [16]).** *Given a term $t$,*

$$||t|| = \begin{cases} m + \sum_{i=1}^{n} k_i||t_i|| & \text{if } t = f(t_1, \ldots, t_n),\ n \geqslant 0 \\ t & \text{if } t \text{ is a variable} \end{cases}$$

*where $m$ and $k_1, \ldots, k_n$ are non-negative integer constants depending only on $f/n$. Note that we associate a variable over integers with each logical variable (we use the same name for both since the meaning is clear from the context).*

The introduction of variables in the range of the norm provides a simple mechanism to express dependencies between the sizes of terms.

The associated induced orders $(\succsim, \succ)$ are defined as follows: $t_1 \succ t_2$ (respec. $t_1 \succsim t_2$) if $||t_1\sigma|| > ||t_2\sigma||$ (respec. $||t_1\sigma|| \geqslant ||t_2\sigma||$) for all substitutions $\sigma$ that make $||t_1\sigma||$ and $||t_2\sigma||$ ground (e.g., an integer constant). Two popular instances of symbolic norms are the symbolic *term-size* norm $||\cdot||_{ts}$ (which counts the arities of the term symbols) and the symbolic *list-length norm* $||\cdot||_{ll}$ (which counts the number of elements of a list), e.g.,

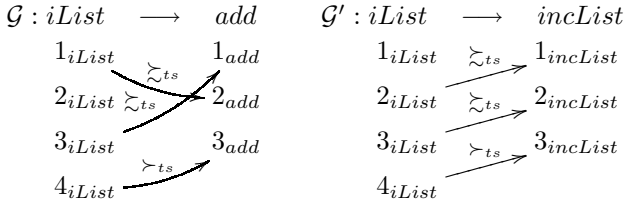
$$f(X,Y) \succ_{ts} f(X,a) \text{ since } ||f(X,Y)||_{ts} = X + Y + 2 > X + 2 = ||f(X,a)||_{ts}$$
$$[X|R] \succsim_{ll} [s(X)|R] \text{ since } |||[X|R]|||_{ll} = R + 1 \geqslant R + 1 = |||[s(x)|R]|||_{ll}$$

Now, we produce a *size-change graph* $\mathcal{G}$ for every pair $(H, B_i)$ of every clause $H \leftarrow B_1, \ldots, B_n$ of the program, with edges between the arguments of $H$—the *output nodes*—and the arguments of $B_i$—the *input nodes*—when the size of the corresponding terms decrease w.r.t. a given reduction pair $(\succsim, \succ)$. A size-change graph is thus a bipartite labelled graph $\mathcal{G} = (V, W, E)$ where $V$ and $W$ are the labels of the output and input nodes, respectively, and $E \subseteq V \times W \times \{\succsim, \succ\}$ are the edges.

*Example 1.* Consider the following simple program:

$(c_1)$ $incList([\,], \_, [\,]).$
$(c_2)$ $incList([X|R], I, L) \leftarrow iList(X, R, I, L).$
$(c_3)$ $iList(X, R, I, [XI|RI]) \leftarrow add(I, X, XI), incList(R, I, RI).$
$(c_4)$ $add(0, Y, Y).$
$(c_5)$ $add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z).$

Here, the size-change graphs associated to, e.g., clause $c_3$ are as follows:




using a reduction pair $(\succsim_{ts}, \succ_{ts})$ induced from the symbolic term-size norm.

In order to identify the program *loops*, we should compute roughly a transitive closure of the size-change graphs by composing them in all possible ways. Basically, given two size-change graphs:

$$\mathcal{G} = (\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1) \qquad \mathcal{H} = (\{1_q, \ldots, m_q\}, \{1_r, \ldots, l_r\}, E_2)$$

w.r.t. the same reduction pair $(\succsim, \succ)$, their concatenation is defined by

$$\mathcal{G} \bullet \mathcal{H} = (\{1_p, \ldots, n_p\}, \{1_r, \ldots, l_r\}, E)$$

where $E$ contains an edge from $i_p$ to $k_r$ iff $E_1$ contains an edge from $i_p$ to some $j_q$ and $E_2$ contains an edge from $j_q$ to $k_r$. Furthermore, if some of the edges are labelled with $\succ$, then so is the edge in $E$; otherwise, it is labelled with $\succsim$.

In particular, we only need to consider the *idempotent* size-change graphs $\mathcal{G}$ with $\mathcal{G} \bullet \mathcal{G} = \mathcal{G}$, because they represent the (potential) program loops.

*Example 2.* For the program of Example 1, we compute the following idempotent size-change graphs:

$\mathcal{G}_1 : incList \longrightarrow incList$

$1_{incList} \xrightarrow{\succ_{ts}} 1_{incList}$
$2_{incList} \xrightarrow{\succsim_{ts}} 2_{incList}$
$3_{incList} \xrightarrow{\succ_{ts}} 3_{incList}$

$\mathcal{G}_2 : iList \longrightarrow iList$

$1_{iList} \xrightarrow{\succ_{ts}} 1_{iList}$
$2_{iList} \xrightarrow{\succ_{ts}} 2_{iList}$
$3_{iList} \xrightarrow{\succsim_{ts}} 3_{iList}$
$4_{iList} \xrightarrow{\succ_{ts}} 4_{iList}$

$\mathcal{G}_3 : add \longrightarrow add$

$1_{add} \xrightarrow{\succ_{ts}} 1_{add}$
$2_{add} \xrightarrow{\succsim_{ts}} 2_{add}$
$3_{add} \xrightarrow{\succ_{ts}} 3_{add}$

that represent how the size of the arguments of the three potentially looping predicates changes from one call to another.

### 3.1 Ensuring Local Termination

In this section, we consider the local termination of the specialisation process, i.e., we analyse whether the unfolding of an atom terminates for any selection strategy. Then, we present an appropriate definition for the function *unfold* of the BTA shown in Fig. 2.

Let us first recall the notion of *instantiated enough* w.r.t. a symbolic norm from [16]: a term $t$ is instantiated enough w.r.t. a symbolic norm $||\cdot||$ if $||t||$ is an integer constant. We now present a sufficient condition for termination. Basically, we require the decreasing parameters of (potentially) looping predicates to be instantiated enough w.r.t. a symbolic norm in the considered computations.

**Theorem 1 (termination [18]).** *Let $P$ be a program and let $(\succsim, \succ)$ be a reduction pair induced from a symbolic norm $||\cdot||$. Let $\mathcal{A}$ be a set of atoms. If every idempotent size-change graph for $P$ contains at least one edge $i_p \xrightarrow{\succ} i_p$ such that, for every atom $A \in \mathcal{A}$, computation rule $\mathcal{R}$, and atom $p(t_1, \ldots, t_n) \in calls_P^{\mathcal{R}}(A)$, $t_i$ is instantiated enough w.r.t. $||\cdot||$, then $P$ is terminating w.r.t. $\mathcal{A}$.*

For instance, according to the idempotent graphs of Example 2, computations terminate whenever either the first or the last argument of *incList* is instantiated enough w.r.t. $||\cdot||_{ts}$, either the second or the fourth argument of *iList* is instantiated enough w.r.t. $||\cdot||_{ts}$, and either the first or the third argument of *add* is instantiated enough w.r.t. $||\cdot||_{ts}$.

Note that the strictly decreasing arguments should be instantiated enough in every possible derivation w.r.t. any computation rule. Although this condition is undecidable in general, it can be approximated by using the binding-times of the computed call patterns. In the following, we extend the notion of "instantiated enough" to binding-times as follows: a binding-time $b$ is instantiated enough w.r.t. a symbolic norm $||\cdot||$ if, for all term $t$ approximated by the binding-time $b$, $t$ is instantiated enough w.r.t. $||\cdot||$.

Now, we introduce an appropriate definition of *unfold*:

**Definition 2 (local termination).** *Let $P$ be a program and let $(\succsim, \succ)$ be a reduction pair induced from a symbolic norm $||\cdot||$. Let $\mathcal{G}$ be the idempotent size-change graphs from the size-change analysis and $pat = p(b_1, \ldots, b_n)$ be a pattern. Function unfold($pat$) returns true if every size-change graph for $p/n$ in $\mathcal{G}$ contains at least one edge $i_p \xrightarrow{\succ} i_p$, $1 \leqslant i \leqslant n$, such that $b_i$ is instantiated enough w.r.t. $||\cdot||$.*

For instance, given the idempotent size-change graphs of Example 2, we have

$$unfold(incList(\mathsf{static}, \mathsf{dynamic}, \mathsf{dynamic})) = true$$

since there is an edge $1_{incList} \xrightarrow{\succ} 1_{incList}$ and the binding-time static is clearly instantiated enough w.r.t. any norm. In contrast, we have

$$unfold(incList(\mathsf{dynamic}, \mathsf{static}, \mathsf{dynamic})) = false$$

since only the edges $1_{incList} \xrightarrow{\succ} 1_{incList}$ and $3_{incList} \xrightarrow{\succ} 3_{incList}$ are labeled with $\succ$ and the binding-time dynamic is not instantiated enough w.r.t. any norm.

### 3.2 Ensuring Global Termination

In order to ensure the global termination of the specialisation process, we should ensure that only a finite number of non-variant atoms are added to the set of (to be) partially evaluated atoms, i.e., that the sequence of atoms is *quasi-terminating*.

In principle, following the results in [18], function *gen* could be defined as follows: Given a pattern $pat = p(b_1, \ldots, b_n)$, function $gen(pat)$ returns $p(b'_1, \ldots, b'_n)$ where $b'_i = b_i$ if every idempotent size-change graph for $p/n$ computed by the size-change analysis contains an edge $i_p \xrightarrow{R} i_p$, $R \in \{\succ, \succsim\}$, and dynamic otherwise. Furthermore, the considered reduction pair should be induced from a *bounded* symbolic norm $||\cdot||$, i.e., a norm such that the set $\{s \mid ||t|| \geqslant ||s||\}$ contains a finite number of nonvariant terms for any term $t$.

A main limitation of this approach comes from requiring $i_p \xrightarrow{R} i_p$, $R \in \{\succ, \succsim\}$, which is too restrictive for some examples. Consider, e.g., the size-change graphs of Example 2. Here, we would have $gen(iList(b_1, b_2, b_3, b_4)) = iList(\mathsf{dynamic}, b_2, b_3, b_4)$ for any binding-times $b_1, b_2, b_3, b_4$ since there is no edge $1_{iList} \xrightarrow{R} 1_{iList}$, $R \in \{\succ, \succsim\}$.

However, any sequence of calls to predicate *iList* is clearly quasi-terminating because the size of all four predicate arguments is bounded by the size of *some*—not necessarily the same—argument of the previous call. Therefore, in this paper, we consider the following improved definition of *gen*:

**Definition 3 (global termination).** *Let $P$ be a program and let $(\succsim, \succ)$ be a reduction pair induced from a bounded symbolic norm $||\cdot||$. Let $\mathcal{G}$ be the idempotent size-change graphs computed by the size-change analysis. Given a pattern $pat = p(b_1, \ldots, b_n)$, function $gen(pat)$ returns $p(b'_1, \ldots, b'_n)$ where $b'_i = b_i$ if every size-change graph for $p/n$ in $\mathcal{G}$ contains an edge $j_p \xrightarrow{R} i_p$, $R \in \{\succ, \succsim\}$, for some $j \in \{1, \ldots, n\}$, and* dynamic *otherwise.*

Now, given the idempotent size-change graphs of Example 2, we have $gen(pat) = pat$ for all pattern *pat* since there is an entry edge to every predicate argument, i.e., no generalisation is required at the global level.

Another limitation of [18] comes from the use of bounded norms, which excludes for instance the use of the well-known list-length norm. This approach, however, might be too coarse to produce useful results in some cases. In fact, by taking into account that some generalisation can be done during the specialisation process (i.e., at the global level), a weaker, more useful condition can be formulated.

Consider, e.g., the program $\{p([X]) \leftarrow p([s(X)]).\}$. This program cannot be proved quasi-terminating according to [18] because the symbolic list-length norm cannot be used; actually, the program is not quasi-terminating:

$$p([a]) \rightsquigarrow p([s(a)]) \rightsquigarrow p([s(s(a))]) \rightsquigarrow \ldots$$

However, it can be acceptable for partial evaluation as long as those symbols that are not taken into account by this norm are generalised in the global level, i.e., as long as the list arguments are replaced with fresh variables at the global level.

Further details on this extension can be found in [12].

## 4 The BTA in Practice

Our new binding-time analysis is still being continuously extended and improved. The implementation was done using SICStus Prolog and provides a command-line interface. The BTA is by default polyvariant (but can be forced to be monovariant) and uses a domain with the following values: static, list_nv (for lists of non-variable terms), list, nv (for non-variable terms), and dynamic. The user

| benchmark | original runtime | BTA analysis | LOGEN spec. time | specialised runtime | ECCE spec. time | specialised runtime |
|---|---|---|---|---|---|---|
| contains.kmp | 0.18 | 0.01 | 0.002 | 0.18 | 0.11 | 0.02 |
| imperative-power | 0.33 | 2.35 | 0.009 | 0.38 | 0.82 | 0.15 |
| liftsolve.app | 0.46 | 0.02 | 0.014 | 0.28 | 0.10 | 0.02 |
| match-kmp | 1.82 | 0.00 | 0.002 | 1.52 | 0.02 | 0.91 |
| regexp.r3 | 1.71 | 0.01 | 0.005 | 0.86 | 0.05 | 0.81 |
| ssuply | 0.23 | 0.01 | 0.001 | 0.00 | 0.02 | 0.00 |
| vanilla | 0.19 | 0.01 | 0.004 | 0.14 | 0.04 | 0.03 |
| lambdaint | 0.60 | 0.17 | 0.005 | 0.61 | 0.26 | *0.02 |
| picemul | - | 0.31 | 4.805 | - | 576.370 | - |

**Fig. 3.** Empirical Results

can also provide hints to the BTA (see below). The implemented size-change analysis uses a reduction pair induced from the symbolic term-size norm.

We provide some preliminary experimental results below. The experiments were run on a MacBook Pro with a 2.33 GHz Core2 Duo Processor and 3 GB of RAM. Our BTA was run using SICStus Prolog 4.04, LOGEN and its generated specialisers were run using Ciao Prolog 1.13. We compared the results against the online specialiser ECCE [15], which was compiled using SICStus Prolog 3.12.8.

### 4.1 Results in Fully Automatic Mode

Figure 3 contains an overview of our empirical results, where all times are in seconds. A value of 0 means that the timing was below our measuring threshold. The first six benchmarks come from the DPPD [13] library, vanilla and lambdaint come from [10] and picemul from [7].

**DPPD.** Let us first examine some the results from Fig. 3 for the DPPD benchmarks [13]. First, we consider a well-known match-kmp benchmark. The original run time for 100,000 executions of the queries from [13] was 1.82 s. The run time of our BTA was below the measuring threshold and running LOGEN on the result also took very little time (0.00175 s). The specialised program took 1.52 s (i.e., a speedup of 1.20). For comparison, ECCE took 0.02 s to specialise the program; the resulting specialised program is faster still (0.91 s, i.e., a speedup of 2), as the online specialiser was able to construct a KMP-like specialised pattern matcher. So, our offline approach achieves some speedup, but the online specialiser is (not surprisingly) considerably better. However, the specialisation process itself is (again, not surprisingly) much faster using the offline approach.

A good example is the regular expression interpreter from [13] (benchmark regexp.r3 in Fig. 3). Here, the original took 1.71 s for 100,000 executions of the queries (r3 in [13]). Our BTA took 0.01 s, LOGEN took 0.005 s to specialise the program, which then took 0.86 s to run the queries (a speedup of 1.99). ECCE took 0.05 s to specialise the program; the specialised program runs slightly faster (0.81 s; a speedup of 2.11). So, here we obtain almost the same speedup as the

online approach but using a much faster and predictable specialisation process. A similar outcome is achieved for the ssuply benchmark, where we actally obtain the same speedup as the online approach.

For liftsolve.app (an interpreter for the ground representation specialised for append as object program) we obtain a reasonable speedup, but the specialised program generated by ECCE is more than 10 times faster. The most disappointing benchmark is probably imperative-power, where the size change analysis takes over two seconds[3] and the specialised program is slower than the original.

In summary, for the DPPD benchmarks we obtain usually very good BTA speeds (apart from imperative-power), very good specialisation speeds, along with a mixture of some good and disappointing speedups.

**PIC Emulator.** To validate the scalability of our BTA we have tried our new BTA on a larger example, the PIC processor emulator from [7]. It consists of 137 clauses and 855 lines of code. The purpose here was to specialise the PIC emulator for a particular PIC machine program, in order to run various static analyses on it.[4] The old BTA from [4] took 1 m 39 s (on a Linux server which corresponds roughly to 67 seconds on the MacBook Pro).[5] Furthermore the generated annotation file is erroneous and could not be used for specialisation. With our new BTA a correct annotation is generated less than half a second; the ensuing specialisation by LOGEN took 4.8 s. The generated code is very similar to the one obtained using a manually constructed annotation in Section 3 of [7] or in [14]. In fact, it is slightly more precise and with a simple hint (see Sect. 4.2), we were able to *reduce* specialisation so as to obtain the exact same code as [14] for the main interpreter loop. Also, with ECCE it took 9 m 30 s to construct a (very large) specialised program.

In summary, this example clearly shows that we have attained our goal of being able to successfully analyse medium-sized examples with our BTA.

**Vanilla and Lambda Interpreter.** We now turn our attention to two interpreters from [10]. For vanilla (a variation of the vanilla metainterpreter specialised for double-append as object program) we obtain a reasonable speedup, but the specialised program generated by ECCE is more than 4 times faster.

A more involved example, is the lambda interpreter for a small functional language from [10]. This interpreter contains some side-effects and non-declarative features. It can still be run through ECCE, but there is actually no guarantee that that ECCE will preserve the side-effects and their order. (This is identified by the asterisk in the table; however, in this particular case, the specialised program is correct.) Our BTA and LOGEN are very fast, but unfortunately resulting in no speedup over the original. Still, the BTA from [4] cannot cope with the program at all and we at least obtain a correct starting point. In the next subsection we

[3] This is mainly due to the large number (657) of size change graphs generated.

[4] The emulator cannot be run as is using an existing Prolog system, as the built-in arithmetic operations have to be treated like constraints.

[5] We were unable to get [4] working on the MacBook Pro and had to resort to using our webserver (with 26 MLIPS compared to the MacBook Pro's 38 MLIPS).

show that through the use of some selective hints, we can actually obtain very good speedups for this example, as well as for all examples we have seen so far.

### 4.2 Improving the Results with Hints

While the above experiments show that we have basically succeeded in obtaining a fast BTA, the specialisation results are still unsatisfactory for many examples. There are several causes for this:

1. One is a limitation of the current approach which we plan to remedy in future work: namely that when the BTA detects a dangerous loop it is sufficient to "cut" the loop once somewhere in the loop (by inserting a memoisation point in the case of local termination) and not at all points on the loop. This idea for improvement is explored further in [12].
2. Another cause regards the particular binding-time domain used. For example, the lambdaint interpreter contains an environment which is a list of bindings from variables to values, such as `[x/2, y/3]`. During specialisation, the length of the list as well as the variable names are known, and the values are unknown. However, the closest binding-time value is `list_nv`, meaning that the BTA and the specialiser would throw away the variable names (i.e., the specialiser will work with `[A/B,C/D]` rather than with `[x/B,y/D]`). One solution is to improve our BTA to work with more sophisticated binding-time domains, possibly inferring the interesting abstract values using a type inference. Another solution is a so-called "binding-time improvement" (bti) [8], whereby we rewrite the interpreter to work with two lists (i.e., `[x,y]` and `[2,3]`) rather than one. The first list (i.e., `[x,y]`) can now be classified as `static`, thereby keeping the desired information and allowing the specialiser to remove the overhead of variable lookups. We have performed this transformation for lambdaint and liftsolve.app for Fig. 4.
3. The final reason is an inherent limitation of using size-change analysis, namely the fact that the selection rule is ignored. This both gives our BTA its speed and scalability, but it also induces a precision loss. One way to solve this issue is for the user to be able to selectively insert "hints" into the source code, overriding the BTA. For the moment we support hints that force unfolding (resp. memoisation) of certain calls or predicates, as well as ways to prevent generalisation of arguments of memoised predicates. These hints can also be used as temporary fix for point 1 above.

The main idea of using hints is to have just a few of them, in the original source code in a format that a user can comprehend. Compared to editing the annotation file generated by our BTA, the advantage of hints is that the source file can still be freely edited; there is no need to synchronise annotations with edited code as in earlier work (such as the Pylogen interface [5]). Also, the propagation of binding-times is still fully performed by the BTA (and no binding-time errors can be introduced). Also, unfolding and generalisation decisions for predicates without hints are also fully taken care of by our algorithm. There is obviously the potential for a user to override the BTA in such a way that the

| benchmark | original | BTA | LOGEN | specialised | ECCE | specialised |
|---|---|---|---|---|---|---|
| contains.kmp | 0.18 | 0.01 | 0.002 | 0.18 | 0.11 | 0.02 |
| + hints | | 0.01 | 0.002 | 0.03 | | |
| imperative-power | 0.33 | 2.35 | 0.009 | 0.38 | 0.82 | 0.15 |
| + hints | | 2.36 | 0.005 | 0.21 | | |
| liftsolve.app | 0.46 | 0.02 | 0.014 | 0.28 | 0.10 | 0.02 |
| bti + hints | 0.46 | 0.02 | 0.002 | 0.03 | | |
| vanilla | 0.19 | 0.01 | 0.004 | 0.14 | 0.04 | 0.03 |
| + hints | | 0.01 | 0.002 | 0.05 | | |
| lambdaint | 0.60 | 0.17 | 0.005 | 0.61 | 0.26 | *0.02 |
| by hand | | hand | 0.002 | 0.05 | | |
| +hints | | 0.16 | 0.009 | 0.33 | | |
| bti | 0.69 | 0.17 | 0.005 | 0.67 | 0.26 | *0.02 |
| bti+hints | | 0.19 | 0.006 | 0.05 | | |
| ctl | 4.64 | 0.03 | 0.005 | 7.64 | 0.29 | 0.67 |
| + hints | - | 0.03 | 0.003 | 0.63 | | |

**Fig. 4.** Empirical Results with Hints

specialisation process will no longer terminate. Note, however, that one can still use the watchdog mode [14] to pinpoint such errors.

Figure 4 contains a selection of benchmarks from Fig. 3, where we have applied hints and sometimes also binding-time improvements. One new benchmark is ctl, a CTL model checker specialised for the formula `ef(p(unsafe))` and a parametric Petri net (see, e.g., [11]).

For vanilla, the following two hints are enough to get the Jones-optimal specialisation (i.e., the specialised program being isomorphic to the object program being interpreted):

```
'$MEMOANN'(solve_atom,1,[nv]).        '$UNFOLDCALLS'(solve(_)) :- true.
```

Without hints we do not get the optimal result for two reasons.

- First, because the termination analysis is overly conservative and does not realise that we can keep the top-level predicate symbol. The first hint remedies this. Its purpose is actually twofold, telling the BTA not to abstract nv (nonvar) binding-time values in the global control but also to raise an error if a binding-time value less precise than nv is encountered.
- Second, because the current algorithm breaks a loop at every point, while it is sufficient to break every loop once. The second hint remedies this problem, by forcing the unfold of `solve`.

For the liftsolve interpreter for ground representation, we have performed a binding-time improvement, after which one unfolding hint was sufficient to get a near optimal result. For the lambdaint interpreter, we again performed a bti, after which we get good compiled programs, corresponding almost exactly to the results obtained in [10], when hand-crafting the annotations with custom binding-times (this is the entry "hand" in Fig 4). Other large Prolog programs that we were able to successfully specialise this way were various Java Bytecode

interpreters from [6] with roughly 100 clauses. Here, we were able to reproduce the decompilation from Java bytecode to CLP from [6] using our BTA together with LOGEN. In summary, our new BTA is very fast and together with some simple hints we can get both fast and good specialisation.

## 5 Discussion and Future Work

In conclusion, we have presented a very fast BTA, able to cope with larger programs and for the first time ensuring both local and global termination. Compared to [18] we have a stronger quasi-termination result, allow non-bounded norms and have a new more precise annotation procedure. While the accuracy of our BTA is reasonable, and excellent results can be obtained with the use of a few selective hints, there is still much room for improvement.

One way to improve the accuracy of the BTA consists in also running a standard left-termination analysis (such as, e.g., the termination analysis based on the abstract binary unfoldings [3]), so that left-terminating atoms (i.e., atoms whose computation terminate using the leftmost selection rule) are marked with a new annotation call (besides unfold and memo, which keep the same meaning). Basically, while atoms annotated with unfold allow us to perform an unfolding step and then the annotations of the derived goal must be followed, atoms annotated with call are *fully* unfolded. In principle, this extension may allow us to avoid some of the loss of accuracy due to considering a *strong* termination analysis during the BTA.

Another way to improve the accuracy of the BTA is to generate semi-online annotations. In other words, instead of generating `memo` we produce the annotation `online`, which tries to unfold an atom if this is safe given the unfolding history, and marking arguments as `online` rather than `dynamic`. This should yield a fast but still precise partial evaluator: the online overhead is only applied to those places in the source code where the BTA was imprecise.

In conclusion, our BTA is well suited to be applied to larger programs. The accuracy of the annotations is not yet optimal, but in conjunction with hints we have obtained both a fast BTA with very good specialisation results.

**Acknowledgments.** We would like to thank Maurice Bruynooghe for suggesting the introduction of the new annotation call, as discussed above. We also gratefully acknowledge the anonymous referees as well as the participants of LOPSTR 2008 for many useful comments and suggestions.

## References

1. Albert, E., Puebla, G., Gallagher, J.: Non-Leftmost Unfolding in Partial Deduction of Logic Programs with Impure Predicates. In: Hill, P.M. (ed.) LOPSTR 2005. LNCS, vol. 3901, pp. 115–132. Springer, Heidelberg (2006)
2. Apt, K.R.: Introduction to logic programming. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, ch. 10, pp. 495–574. North-Holland, Amsterdam (1990)

3. Codish, M., Taboch, C.: A Semantic Basis for the Termination Analysis of Logic Programs. Journal of Logic Programming 41(1), 103–123 (1999)
4. Craig, S.-J., Gallagher, J., Leuschel, M., Henriksen, K.S.: Fully Automatic Binding Time Analysis for Prolog. In: Etalle, S. (ed.) LOPSTR 2004. LNCS, vol. 3573, pp. 53–68. Springer, Heidelberg (2005)
5. Craig, S.-J.: Practicable Prolog Specialisation. PhD thesis, University of Southampton, U.K. (June 2005)
6. Gómez-Zamalloa, M., Albert, E., Puebla, G.: Improving the decompilation of java bytecode to prolog by partial evaluation. Electr. Notes Theor. Comput. Sci. 190(1), 85–101 (2007)
7. Henriksen, K.S., Gallagher, J.P.: Abstract interpretation of pic programs through logic programming. In: SCAM, pp. 184–196. IEEE Computer Society, Los Alamitos (2006)
8. Jones, N.D., Gomard, C.K., Sestoft, P.: Partial Evaluation and Automatic Program Generation. Prentice-Hall, Englewood Cliffs (1993)
9. Leuschel, M., Bruynooghe, M.: Logic Program Specialisation through Partial Deduction: Control Issues. Theory and Practice of Logic Programming 2(4-5), 461–515 (2002)
10. Leuschel, M., Craig, S.-J., Bruynooghe, M., Vanhoof, W.: Specialising Interpreters Using Offline Partial Deduction. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 340–375. Springer, Heidelberg (2004)
11. Leuschel, M., Jørgensen, J., Vanhoof, W., Bruynooghe, M.: Offline Specialisation in Prolog using a Hand-Written Compiler Generator. Theory and Practice of Logic Programming 4(1-2), 139–191 (2004)
12. Leuschel, M., Tamarit, S., Vidal, G.: Improving size-change analysis in offline partial evaluation. In: Arenas, P., Zanardini, D. (eds.) WLPE (2008)
13. Leuschel, M.: The ECCE partial deduction system and the dppd library of benchmarks (1996-2002), `http://www.ecs.soton.ac.uk/~mal`
14. Leuschel, M., Craig, S.-J., Elphick, D.: Supervising offline partial evaluation of logic programs using online techniques. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 43–59. Springer, Heidelberg (2007)
15. Leuschel, M., Martens, B., De Schreye, D.: Controlling generalisation and polyvariance in partial deduction of normal logic programs. ACM Transactions on Programming Languages and Systems 20(1), 208–258 (1998)
16. Lindenstrauss, N., Sagiv, Y.: Automatic Termination Analysis of Logic Programs. In: Proc. ICLP 1997, pp. 63–77. MIT Press, Cambridge (1997)
17. Lloyd, J.W.: Foundations of Logic Programming. Springer, Heidelberg (1987)
18. Vidal, G.: Quasi-Terminating Logic Programs for Ensuring the Termination of Partial Evaluation. In: Proc. PEPM 2007, pp. 51–60. ACM Press, New York (2007)

# An Inference Algorithm for Guaranteeing Safe Destruction*

Manuel Montenegro, Ricardo Peña, and Clara Segura

Universidad Complutense de Madrid, Spain
C/ Prof. José García Santesmases s/n. 28040 Madrid
montenegro@fdi.ucm.es, {ricardo,csegura}@sip.ucm.es
Tel.: 91 394 7646 / 7627 / 7625; Fax: 91 394 7529

**Abstract.** *Safe* is a first-order eager functional language with destructive pattern matching controlled by the programmer. A previously presented type system is used to avoid dangling pointers arising from the inadequate usage of this facility. In this paper we present a type inference algorithm, prove its correctness w.r.t. the type system, describe its implementation and give a number of successfully typed examples.

**Keywords:** memory management, type-based analysis, type inference.

## 1 Introduction

*Safe*[1] [15,11] was introduced as a research platform for investigating the suitability of functional languages for programming small devices and embedded systems with strict memory requirements. The final aim is to be able to infer —at compile time— safe upper bounds on memory consumption for most *Safe* programs. The compiler produces as target language Java bytecode, so that *Safe* programs can be executed in most mobile devices and web navigators.

In most functional languages memory management is delegated to the runtime system. Fresh heap memory is allocated during program evaluation as long as there is enough free memory available. Garbage collection interrupts program execution in order to copy or mark the live part of the heap so that the rest is considered as free. This does not avoid memory exhaustion if not enough free memory is recovered to continue execution. The main advantage of this approach is that programmers do not have to bother about low level details concerning memory management. Its main disadvantages are:

1. The time delay introduced by garbage collection may prevent the program from providing an answer in a required reaction time.
2. Memory exhaustion may provoke unacceptable personal or economic damage to program users.

* Work supported by the projects TIN2008-06622-C03-01/TIN (STAMP), S-0505/TIC/0407 (PROMESAS) and the MEC FPU grant AP2006-02154.

[1] http://dalila.sip.ucm.es/safe

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 135–151, 2009.

3. It is difficult to predict at compile time when garbage collection will take place during execution and consequently also to determine the lifetime of data structures and to reason about memory consumption.

These reasons make this memory management unacceptable in small devices where garbage collectors are a burden both in space and in service availability. Programmers of such devices would like both to have more control over memory and to be able to reason about the memory consumption of their programs. Some works have been done in order to perform compile-time garbage collection [7,8,9], or to detect safe destructive updates of data structures [6,13]. However, these implicit approaches do not avoid completely the need for a garbage collector.

Another possibility is to use heap regions, which are parts of the heap that are dynamically allocated and deallocated. Many work has been done in order to incorporate regions in functional languages. They were introduced by Tofte and Talpin [17] in MLKit by means of a nested **letregion** construct inferred by the compiler. The drawbacks of nested regions are well-known and they have been discussed in many papers [4]. The main problem is that in practice data structures do not have the nested lifetimes required by the stack-based region discipline. In order to overcome this limitation several mechanisms have been proposed. An extension of Tofte and Talpin's work [3,16] allows to *reset* all the data structures in a region without deallocating the whole region. The AFL system [1] inserts (as a result of an analysis) allocation and deallocation commands separated from the **letregion** construct, which now only brings new regions into scope. In [4] a comparison of these works is done. In both cases, although it is not required to write in the program the memory commands, a deep knowledge about the hidden mechanism is needed in order to optimize the memory usage. In particular, it is required to write copy functions in the program which are difficult to justify without knowing the annotations inferred by the compiler.

Another more explicit approach is to introduce a language construct to free heap memory. Hofmann and Jost [5] introduce a *match* construct which destroys individual constructor cells than can be reused by the memory management system. This allows the programmer to control the memory consumed by her program and to reason about it. However, this approach gives the programmer the whole responsibility for reusing memory unless garbage collection is used.

Our functional language *Safe* is a semi-explicit approach to memory control which combines regions and a deallocation construct but with a very low effort from the programmer's point of view.

*Safe* uses implicit regions to destroy garbage. In our language regions are allocated/deallocated by following a stack discipline associated to function calls and returns. Each function call allocates a local working region, which is deallocated when the function returns. The compiler infers which data structures may be allocated in this local region because they are not needed as part of the result of the function. Region management does not add a significant runtime overhead because its related operations run in constant time [14].

In order to overcome the problems related to nested regions, *Safe* also provides the programmer with a construct **case!** to deallocate individual cells of a

data structure, so that they can be reused by the memory management system. Regions and explicit destruction are orthogonal mechanisms: we could have destruction without regions and the other way around. This combination of explicit destruction and implicit regions is novel in the functional programming field.

*Safe*'s syntax is a first-order subset of Haskell extended with destructive pattern matching. Consequently, programming in *Safe* is straightforward for Haskell programmers. They only have to write a destructive pattern matching when they want the cell to be reused (see the examples in Sec. 2). Programmer controlled destruction may create dangling references as a side effect. For this reason, we have defined a type system [11] guaranteeing that programmer destructions and region management done by the system do not create dangling pointers in the heap. A correct region inference algorithm was also described in [10].

The contribution of this paper is the description of an inference algorithm for the type system presented in [11]. This task is not trivial as the rules in the type system are non-deterministic and additionally some of them are not syntax-directed. We provide a high level description of its implementation, give some examples of its use, and also prove its correctness with respect to the type system. Our algorithm has been fully implemented as a part of our *Safe* compiler and has an average time cost near to $\Theta(n^2)$ for each function definition, where $n$ is the size of its abstract syntax tree (see Sec. 4).

In Sec. 2 we summarize the language. The type system is presented in Sec. 3 and the corresponding inference algorithm is explained and proven correct in Sec. 4. Section 5 shows some examples whose types have been successfully inferred. Finally, Sec. 6 compares this work with related analyses in other languages with memory management facilities.

## 2 Summary of *Safe*

*Safe* is a first-order polymorphic functional language with some facilities to manage memory. These are destructive pattern matching, copy and reuse of data structures. We explain them with examples below.

Destructive pattern matching, denoted by ! or a **case!** expression, deallocates the cell corresponding to the outermost constructor. In this case we say that the data structure is *condemned*. As an example, we show an append function destroying the first list's spine, while keeping its elements in order to build the result. Using recursion the recursive spine of the first list is deallocated:

```
concatD []!     ys  = ys
concatD (x:xs)! ys  = x : concatD xs ys
```

This version of appending needs constant additional heap space (a cell is destroyed and another one is created at each call), while the usual version needs additional linear heap space. The fact that the first list is lost must be remembered by the programmer as he will not be able to use it anymore in the program. This is reflected in the type of the function: `concatD :: [a]! -> [a] -> [a]`.

$$
\begin{array}{lll}
prog \rightarrow \overline{data_i}^{\,n}\,;\,\overline{dec_j}^{\,m}\,;\,e & & \\
data \rightarrow \textbf{data}\ T\ \overline{\alpha_i}^{\,n}\ @\ \overline{\rho_j}^{\,m} = \overline{C_k\ \overline{t_{ks}}^{\,n_k}\ @\ \rho_m}^{\,l} & & \{\text{recursive, polymorphic data type}\} \\
dec \rightarrow f\ \overline{x_i}^{\,n}\ @\ \overline{r_j}^{\,l} = e & & \{\text{recursive, polymorphic function}\} \\
e \rightarrow a & & \{\text{atom: literal } c \text{ or variable } x\} \\
\quad \mid x\ @\ r & & \{\text{copy}\} \\
\quad \mid x! & & \{\text{reuse}\} \\
\quad \mid f\ \overline{a_i}^{\,n}\ @\ \overline{r_j}^{\,l} & & \{\text{function application}\} \\
\quad \mid \textbf{let}\ x_1 = be\ \textbf{in}\ e & & \{\text{non-recursive, monomorphic}\} \\
\quad \mid \textbf{case}\ x\ \textbf{of}\ \overline{alt_i}^{\,n} & & \{\text{read-only case}\} \\
\quad \mid \textbf{case!}\ x\ \textbf{of}\ \overline{alt_i}^{\,n} & & \{\text{destructive case}\} \\
alt \rightarrow C\ \overline{x_i}^{\,n} \rightarrow e & & \\
be \rightarrow C\ \overline{a_i}^{\,n}\ @\ r & & \{\text{constructor application}\} \\
\quad \mid e & &
\end{array}
$$

**Fig. 1.** *Core-Safe* language definition

Data structures may also be copied by using a copy expression (denoted by @). Only the recursive spine of the structure is copied, while the elements are shared with the old one. This allows more control over sharing of data structures. In the following function

```
concat []     ys  = ys @
concat (x:xs) ys  = x : concat xs ys
```

the resulting list only shares the elements with the input lists. We could safely destroy this list while preserving the original ones.

When a data structure is condemned by a destructive pattern matching, its recursive children are also condemned in order to avoid dangling pointers (see the type system of Sec. 3). This means that they cannot be returned as part of the result of a function even when they are not explicitly destroyed. This would force us to copy those data substructures if we want them as part of the result, which would be costly. As an example, consider the following destructive tail function: `tailD (x:xs)! = xs@`. The sublist `xs` is condemned so it cannot be returned as result of `tailD`. We need to copy it if we want to keep safety. In order to avoid this costly copies the language offers a safe reuse operator ! which allows to turn a condemned data structure into a safe one so that it can be part of the result of a function. The original reference is no longer accessible in order to keep safety. In the previous example we would write `tailD (x:xs)! = xs!`.

We show another example whose type is successfully inferred. The following function is the destructive version of insertion in a binary search tree:

```
insertD :: Int -> Tree Int! -> Tree Int
insertD x Empty! = Node Empty x Empty
insertD x (Node lt y rt)! | x == y = Node lt! y  rt!
                          | x > y  = Node lt! y (insertD x rt)
                          | x < y  = Node (insertD x lt) y rt!
```

In the first guard the cell just destroyed must be built again since `lt` and `rt` are condemned; they must be reused in order to be part of the result.

$$
\begin{array}{lll}
\tau \rightarrow & t & \{\text{external}\} \\
\mid & r & \{\text{in-danger}\} \\
\mid & \sigma & \{\text{polymorphic function}\} \\
\mid & \rho & \{\text{region}\} \\
t \rightarrow & s & \{\text{safe}\} \\
\mid & d & \{\text{condemned}\} \\
s \rightarrow & T\ \overline{s}@\overline{\rho} & \\
\mid & b & \\
d \rightarrow & T\ \overline{t}!@\overline{\rho} &
\end{array}
\qquad
\begin{array}{lll}
r & \rightarrow & T\ \overline{s}\#@\overline{\rho} \\
b & \rightarrow & a \qquad\qquad\qquad\qquad \{\text{variable}\} \\
 & \mid & B \qquad\qquad\qquad\qquad \{\text{basic}\} \\
tf & \rightarrow & \overline{t_i}^n \rightarrow \overline{\rho_j}^l \rightarrow T\ \overline{s}@\overline{\rho_k}^m \quad \{\text{function}\} \\
 & \mid & \overline{s_i}^n \rightarrow \rho \rightarrow T\ \overline{s}@\overline{\rho_k}^m \quad \{\text{constructor}\} \\
\sigma & \rightarrow & \forall a.\sigma \\
 & \mid & \forall \rho.\sigma \\
 & \mid & tf
\end{array}
$$

**Fig. 2.** Type expressions

### 2.1 *Core-Safe*

*Full-Safe* is desugared into an intermediate language called *Core-Safe* where regions are explicit. However, regions can be completely ignored in this paper, as the inference algorithm explained here only concerns destruction. We just show them for completeness. In Fig. 1 we show the syntax of *Core-Safe*. A program *prog* is a sequence of possibly recursive polymorphic data and function definitions followed by a main expression $e$, using them, whose value is the program result. The abbreviation $\overline{x_i}^n$ stands for $x_1 \cdots x_n$. Destructive pattern matching is desugared into **case!** expressions. Constructions are only allowed in **let** bindings, and atoms are used in function applications, **case**/**case!** discriminant, copy and reuse. Regions are explicit in constructor application and in the copy expression. Function definitions have additional parameters $\overline{r_j}^l$ where data structures may be built. In the right hand side expression only the $r_j$ and its local working region may be used. As an example, let us consider the *Core-Safe* code of the function `concatD`:

$$
\begin{array}{l}
\mathit{concatD}\ \mathit{zs}\ \mathit{ys}\ @\ r = \mathbf{case!}\ \mathit{zs}\ \mathbf{of} \\
\qquad\qquad [\,] \rightarrow \mathit{ys} \\
\qquad\qquad (x : \mathit{xs}) \rightarrow \mathbf{let}\ x_1 = \mathit{concatD}\ \mathit{xs}\ \mathit{ys}\ @\ r\ \mathbf{in} \\
\qquad\qquad\qquad\qquad \mathbf{let}\ x_2 = (x : x_1)@r\ \mathbf{in}\ x_2
\end{array}
$$

In this case the only regions involved are those of the input lists and the output region $r$ where the result is built. The local region of each `concatD` call remains empty during its execution, since nothing is built there.

## 3 Safe Type System

In this section we briefly describe a polymorphic type system with algebraic data types for programming in a safe way when using the destruction facilities offered by the language (see [11] for more details). The syntax of type expressions is shown in Fig. 2. As the language is first-order, we distinguish between functional, $tf$, and non-functional types, $t, r$. Non-functional algebraic types may be safe types $s$, condemned types $d$ or in-danger types $r$. In-danger and condemned types are respectively distinguished by a # or ! annotation. In-danger types arise as an intermediate step during typing and are useful to control the side-effects of the destructions. However, the types of functions only include either safe or condemned types. The intended semantics of these types is the following:

- ***Safe* types ($s$):** A DS of this type can be read, copied or used to build other DSs. They cannot be destroyed or reused by using the symbol !. The predicate *safe*? tells us whether a type is safe.
- ***Condemned* types ($d$):** It is a DS directly involved in a **case**! action. Its recursive descendants inherit the same condemned type. They cannot be used to build other DSs, but they can be read or copied before being destroyed. They can also be reused once. The predicate *cdm*? is true for them.
- ***In-danger* types ($r$):** This is a DS sharing a recursive descendant of a condemned DS, so it can potentially contain dangling pointers. The predicate *danger*? is true for these types. The predicate *unsafe*? is true for condemned and in-danger types. Function $danger(s)$ denotes the in-danger version of $s$.

We will write $T@\overline{\rho}^m$ instead of $T\ \overline{s}@\overline{\rho}^m$ to abbreviate whenever the $\overline{s}$ are not relevant. We shall even use $T@\rho$ to highlight only the outermost region. A partial order between types is defined: $\tau \geq \tau$, $T!@\overline{\rho}^m \geq T@\overline{\rho}^m$, and $T\#@\overline{\rho}^m \geq T@\overline{\rho}^m$.

Predicates $region?(\tau)$ and $function?(\tau)$ respectively indicate that $\tau$ is a region type or a functional type.

Constructor types have one region argument $\rho$ which coincides with the outermost region variable of the resulting algebraic type $T\ \overline{s}@\overline{\rho}^m$, and reflect that recursive sharing can happen only in the same region. As example:

$$
\begin{array}{l}
[\,] : \forall a, \rho.\rho \to [a]@\rho \\
(:) : \forall a, \rho.a \to [a]@\rho \to \rho \to [a]@\rho \\
\mathit{Empty} : \forall a, \rho.\rho \to \mathit{Tree}\ a@\rho \\
\mathit{Node} : \forall a, \rho.\mathit{Tree}\ a@\rho \to a \to \mathit{Tree}\ a@\rho \to \rho \to \mathit{Tree}\ a@\rho
\end{array}
$$

We assume that the types of the constructors are collected in an environment $\Sigma$, easily built from the **data** type declarations. In functional types there may be several region arguments $\overline{\rho_j}^l$ where data structures may be built.

In the type environments, $\Gamma$, we can find region type assignments $r : \rho$, variable type assignments $x : t$, and polymorphic scheme assignments to functions $f : \sigma$. In the rules we will also use $gen(tf, \Gamma)$ and $tf \trianglelefteq \sigma$ to respectively denote (standard) generalization of a monomorphic type and restricted instantiation of a polymorphic type with safe types.

Several operators on environments are used in the rules. The usual operator $+$ demands disjoint domains. Operators $\otimes$ and $\oplus$ are defined only if common variables have the same type, which must be safe in the case of $\oplus$. If one of this operators is not defined in a rule, we assume that the rule cannot be applied. Operator $\rhd^L$ is explained below. The predicate $utype?(t, t')$ is true when the underlying Hindley-Milner types of $t$ and $t'$ are the same.

In Fig. 3, the rules for typing expressions are shown. Function $sharerec(x, e)$ gives an upper approximation to the set of variables in scope in $e$ which share a recursive descendant of the DS starting at $x$. This set is computed by the abstract interpretation based sharing analysis defined in [15].

An invariant of the type system tells that if a variable appears as condemned in the typing environment, then those variables sharing a recursive substructure appear also in the environment with unsafe types. This is necessary in order to propagate information about the possibly damaged pointers.

$$\frac{\Gamma \vdash e : s \quad x \notin dom(\Gamma) \quad safe?(\tau) \vee danger?(\tau) \vee region?(\tau) \vee function?(\tau)}{\Gamma + [x:\tau] \vdash e : s}\ [\text{EXTS}] \qquad \frac{\Gamma \vdash e : s \quad x \notin dom(\Gamma) \quad R = sharerec(x,e) - \{x\} \quad \Gamma_R = \{y : danger(type(y)) |\ y \in R\}}{\Gamma \otimes \Gamma_R + [x:d] \vdash e : s}\ [\text{EXTD}]$$

$$\frac{}{\emptyset \vdash c : B}\ [\text{LIT}] \qquad \frac{}{[x:s] \vdash x : s}\ [\text{VAR}] \qquad \frac{}{[r:\rho] \vdash r : \rho}\ [\text{REGION}] \qquad \frac{tf \trianglelefteq \sigma}{[f:\sigma] \vdash f : tf}\ [\text{FUNCTION}]$$

$$\frac{R = sharerec(x, x!) - \{x\} \quad \Gamma_R = \{y : danger(type(y)) |\ y \in R\}}{\Gamma_R + [x : T!@\rho] \vdash x! : T@\rho}\ [\text{REUSE}] \qquad \frac{\Gamma_1 \vdash e_1 : s_1 \quad \Gamma_2 + [x_1 : \tau_1] \vdash e_2 : s \quad utype?(\tau_1, s_1)}{\Gamma_1 \rhd^{fv(e_2)} \Gamma_2 \vdash \textbf{let } x_1 = e_1 \textbf{ in } e_2 : s}\ [\text{LET}]$$

$$\frac{\Gamma_1 \geq_{x@r} [x : T@\rho', r : \rho]}{\Gamma_1 \vdash x@r : T\ @\rho}\ [\text{COPY}] \qquad \frac{\Sigma(C) = \sigma \quad \overline{s_i}^n \to \rho \to T\ @\overline{\rho}^m \trianglelefteq \sigma \quad \Gamma = \bigoplus_{i=1}^n [a_i : s_i] + [r : \rho]}{\Gamma \vdash C\ \overline{a_i}^n @ r : T\ @\overline{\rho}^m}\ [\text{CONS}]$$

$$\frac{\begin{array}{c}\overline{t_i}^n \to \overline{\rho_j}^l \to T\ @\overline{\rho}^m \trianglelefteq \sigma \qquad \Gamma = [f:\sigma] + \bigoplus_{j=1}^l [r_j : \rho_j] + \bigoplus_{i=1}^n [a_i : t_i] \\ R = \bigcup_{i=1}^n \{sharerec(a_i, f\ \overline{a_i}^n @ \overline{r_j}^l) - \{a_i\} \mid cdm?(t_i)\} \quad \Gamma_R = \{y : danger(type(y)) |\ y \in R\}\end{array}}{\Gamma_R + \Gamma \vdash f\ \overline{a_i}^n @\ \overline{r_j}^l : T\ @\overline{\rho}^m}\ [\text{APP}]$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}.\Sigma(C_i) = \sigma_i \qquad \forall i \in \{1..n\}.\overline{s_i}^{n_i} \to \rho \to T\ @\rho \trianglelefteq \sigma_i \\ \Gamma \geq_{\textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n} [x : T@\rho] \quad \forall i \in \{1..n\}.\forall j \in \{1..n_i\}.inh(\tau_{ij}, s_{ij}, \Gamma(x)) \\ \forall i \in \{1..n\}.\Gamma + \overline{[x_{ij} : \tau_{ij}]}^{n_i} \vdash e_i : s\end{array}}{\Gamma \vdash \textbf{case } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n : s}\ [\text{CASE}]$$

$$\frac{\begin{array}{ll}(\forall i \in \{1..n\}).\ \Sigma(C_i) = \sigma_i & \forall i \in \{1..n\}.\ \overline{s_i}^{n_i} \to \rho \to T\ @\rho \trianglelefteq \sigma_i \\ R = sharerec(x, \textbf{case! } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n) - \{x\} & \forall i \in \{1..n\}.\ \forall j \in \{1..n_i\}.inh!(t_{ij}, s_{ij}, T\ !@\rho) \\ \forall z \in R \cup \{x\}, i \in \{1..n\}.z \notin fv(e_i) & \forall i \in \{1..n\}.\ \Gamma + [x : T\ \#@\rho] + \overline{[x_{ij} : t_{ij}]}^{n_i} \vdash e_i : s \\ \multicolumn{2}{c}{\Gamma_R = \{y : danger(type(y)) \mid y \in R\}}\end{array}}{\Gamma_R \otimes \Gamma + [x : T\ !@\rho] \vdash \textbf{case! } x \textbf{ of } \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n : s}\ [\text{CASE!}]$$

**Fig. 3.** Type rules for expressions

There are rules for typing literals ([LIT]), and variables of several kinds ([VAR], [REGION] and [FUNCTION]). Notice that these are given a type under the smallest typing environment. Rules [EXTS] and [EXTD] allow to extend the typing environments according to the invariant mentioned above. Notation $type(y)$ represents the Hindley-Milner type inferred for variable $y$[2].

Rule [COPY] allows any variable to be copied. This is expressed by extending the previously defined partial order between types to environments.

Rule [LET] controls the intermediate data structures, that may be safe, condemned or in-danger in the main expression ($\tau$ covers the three cases). Operator $\rhd^L$ guarantees that: (1) Each variable $y$ condemned or in-danger in $e_1$ may not be referenced in $e_2$ (i.e. $y \notin fv(e_2)$), as it could be a dangling reference. (2) Those variables marked as unsafe either in $\Gamma_1$ or in $\Gamma_2$ will keep those types in the combined environment.

Rule [REUSE] establishes that in order to reuse a variable, it must have a condemned type in the environment. Those variables sharing its recursive descendants are given in-danger types in the environment.

Rule [APP] deals with function application. The use of the operator $\oplus$ avoids a variable to be used in two or more different positions unless they are all safe parameters. Otherwise undesired side-effects could happen. The set $R$ collects all the variables sharing a recursive substructure of a condemned parameter, which

[2] Inference implementation first infers H-M types and then destruction annotations.

are marked as in-danger in environment $\Gamma_R$. Rule [CONS] is more restrictive as only safe variables can be used to construct a DS.

Rule [CASE] allows its discriminant variable to be safe, in-danger, or condemned as it only reads the variable. Relation *inh* determines which types are acceptable for pattern variables. Apart from the fact that the underlying types are correct from the Hindley-Milner point of view: if the discriminant is safe, so must be all the pattern variables; if it is in-danger, the pattern variables may be safe or in-danger; if it is condemned, recursive pattern variables are in-danger while non-recursive ones are safe.

In rule [CASE!] the discriminant is destroyed and consequently the text should not try to reference it in the alternatives. The same happens to those variables sharing a recursive substructure of $x$, as they may be corrupted. All those variables are added to the set $R$. Relation *inh*! determines the types inherited by pattern variables: recursive ones are condemned while non-recursive ones are safe. As recursive pattern variables inherit condemned types, the type environments for the alternatives contain all the variables sharing their recursive substructures as in-danger. In particular $x$ may appear with an in-danger type. In order to type the whole expression we must change it to condemned.

## 4 Inference Algorithm

The typing rules presented in Sec. 3 allow in principle several correct typings for a program. On the one hand, this is due to polymorphism and, on the other hand, to the fact that it may assign more condemned and in-danger types that those really needed. We are interested in *minimal* types in the sense of being as much polymorphic as possible and having as few unsafe types as possible.

As an example, let us consider the following definition: `f (x:xs) = xs@`. The type system can give $f$ type $[a] \to [a]$ but also the type $[a]! \to [a]$. Our inference algorithm will return the first one.

Also, we are not interested in having mandatory explicit type declarations. This is what the inference algorithm presented in this section achieves. It has two different phases: a (modified) Hindley-Milner phase and an unsafety propagation phase. The first one is rather straightforward with the added complication of region inference, which is done at this stage. Its output consists of decorating each applied occurrence of a variable and each defining occurrence of a function symbol in the abstract syntax tree (AST) with its Hindley-Milner type. We will not insist further in this phase here.

The second phase propagates unsafety information from the parts of the text where condemned and in-danger types arise to the rest of the program text. As the Hindley-Milner types are already available, the only additional information needed for each variable is a *mark* telling whether it is a safe, in-danger or condemned one. Condemned and in-danger marks arise for instance in the [CASE!], [REUSE], and [APP] typing rules while mandatory safe marks arise for instance in rules for constructor applications. The algorithm generates minimal sets of these marks in the program sites where they are mandatory and propagates this

$$\frac{}{c \vdash_{inf} (\emptyset,\emptyset,\emptyset,\emptyset)}\ [\text{LIT}_I] \qquad \frac{}{x \vdash_{inf} (\emptyset,\emptyset,\{x\},\emptyset)}\ [\text{VAR}_I] \qquad \frac{}{x@r \vdash_{inf} (\emptyset,\emptyset,\emptyset,\{x\})}\ [\text{COPY}_I]$$

$$\frac{R = \mathit{sharerec}(x, x!) - \{x\} \quad \mathit{type}(x) = T@\,\rho}{x! \vdash_{inf} (\{x\}, R, \emptyset, \emptyset)}\ [\text{REUSE}_I] \qquad \frac{\forall i \in \{1..n\}.a_i \vdash_{inf} (\emptyset,\emptyset,S_i,\emptyset)}{C\ \overline{a_i}^n\ @r \vdash_{inf} (\emptyset,\emptyset,\bigcup_{i=1}^n S_i,\emptyset)}\ [\text{CONS}_I]$$

$$\frac{\begin{array}{c}\forall i \in \{1..n\}.D_i = \{a_i \mid i \in I_D\} \quad (\bigcup_{i=1}^n D_i) \cap (\bigcup_{i=1}^n S_i) = \emptyset \qquad R \cap (\bigcup_{i=1}^n S_i) = \emptyset \\ \forall i \in \{1..n\}.S_i = \{a_i \mid i \in I_S\} \quad (\bigcup_{i=1}^n D_i) \cap (\bigcup_{i=1}^n N_i) = \emptyset \qquad R \cap (\bigcup_{i=1}^n D_i) = \emptyset \\ \forall i \in \{1..n\}.N_i = \{a_i \mid i \in I_N\} \quad \forall i,j \in \{1..n\}\,.\, i \neq j \Rightarrow D_i \cap D_j = \emptyset \quad R \cap (\bigcup_{i=1}^n N_i) = \emptyset \\ \Sigma \vdash f : (I_D, \emptyset, I_S, I_N) \qquad R = \bigcup_{i=1}^n \{\mathit{sharerec}(a_i, f\ \overline{a_i}^n\ @\overline{r_j}^l) - \{a_i\} \mid a_i \in D_i\}\end{array}}{f\ \overline{a_i}^n\ @\overline{r_j}^l \vdash_{inf} (\bigcup_{i=1}^n D_i, R, \bigcup_{i=1}^n S_i, (\bigcup_{i=1}^n N_i) - (\bigcup_{i=1}^n S_i))}\ [\text{APP}_I]$$

$$\frac{\begin{array}{c}e_1 \vdash_{inf} (D_1, R_1, S_1, N_1) \qquad (D_1 \cup R_1) \cap \mathit{fv}(e_2) = \emptyset \\ e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \qquad N = (N_1 - (D_2 \cup R_2 \cup S_2)) \cup N_2 \\ (\emptyset,\emptyset,N_1 \cap (D_2 \cup R_2 \cup S_2)) \vdash_{check} e_1 \quad (\emptyset,\emptyset,(S_1 \cup \{x_1\}) \cap N_2) \vdash_{check} e_2\end{array}}{\begin{array}{l}\textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2 \vdash_{inf} ((D_1 \cup D_2) - \{x_1\}, R_1 \cup (R_2 - (D_1 \cup \{x_1\}))), \\ \qquad\qquad ((S_1 - N_2) \cup S_2) - (\{x_1\} \cup D_2 \cup R_2), N - \{x_1\})\end{array}}\ [\text{LET}_I]$$

$$\frac{\begin{array}{l}\forall i \in \{1..n\}\,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \qquad \mathit{def}(\sqcup_{i=1}^n (D_i, R_i, S_i, N_i, P_i)) \\ \forall i \in \{1..n\}\,.\, P_i = \bigcup_{j=1}^{n_i} \{x_{ij}\} \qquad \forall i \in \{1..n\}\,.\, \mathit{def}(\mathit{inh}(\mathit{type}(x), D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \\ \forall i \in \{1..n\}\,.\, \mathit{Rec}_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in \mathit{RecPos}(C_i)\} \\ \mathit{type}(x) = \begin{cases} d \text{ if } x \in D \\ r \text{ if } x \in R \\ s \text{ if } x \in S \\ n \text{ e. o. c.} \end{cases} \qquad (D,R,S,N) = \sqcup_{i=1}^n (D_i, R_i, S_i, N_i, P_i) \\ \qquad\qquad N' = \begin{cases} N & \text{if } x \in D \cup R \cup S \\ N \cup \{x\} & \text{if } x \notin D \cup R \cup S \end{cases} \\ \forall i \in \{1..n\}\,.\, ((D \cup D_i') \cap N_i, R \cup ((R_i' \cup R_i'' \cup R_i''') - D_i), (S \cup S_i') \cap N_i) \vdash_{check} e_i \\ \textbf{where}\ \ D_i' = \emptyset \qquad R_i' = \begin{cases} \mathit{Rec}_i \text{ if } \mathit{type}(x) = d \\ \emptyset \quad \text{otherwise} \end{cases} \qquad S_i' = \begin{cases} P_i - \mathit{Rec}_i & \text{if } \mathit{type}(x) = d \\ P_i - R_i'' & \text{if } \mathit{type}(x) = r \\ P_i & \text{if } \mathit{type}(x) = s \\ \emptyset & \text{otherwise} \end{cases} \\ \qquad R_i'' = \{y \in P_i \cap \mathit{sharerec}(z, e_i) \mid z \in (D \cup D_i') \cap N_i\} \\ \qquad R_i''' = \{y \in D \cap \mathit{sharerec}(z, e_i) \mid z \in (D \cup D_i') \cap N_i\} - (D \cap N_i) \\ \qquad R_i' \cap (S_i \cup S_i') = \emptyset\end{array}}{\textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \vdash_{inf} (D,R,S,N')}\ [\text{CASE}_I]$$

$$\frac{\begin{array}{l}\forall i \in \{1..n\}\,.\, e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \qquad \mathit{def}(\sqcup_{i=1}^n (D_i, R_i, S_i, N_i, P_i)) \\ \forall i \in \{1..n\}\,.\, P_i = \bigcup_{j=1}^{n_i} \{x_{ij}\} \qquad \forall i \in \{1..n\}\,.\, \mathit{def}(\mathit{inh}!(D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \\ \forall i \in \{1..n\}\,.\, \mathit{Rec}_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in \mathit{RecPos}(C_i)\} \qquad R \cap L = \emptyset\ \wedge\ \mathit{type}(x) = T@\rho \\ R = \mathit{sharerec}(x, \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n) \\ L = \bigcup_{i=1}^n \mathit{fv}(e_i) \qquad (D,R',S,N) = \sqcup_{i=1}^n (D_i, R_i, S_i, N_i, P_i) \\ \forall i \in \{1..n\}\,.\, ((D \cup \mathit{Rec}_i) \cap N_i, R \cup R' \cup (R_i' \cup R_i'') - D_i, (S \cup (P_i - \mathit{Rec}_i)) \cap N_i) \vdash_{check} e_i \\ \textbf{where}\ \ R_i' = \{y \in P_i \cap \mathit{sharerec}(z, e_i) \mid z \in (D \cup \mathit{Rec}_i) \cap N_i\} - (\mathit{Rec}_i \cap N_i) \\ \qquad R_i'' = \{y \in D \cap \mathit{sharerec}(z, e_i) \mid z \in D \cap N_i\} - (D \cap N_i) \\ \qquad R_i' \cap (P_i - \mathit{Rec}_i) = \emptyset\ \wedge\ \{y \in \mathit{sharerec}(z, e_i) \mid z \in \mathit{Rec}_i\} \cap (P_i - \mathit{Rec}_i) = \emptyset\end{array}}{\textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^n \vdash_{inf} (D \cup \{x\}, (R \cup R') - \{x\}, S, N)}\ [\text{CASE!}_I]$$

**Fig. 4.** Bottom-up inference rules

$$\begin{array}{l}\mathit{def}(\mathit{inh}(n, D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \equiv \mathit{true} \\ \mathit{def}(\mathit{inh}(s, D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \equiv P_i \cap (D_i \cup R_i) = \emptyset \\ \mathit{def}(\mathit{inh}(r, D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \equiv P_i \cap D_i = \emptyset \\ \mathit{def}(\mathit{inh}(d, D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \equiv \mathit{Rec}_i \cap (D_i \cup S_i) = \emptyset\ \wedge\ (P_i - \mathit{Rec}_i) \cap (D_i \cup R_i) = \emptyset \\ \mathit{def}(\mathit{inh}!(D_i, R_i, S_i, P_i, \mathit{Rec}_i)) \equiv \mathit{Rec}_i \cap (R_i \cup S_i) = \emptyset\ \wedge\ (P_i - \mathit{Rec}_i) \cap (D_i \cup R_i) = \emptyset\end{array}$$

**Fig. 5.** Predicates *inh* and *inh*!

$$def(\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i)) \equiv \forall i, j \in \{1..n\} \,.\, i \neq j \Rightarrow \begin{array}{l}(D_i - P_i) \cap (R_j - P_j) = \emptyset \wedge \\ (D_i - P_i) \cap (S_j - P_j) = \emptyset \wedge (R_i - P_i) \cap (S_j - P_j) = \emptyset\end{array}$$

$$\sqcup_{i=1}^{n}(D_i, R_i, S_i, N_i, P_i) \stackrel{\text{def}}{=} (D, R, S, N) \textbf{ where } \begin{cases} D = \bigcup_{i=1}^{n}(D_i - P_i) & R = \bigcup_{i=1}^{n}(R_i - P_i) \\ S = \bigcup_{i=1}^{n}(S_i - P_i) & N = \left(\bigcup_{i=1}^{n}(N_i - P_i)\right) - (D \cup R \cup S) \end{cases}$$

**Fig. 6.** Least upper bound definitions

information bottom-up in the AST looking for consistency of the marks. It may happen that a safe mark is inferred for a variable in a program site and a condemned mark is inferred for the same variable in another site. This sometimes is allowed by the type system —e.g. it is legal to read a variable in the auxiliary expression of a **let** and to destroy it in the main expression—, and disallowed some other times—e.g. in a **case**, it is not legal to have a safe type for a variable in one alternative and a condemned or in-danger type for it in another alternative.

So, the algorithm has two working modes. In the bottom-up working mode, it accumulates sets of marks for variables. In fact, it propagates bottom-up four sets of variables $(D, R, S, N)$ respectively meaning condemned, in-danger, safe, and don't-know variables in the corresponding expression. The fourth set arises from the non-deterministic typing rules for [COPY] and [CASE] expressions.

The algorithm checks for consistency the information coming from two or more different branches of the AST. This happens for instance in **let** and **case** expressions. Even though the information is consistent it may be necessary to propagate some information down the AST. For instance, $x \in D_1$ and $x \in N_2$ is consistent in two different branches 1 and 2 of a **case** or a **case!**, but a $D$ mark for $x$ must be propagated down the branch 2.

So, the algorithm consists of a single bottom-up traversal of the AST, occasionally interrupted by top-down traversals when new information must be propagated in one or more branches. If the propagation does not raise an error, then the bottom-up phase is resumed.

In Fig. 4 we show the rules that drive the bottom-up working mode. A judgement of the form $e \vdash_{inf} (D, R, S, N)$ should be read as: from expression $e$ the 4-tuple $(D, R, S, N)$ of marked variables is inferred. A straightforward invariant of this set of rules is that the four sets inferred for each expression $e$ are pairwise disjoint and their union is a superset of $e$'s free variables. The set $R$ may contain variables in scope but not free in $e$. This is due to the use of the set *sharerec* consisting of *all* variables in scope satisfying the sharing property. The predicates and least upper bound appearing in the rules [CASE$_I$] and [CASE!$_I$] are defined in Figs. 5 and 6.

In Fig. 7 we show the top-down checking rules. A judgement $(D, R, S) \vdash_{check} e$ should be understood that the sets of marked variables $D, R, S$ are correctly propagated down the expression $e$. One invariant in this case is that the three sets are pairwise disjoint and that the union of $D$ and $S$ is contained in the fourth set $N$ inferred from the expression by the $\vdash_{inf}$ rules. It can be seen that the $\vdash_{inf}$ rules may invoke the $\vdash_{check}$ rules. However, the $\vdash_{check}$ rules do not invoke the $\vdash_{inf}$ ones. The occurrences of $\vdash_{inf}$ in the $\vdash_{check}$ rules should be interpreted as a

$$\frac{}{(\emptyset, R, \emptyset) \vdash_{check} c}\ [\text{LIT}_C] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x}\ [\text{VAR}_C] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x!}\ [\text{REUSE}_C]$$

$$\frac{}{(\{x\}, R, \emptyset) \vdash_{check} x@r}\ [\text{COPY1}_C] \qquad \frac{}{(\emptyset, R, \emptyset) \vdash_{check} x@r}\ [\text{COPY2}_C] \qquad \frac{}{(\emptyset, R, \{x\}) \vdash_{check} x@r}\ [\text{COPY3}_C]$$

$$\frac{}{(\emptyset, R, \emptyset) \vdash_{check} C\ \overline{a_i}^n\ @r}\ [\text{CONS}_C] \qquad \frac{f\ \overline{a_i}^n\ @\overline{r_j}^l \vdash_{inf} (D, R, S, N) \quad \forall a_i \in D_p\ .\ (\#j : 1 \le j \le n : a_i = a_j) = 1}{(D_p, R_p, S_p) \vdash_{check} f\ \overline{a_i}^n @ \overline{r_j}^l}\ [\text{APP}_C]$$

$$\frac{\begin{array}{c} e_1 \vdash_{inf} (D_1, R_1, S_1, N_1)\ R_p \cap S_1 = \emptyset\ \wedge\ ((D_p \cap N_1) \cup R_p \cup R''_p) \cap fv(e_2) = \emptyset \\ e_2 \vdash_{inf} (D_2, R_2, S_2, N_2) \quad \exists z \in D_p \cap N_2\ .\ x_1 \in sharerec(z, e_2) \Rightarrow x_1 \in D_2 \\ (D_p \cap N_1, R_p, S_p \cap N_1) \vdash_{check} e_1 \qquad (D_p \cap N_2, R_p \cup (R'_p - D_2), S_p \cap N_2) \vdash_{check} e_2 \\ \textbf{where}\ R'_p = \{y \in ((D_p \cap N_1) \cup D_1) \cap sharerec(z, e_2) \mid z \in D_p \cap N_2\} - (N_2 \cup \{x_1\}) \\ R''_p = \{y \in sharerec(z, e_1) \mid z \in D_p \cap N_1\} \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{let}\ x_1 = e_1\ \textbf{in}\ e_2}\ [\text{LET}_C]$$

$$\frac{\begin{array}{c} \forall i \in \{1..n\}\ .\ e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \\ \forall i \in \{1..n\}\ .\ P_i = \bigcup_{j=1}^{n_i} \{x_{ij}\} \\ \forall i \in \{1..n\}\ .\ Rec_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in RecPos(C_i)\} \\ D = \bigcup_{i=1}^{n} (D_i - P_i) \\ x \in D_p \cup R_p \cup S_p \Rightarrow \forall i \in \{1..n\}\ .\ def(inh(type(x), D_i, R_i, S_i, P_i, Rec_i)) \end{array} \quad type(x) = \begin{cases} d \text{ if } x \in D_p \\ r \text{ if } x \in R_p \\ s \text{ if } x \in S_p \\ n \text{ otherwise} \end{cases}}{\begin{array}{c} \forall i \in \{1..n\}\ .\ ((D_p \cup D_{p_i}) \cap N_i, (R_p \cup R_{p_i} \cup R'_{p_i} \cup R''_{p_i}) - D_i, (S_p \cup S_{p_i}) \cap N_i) \vdash_{check} e_i \\ \textbf{where}\ D_{p_i} = \emptyset \qquad R_{p_i} = \begin{cases} Rec_i \text{ if } type(x) = d \\ \emptyset \quad \text{otherwise} \end{cases} \qquad S_{p_i} = \begin{cases} P_i - Rec_i & \text{if } type(x) = d \\ P_i - R'_{p_i} & \text{if } type(x) = r \\ P_i & \text{if } type(x) = s \\ \emptyset & \text{otherwise} \end{cases} \end{array}}\ [\text{CASE}_C]$$

$$\frac{\begin{array}{c} R'_{p_i} = \{y \in P_i \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} \\ R''_{p_i} = \{y \in (D_p \cup D) \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} - (D_p \cap N_i) \\ R'_{p_i} \cap (S_i \cup S_{p_i}) = \emptyset\ \wedge\ R_p \cap S_i = \emptyset \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}\ [\text{CASE}_C]$$

$$\frac{\begin{array}{c} \forall i \in \{1..n\}\ .\ e_i \vdash_{inf} (D_i, R_i, S_i, N_i) \\ \forall i \in \{1..n\}\ .\ P_i = \bigcup_{j=1}^{n_i} \{x_{ij}\} \qquad D = \bigcup_{i=1}^{n} (D_i - P_i) \\ \forall i \in \{1..n\}\ .\ Rec_i = \bigcup_{j=1}^{n_i} \{x_{ij} \mid j \in RecPos(C_i)\} \\ \forall i \in \{1..n\}\ .\ \{y \in (P_i - Rec_i) \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} = \emptyset \\ \forall i \in \{1..n\}\ .\ (D_p \cap N_i, R_p \cup (R'_{p_i} - D_i), S_p \cap N_i) \vdash_{check} e_i \\ \textbf{where}\ R'_{p_i} = \{y \in (D_p \cup D) \cap sharerec(z, e_i) \mid z \in D_p \cap N_i\} - (D_p \cup N_i) \end{array}}{(D_p, R_p, S_p) \vdash_{check} \textbf{case!}\ x\ \textbf{of}\ \overline{C_i\ \overline{x_{ij}}^{n_i} \to e_i}^{n}}\ [\text{CASE!}_C]$$

**Fig. 7.** Top-down checking rules

remembering of the sets that were inferred in the bottom-up mode and that the algorithm recorded in the AST. So there is no need to infer them again.

The rules [$\text{VAR}_I$], [$\text{COPY}_I$] and [$\text{REUSE}_I$] assign to the corresponding variable a safe, don't-know and condemned mark respectively. If a variable occurs as a parameter of a data constructor then it gets a safe mark, as specified by the rule [$\text{CONS}_I$]. For the case of function application (rule [$\text{APP}_I$]) we obtain from the signature $\Sigma$ the positions of the parameters which are known to be condemned ($I_D$) and safe ($I_S$). The remaining ones belong to the set $I_N$ of unknown positions. The actual parameters in the function application get the corresponding

mark. The disjointness conditions in the rule [APP$_I$] prevent a variable from occurring at two condemned positions, or at a safe and a condemned position simultaneously. In rules [REUSE$_I$] and [APP$_I$] all variables belonging to the set $R$ are returned as in-danger, in order to preserve the invariant of the type system mentioned above.

In rule [LET$_I$], the results of the subexpressions $e_1$ and $e_2$ are checked by means of the assumption $(D_1 \cup R_1) \cap fv(e_2) = \emptyset$, corresponding to the $\rhd^L$ operator of the type system. Moreover, if a variable gets a condemned, in-danger or safe mark in $e_2$ then it can't be used destructively or become in danger in $e_1$, because of the operator $\rhd^L$. Hence this variable has to be propagated as safe through $e_1$ by means of a $\vdash_{check}$. According to the type system, the variables belonging to $R_2$ could also be propagated through $e_1$ with an unsafe mark. However, the inference algorithm resolves the non-determinism of the type system by assigning a maximal number of safe marks.

To infer the four sets for a **case**/**case!** expression (rules [CASE$_I$] and [CASE!$_I$]) we have to infer the result from each alternative. The function *RecPos* returns the recursive parameter positions of a given data constructor. The operator $\sqcup$ ensures the consistency of the marks inferred for a variable: if a variable gets two different marks in two distinct branches then at least one of them must be a don't-know mark. On the other hand, the inherited types of the pattern variables in each branch are checked via the *inh* and *inh*! predicates. A mark may be propagated top-down through the AST (by means of $\vdash_{check}$ rules) in one of the following cases:

1. A variable gets a don't-know mark in a branch $e_j$ and a different mark in a branch $e_k$. The mark obtained from $e_k$ must be propagated through $e_j$.
2. A pattern variable gets a don't-know mark in a branch $e_j$. Its inherited type must be propagated through $e_j$. That is what the sets $D'_i$, $R'_i$ and $S'_i$ of the rule [CASE$_I$] achieve.
3. A variable belongs to $R''_i$ or $R'''_i$ (see below).

There exists an invariant in the $\vdash_{check}$ rules (see below) which specifies the following: if a variable $x$ is propagated top-down with a condemned mark, those variables sharing a recursive substructure with $x$ either have been inferred previously as condemned (via the $\vdash_{inf}$ rules) or have been propagated with an unsafe (i.e. in-danger or condemned) mark as well. The sets $R''_i$ and $R'''_i$ occur in the [CASE$_I$] and [CASE!$_I$] rules in order to preserve this invariant. The set $R''_i$ contains the pattern variables which may share a recursive substructure with some condemned variable being propagated top-down through the $e_i$. The set $R'''_i$ contains those variables that do not belong to any of the $(D_i, R_i, S_i, N_i)$ sets corresponding to the $i$-th **case** branch, but they share a recursive descendant of a variable being propagated top-down through this branch as condemned. In [12] a few explanatory examples on these sets are given.

The $\vdash_{check}$ rules capture the same verifications as the $\vdash_{inf}$ rules, but in a top-down fashion. See [12] for more details about $R'_p$ in [LET$_C$].

The algorithm is modular in the sense that each function body is independently inferred. The result is reflected in the function type and this type is

available for typing the remaining functions. For typing a recursive function a fixpoint computation is needed. In the initial environment a don't-know mark is assigned to each formal argument. After each iteration, some don't-know marks may have turned into condemned, in-danger or safe marks. This procedure continues until the mark for each argument stabilises. If the fixpoint assigns an in-danger mark to an argument, this is rejected as a bad typing. Otherwise, if any don't-know mark remains, this is forced to be a safe mark by the algorithm and propagated down the whole function body by using the $\vdash_{check}$ rules once more. As a consequence, if the algorithm succeeds, every variable inferred as don't-know during the bottom-up traversal will eventually get a $d$, $r$ or $s$ mark (see [12] for a detailed proof).

If $n$ is the size of the AST for a function body and $m$ is the number of its formal arguments, the algorithm runs in $\Theta(mn^3)$ in the worst case. This corresponds to $m$ iterations of the fixpoint and a top-down traversal at each intermediate expression. However in most cases it is near to $\Theta(n^2)$, corresponding to a single bottom-up traversal and two fixpoint iterations.

### 4.1 Correctness of the Inference Algorithm

**Lemma 1.** *Let us assume that during the inference algorithm we have* $e \vdash_{inf} (D, R, S, N)$ *and* $(D', R', S') \vdash_{check} e$ *for an expression* $e$*. Then*

1. $D, R, S$ *and* $N$ *are pairwise disjoint.*
2. $D \cup S \cup N \subseteq FV(e)$, $R \subseteq scope(e)$ *and* $D \cup R \cup S \cup N \supseteq FV(e)$.
3. $\bigcup_{z \in D} sharerec(z, e) \subseteq D \cup R$.
4. $D', R'$ *and* $S'$ *are pairwise disjoint.*
5. $D' \cup S' \subseteq N$, $R' \subseteq scope(e)$.
6. $\bigcup_{z \in D'} sharerec(z, e) \subseteq D' \cup R' \cup D$.
7. $R' \cap S = \emptyset$, $R' \cap D = \emptyset$.

*Proof.* By induction on the corresponding $\vdash_{inf}$ and $\vdash_{check}$ derivations [12]. □

A single subexpression $e$ may suffer more than one $\vdash_{check}$ during the inference algorithm but always with different variables. This is due to the fact, not reflected in the rules, that whenever some variables in the set $N$ inferred for $e$ are forced to get a mark different from $n$, the decoration in the AST is changed to the new marks. More precisely, if $e \vdash_{inf} (D, R, S, N)$ and $(D', R', S') \vdash_{check} e$, then the decoration is changed to $(D \cup D', R \cup R', S \cup S', N - (D' \cup R' \cup S'))$. So, the next $\vdash_{check}$ for expression $e$ will get a smaller set $N - (D' \cup R' \cup S')$ of don't-know variables and, by Lemma 1, only those variables can be forced to change its mark. As a corollary, the mark for a variable can change during the algorithm from $n$ to $d, r$ or $s$, but no other transitions between marks are possible.

Let $(D', R', S') \vdash^*_{check} e$ denote the accumulation of all the $\vdash_{check}$ involving $e$ during the algorithm and let $D', R'$ and $S'$ represent the union of respectively all the marks $d, r$ and $s$ forced in these calls to $\vdash_{check}$. If $e \vdash_{inf} (D, R, S, N)$ represent the sets inferred during the bottom-up mode, then $D' \cup R' \cup S' \supseteq N$ must hold, since every variable eventually gets a mark $d$, $r$ or $s$.

The next theorem uses the convention $\Gamma(x) = s$ (respectively, $r$ or $d$) to indicate that $x$ has a safe type (respectively, an in danger or a condemned type) without worrying about which precise type it has.

**Theorem 1.** *Let us assume that the function declaration* $f\ \overline{x_i}^n\ @\ \overline{r_j}^l = e$ *has been successfully typed by the inference algorithm and let* $e'$ *be any subexpression of* $e$ *for which the algorithm has got* $e' \vdash_{inf} (D, R, S, N)$ *and* $(D', R', S') \vdash^*_{check} e'$. *Then there exists a safe type* $s'$ *and a well-formed type environment* $\Gamma$ *such that* $\Gamma \vdash e' : s'$, *and* $\forall x \in scope(e')$:

$$[x \in D \cup D' \leftrightarrow \Gamma(x) = d] \wedge [x \in S \cup S' \leftrightarrow \Gamma(x) = s] \wedge [x \in R \cup R' \leftrightarrow \Gamma(x) = r]$$

*Proof.* By structural induction on $e'$ [12]. □

## 5 Small Examples

In this section we show some examples. Firstly, we review the example of appending two lists (*Core-Safe* code of `concatD` in Sec. 2). We shall start with the recursive call to *concatD*. Initially all parameter positions are marked as don't-know and hence the actual arguments *xs* and *ys* belong to set $N$. The variables $x$ and $x_1$ get an $s$ mark since they are used to build a DS. In addition to this, $x_2$ is returned as the function's result, so it gets and $s$ mark. Joining the results of both auxiliary and main expressions in **let** we get the following sets: $D = \emptyset$, $R = \emptyset$, $S = \{x\}$, $N = \{xs, ys\}$. With respect to the **case**! branch guarded by [ ], the variable *ys* gets a safe mark (rule [VAR$_I$]). Information of both alternatives in **case**! is gathered as follows:

| | | | | | | |
|---|---|---|---|---|---|---|
| **([ ] guard)** | $D_1 = \emptyset$ | $R_1 = \emptyset$ | $S_1 = \{ys\}$ | $N_1 = \emptyset$ | $P_1 = \emptyset$ | $Rec_1 = \emptyset$ |
| **($x : xs$ guard)** | $D_2 = \emptyset$ | $R_2 = \emptyset$ | $S_2 = \{x\}$ | $N_2 = \{xs, ys\}$ | $P_2 = \{x, xs\}$ | $Rec_2 = \{xs\}$ |

Since *ys* has a safe mark in the branch guarded by [ ] and a don't-know mark in the branch guarded by $(x : xs)$, the safe mark has to be propagated through the latter by means of the $\vdash_{check}$ rules. Moreover, the pattern variable *xs* is also propagated as condemned. The first bottom-up traversal of the AST terminates with the following result: $D = \{zs\}$, $R = \emptyset$, $S = \{ys\}$ and $N = \emptyset$. Consequently the type signature of *concatD* is updated: the first position is now condemned and the second one is safe. Another bottom-up traversal is needed, as the fixpoint has not been reached yet. Now variables *xs* and *ys* belong to sets $D$ and $S$ respectively in the recursive call to *concatD*. Variable *zs* is also marked as in-danger, since it shares a recursive structure with *xs*. However, neither *xs* nor *zs* occur free in the main expression of **let** and hence the rule [LET$_I$] may still be applied. At the end of this iteration a fixpoint has been reached. The final type signature for *concatD* without regions is $\forall a.[a]! \rightarrow [a] \rightarrow [a]$.

The type of function *insertD*, defined in Sec. 2, is *Int* → *Tree Int!* → *Tree Int*. Other successfully typed destructive functions (whose code is not shown) are the following for splitting a list and for inserting an element in an ordered list:

$$splitD :: \forall a.Int \rightarrow [a]! \rightarrow ([a], [a]) \qquad insertLD :: \forall a.[a]! \rightarrow a \rightarrow [a]$$

## 6 Related Work

Our safety type system has some characteristics of linear types (see [18] as a basic reference). A number of variants of linear types have been developed for years for coping with the related problems of achieving safe updates in place in functional languages [13] or detecting program sites where values could be safely deallocated [9]. The work closest to *Safe*'s type system is [2], where the authors present a type system for a language which explicitly reuses heap cells. They prove that well-typed programs can be safely translated to an imperative language with an explicit deallocation/reusing mechanism. We summarise here the differences and similarities with our work.

In the first place, there are non-essential differences such as: (1) They only admit algorithms running in constant heap space, i.e. for each allocation there must exist a previous deallocation. (2) They use at the source level an explicit parameter $d$ representing a pointer to the cell being reused. (3) They distinguish two different cartesian products depending on whether there is sharing or not between the tuple components.

Also, there are the following obvious similarities: (1) They allow several accesses to the same variable, provided that only the last one is destructive. (2) They express the nature of arguments (destructive, read-only and shared, or just read-only) in the function type. (3) They need information about sharing between the variables and the final result of expressions.

But, in our view, the following more essential differences makes our language and type system more powerful than theirs:

1. Their uses 2 and 3 (read-only and shared, or just read-only) could be roughly assimilated to our use $s$ (read-only), and their use 1 (destructive), to our use $d$ (condemned). We add a third use $r$ (in-danger) arising from a sharing analysis based on abstract interpretation. This use allows us to know more precisely which variables are in danger when some other is destroyed.
2. Their uses form a total order $1 < 2 < 3$. A type assumption can always be worsened without destroying the well-typedness. Our marks $s, r, d$ do not form a total order. Only in some expressions (**case** and COPY) we allow the partial order $s \leq r$ and $s \leq d$. It is not clear whether that order gives more power to the system or not. In principle it will allow different uses of a variable in different branches of a conditional being the use of the whole conditional the worst one. For the moment our system does not allow this.
3. Their system forbids non-linear applications such as $f(x, x)$. We allow them for $s$-type arguments.
4. Our typing rules for **let** $x_1 = e_1$ **in** $e_2$ allow more combinations than theirs. Let $i \in \{1, 2, 3\}$ the use assigned to $x_1$, be $j$ the use of a variable $z$ in $e_1$ and be $k$ the use of the same variable $z$ in $e_2$. We allow the following combinations $(i, j, k)$ that they forbid: $(1, 2, 2)$, $(1, 2, 3)$ and $(2, 2, 2)$. The deep reason is our more precise sharing information and the new in-danger type. Examples of *Safe* programs using respectively the combinations $(1, 2, 3)$ and $(1, 2, 2)$ are the following, where $x$ and $z$ get an $s$-type in our type system:

$$\textbf{let } x = z : [\,] \textbf{ in case! } x \textbf{ of} \ldots \textbf{case } z \textbf{ of} \ldots$$
$$\textbf{let } x = z : [\,] \textbf{ in case! } x \textbf{ of} \ldots z$$

Both take profit from the fact that $z$ is not a recursive descendant of $x$.

Summarising our contribution, we have developed an inference algorithm for safe destruction which improves on previous attempts on this area, has a low cost, and can be applied to other functional languages smilar to *Safe* (i.e. eager and first-order). In particular, Hofmann and Jost's language [5] could benefit from the work described here.

## References


1. Aiken, A., Fähndrich, M., Levien, R.: Better Static Memory Management: Improving Region-based Analysis of Higher-order Languages. In: PLDI 1995, pp. 174–185. ACM, New York (1995)
2. Aspinall, D., Hofmann, M., Konečný, M.: A Type System with Usage Aspects. Journal of Functional Programming 18(2), 141–178 (2008)
3. Birkedal, L., Tofte, M., Vejlstrup, M.: From Region Inference to von Neumann Machines via Region Representation Inference. In: POPL 1996, pp. 171–183. ACM, New York (1996)
4. Henglein, F., Makholm, H., Niss, H.: A Direct Approach to Control-flow Sensitive Region-based Memory Management. In: PPDP 2001, pp. 175–186. ACM, New York (2001)
5. Hofmann, M., Jost, S.: Static Prediction of Heap Space Usage for First-order Functional Programs. In: POPL 2003, pp. 185–197. ACM, New York (2003)
6. Hudak, P.: A Semantic Model of Reference Counting and its Abstraction. In: Lisp and Functional Programming Conference, pp. 351–363. ACM Press, New York (1986)
7. Inoue, K., Seki, H., Yagi, H.: Analysis of Functional Programs to Detect Run-Time Garbage Cells. ACM TOPLAS 10(4), 555–578 (1988)
8. Jones, S.B., Le Metayer, D.: Compile Time Garbage Collection by Sharing Analysis. In: FPCA 1989, pp. 54–74. ACM Press, New York (1989)
9. Kobayashi, N.: Quasi-linear Types. In: POPL 1999, pp. 29–42. ACM Press, New York (1999)
10. Montenegro, M., Peña, R., Segura, C.: A Simple Region Inference Algorithm for a First-Order Functional Language. In: TFP 2008, pp. 194–208 (2008)
11. Montenegro, M., Peña, R., Segura, C.: A Type System for Safe Memory Management and its Proof of Correctness. In: PPDP 2008, pp. 152–162. ACM, New York (2008)
12. Montenegro, M., Peña, R., Segura, C.: An Inference Algorithm for Guaranteeing Safe Destruction (extended version). Technical report, SIC-8-08. UCM (2008), `http://federwin.sip.ucm.es/sic/investigacion/publicaciones/pdfs/SIC-8-08.pdf`
13. Odersky, M.: Observers for Linear Types. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 390–407. Springer, Heidelberg (1992)
14. Peña, R., Segura, C.: Formally Deriving a Compiler for SAFE. In: Horváth, Z., Zsók, V. (eds.) Preliminary proceedings of IFL 2006, pp. 429–426. Technical Report, 2006-S01. Eötvös Loránd University (2006)

15. Peña, R., Segura, C., Montenegro, M.: A Sharing Analysis for Safe. In: Trends in Functional Programming, vol. 7, pp. 109–128 (2007)
16. Tofte, M., Birkedal, L., Elsman, M., Hallenberg, N., Olesen, T.H., Sestoft, P.: Programming with regions in the MLKit (revised for version 4.3.0). Technical report, IT University of Copenhagen, Denmark (2006)
17. Tofte, M., Talpin, J.-P.: Region-based memory management. Information and Computation 132(2), 109–176 (1997)
18. Wadler, P.: Linear types can change the world! In: IFIP TC 2 Working Conference on Programming Concepts and Methods, pp. 561–581. North-Holland, Amsterdam (1990)

# From Monomorphic to Polymorphic Well-Typings and Beyond

Tom Schrijvers[1,⋆], Maurice Bruynooghe[1], and John P. Gallagher[2,⋆⋆]

[1] Dept. of Computer Science, K.U.Leuven, Belgium
[2] Dept. of Computer Science, Roskilde University, Denmark

**Abstract.** Type information has many applications; it can e.g. be used in optimized compilation, termination analysis and error detection. However, logic programs are typically untyped. A well-typed program has the property that it behaves identically on well-typed goals with or without type checking. Hence the automatic inference of a well-typing is worthwhile. Existing inferences are either cheap and inaccurate, or accurate and expensive. By giving up the requirement that all calls to a predicate have types that are instances of a unique polymorphic type but instead allowing multiple polymorphic typings for the same predicate, we obtain a novel strongly-connected-component-based analysis that provides a good compromise between accuracy and computational cost.

## 1 Introduction

While type information has many useful applications, e.g. in optimized compilation, termination analysis, documentation and debugging, most logic programming languages are untyped. In [4], Mycroft and O'Keefe propose a polymorphic type scheme for Prolog which makes static type checking possible and has the guarantee that well-typed programs behave identically with or without type checking, i.e., the types do not affect the execution (of well-typed goals). Lakshman and Reddy [3] provide a type reconstruction algorithm that, given the type definitions, infers missing predicate signatures.

Our aim is to construct a well-typing for a logic program automatically without prescribing any types or signatures. While there has been plenty of other work on such completely automatic type inference for logic programs (sometimes called "descriptive" typing), the goal was always to construct *success types*, that is, an approximation of the success set of a program represented by typed predicates. A well-typing by contrast represents in general neither an over- nor an under-approximation of the success set of the program. It was, to the best of our knowledge, not until [1] that a method was introduced to infer a well-typing for logic programs automatically, without given type definitions. The paper describes how to infer a so-called monomorphic well-typing which derives a type signature for every predicate. The well-typing has the property that the type

⋆ Post-doctoral researcher of the Fund for Scientific Research - Flanders.
⋆⋆ Work supported by the Danish Natural Science Research Council project *SAFT*.

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 152–167, 2009.

signature of each call is identical to that of the predicate signature. Below is a code fragment, followed by the results of the inference using the method of [1].

*Example 1.*
```
p(R) :- app([a],[b],M), app([M],[M],R).
app([],L,L).
    app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
    % type definition:
    :- type ablist ---> [] ; a ; b ; [ablist | ablist].
    % predicate signatures:
    :- app(ablist,ablist,ablist).
    :- p(ablist).
```

Note that the list type `ablist` is not the standard one. The reason is that `app/3` is called once with lists of `a`s and `b`s and once with lists whose elements are those lists. The well-typing constraint, stating that both calls must have the same signature as the predicate `app/3` enforces the above unnatural solution. Hence, the monomorphic type inference is not so interesting for large programs as they typically use many different type instances of commonly used predicates.

In a language with polymorphic types such as Mercury, [6], one typically declares `app/3` as having type `app(list(T),list(T),list(T))`. The first call instantiates the type parameter `T` with the type `elem` defined as `elem ---> a ; b` while the second call instantiates `T` with `list(elem)`.

The sets of terms denoted by these polymorphic types `list(elem)` and `list(list(elem))` are proper subsets of the monomorphic type `ablist`. For instance, the term `[a|b]` is of type `ablist`, but not of type `list(elem)`. Hence, polymorphic types allow for a more accurate characterization of program terms.

The work in [1] also sketches the inference of a polymorphic well-typing. However, the rules presented there are incomplete. We refer to [5] for a comprehensive set. In this paper, we revisit the problem of inferring a polymorphic typing. However, we impose the restriction that calls to a predicate that occur inside the strongly connected component (SCC) that defines the predicate (in what follows we refer to them as recursive calls, although there can also be mutual recursion) have the same type signature as the predicate. Other calls, appearing higher in the call graph of the program have a type signature that is a polymorphic instance of the definition's type. The motivation of the restriction is that it can be computationally very demanding when a recursive call is allowed to have a type that is a true instance of the definition's type. Indeed, [2] showed that type checking in a similar setting is undecidable, and [5] gave strong indications that this undecidability also holds for type inference in the above setting. Applied on Example 1, polymorphic type inference [5] gives:

*Example 2.*
```
:- type elem  ---> a ; b.
:- type list1(T) ---> [] ; [T | list1(T)].
:- type list2(T) ---> [] ; [T | list2(T)].
:- app(list1(T),list2(T),list2(T)).
:- p(list2(list2(elem))).
:- call app1(list1(elem), list2(elem), list2(elem)).
```

```
:- call app2(list1(list2(elem)),
     list2(list2(elem)), list2(list2(elem))).
```

Both `list1` and `list2` are renamings of the standard `list` type, hence this well-typing is equivalent to what one would declare in a language such as Mercury. As for the type signatures of the calls, `call app`$i$ refers to the $i$th call to `app/3` in `p/1`'s clause. In the first call, the polymorphic parameter is instantiated by `elem`, in the second by `list2(elem)`.

However, applying the analysis on a fragment where the second argument of the first call to `app/3` is not a list but a constant, one obtains a different result.

*Example 3.*

```
q(R) :- app([a],b,M), app([M],[M],R).
% type definition
:- type elem ---> a.                              % <<<
:- type list(T) ---> [] ; [T|list(T)] .
:- type blist(T) ---> [] ; [T|blist(T)] ; b.      % <<<
% signatures
:- app(list(T),blist(T),blist(T)).
:- q(blist(blist(elem))).
:- call app1(list(elem), blist(elem), blist(elem)).
:- call app2(list(blist(elem)),
             blist(blist(elem)), blist(blist(elem))).
```

Note that the "erroneous" call spoils the type of `app/3` by constructing a type that makes that call well-typed. Indeed, the type `blist(T)` has an extra base case with the functor `b`. Moreover, it is not clear from the type information which call is at the origin of the spoiled type. This drawback, together with the high computational cost of the polymorphic analysis motivated us to search for another solution. This led to a third approach where different non-recursive calls to the same predicate can be instances of different polymorphic well-typings of that predicate. Moreover, the inference can be done SCC by SCC, which lowers its computational cost. For the lowest SCC, defining `app/3`, we obtain:

*Example 4.*

```
:- type list(T)   ---> [] ; [T | list(T)].
:- type stream(T) ---> [T | stream(T)].
% signatures
:- app(list(T),stream(T),stream(T)).
```

Note the `stream(T)` type for the second and third argument. This is a well-typing. Nothing in the definition enforces the list structure to be terminated by an empty list, hence this case is absent in the type for second and third argument. Note that neither of the two types is an instance of the other one.

For the SCC defining `p/1` one obtains the following.

*Example 5.*

```
:- type elem ---> a ; b.
:- type elist1 ---> [elem|elist1] ; [].
:- type elist2 ---> [elem|elist2] ; [].
```

```
:- type elistlist1 ---> [elist2|elistlist1] ; [].
:- type elistlist2 ---> [elist2|elistlist2] ; [].
% signatures
:- p(elistlist2).
:- call app1(elist1, elist2, elist2).
:- call app2(elistlist1, elistlist2, elistlist2).
```

In this SCC, `app/3` is called with types that are instances of lists. These instances are represented as monomorphic types; with a small extra computational effort, one could separate them in the underlying polymorphic type and the parameter instances. Finally, for the SCC defining `q/1` one obtains the following type.

*Example 6.*

```
:- type elem ---> a.                                % <<<
:- type elist1 ---> [elem|elist1] ; [].
:- type eblist2 ---> [elem|eblist2] ; b.            % <<<
:- type elistlist1 ---> [eblist2|elistlist1] ; [].
:- type elistlist2 ---> [eblist2|elistlist2] ; [].
% signatures
:- q(elistlist2).
:- call app1(elist1, eblist2, eblist2).
:- call app2(elistlist1, elistlist2, elistlist2).
```

This reveals that `eblist2` is not a standard list and that it is the first call to `app/3` that employs this type.

This example shows that the SCC-based polymorphic analysis provides more useful information than the true polymorphic one, identifying which parts of a predicate's type originate in the predicate definition itself and which in the respective calls to that predicate. It is interesting also in giving up the usual concept underlying polymorphic typing that each predicate should have a unique principal typing and that all calls to that predicate should have a type that is an instance of it. The types of the first and second call to `app` are equivalent to instances of the type signatures `app(list1(T),blist2(T),blist2(T))` and `app(list1(T),list2(T),list2(T))` respectively, where the types `list1(T)` and `list2(T)` are the standard polymorphic list types but `blist2(T)` is defined as `blist2(T) ---> [T|eblist2(T)] ; b`.

Our contributions are summarized as follows.

- A new and efficient SCC-based polymorphic type analysis.
- A comparison with two other analyses, a cheap but inaccurate monomorphic analysis and an accurate but expensive polymorphic analysis.
- An evaluation showing the respective merits of the different analyses.

In Section 2, we introduce the necessary background on logic programs, types and well-typings and introduce a notion of minimal well-typing. In Section 3, we recall the previous work on monomorphic type inference. We do the same, very briefly, in Section 4 for the polymorphic type inference. In Section 5, we introduce our new SCC-based analysis. We experimentally compare the three approaches in Section 6 and conclude in Section 7.

## 2 Problem Statement and Background Knowledge

*Logic Programs* The syntax of logic programs considered here is as follows.

```
Program  := Clause*;
Clause   := Atom ':-' Goal;
Goal     := Atom | '(' Goal , Goal ')' | Term '=' Term | 'true' ;
Atom     := Pred '(' Term ',' ... ',' Term ')' ;
Term     := Var | Functor '(' Term ',' ... ',' Term ')' ;
```

`Pred`, `Functor` and `Var` refer to sets of predicate symbols, function symbols and variables respectively. Elements of `Pred` and `Functor` start with a lower case letter, whereas elements of `Var` start with an upper case letter.

*Types* For type definitions, we adopt the syntax of Mercury [6]. The set of *Type expressions (types)* $\mathcal{T}$ consists of terms constructed from an infinite set of type variables (parameters) $\mathcal{V}_\mathcal{T}$ and a finite alphabet of ranked type symbols $\Sigma_\mathcal{T}$; these are distinguishable —by context— from the set of variables $V$ and alphabet of functors $\Sigma$ used to construct terms. Variable-free types are called monomorphic; the others polymorphic. Type substitutions of the form $\{T_1/\tau_1, \ldots, T_n/\tau_n\}$, where $T_i$ and $\tau_i$ $(1 \leq i \leq n)$ are parameters and types respectively, define mappings from types to types by the simultaneous replacement of each parameter $T_i$ by the corresponding type $\tau_i$.

**Definition 1 (Type definition).** *A* type rule *for a type symbol $h/n \in \Sigma_\mathcal{T}$ is of the form*

$$h(\overline{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k); \quad (k \geq 1)$$

*where $\overline{T}$ is an $n$-tuple of distinct type variables, $f_1, \ldots, f_k$ are distinct function symbols from $\Sigma$, $\bar{\tau}_i$ $(1 \leq i \leq k)$ are tuples of corresponding arity from $\mathcal{T}$, and type variables in the right hand side, if any, are from $\overline{T}$. A* type definition *is a finite set of type rules where no two left hand sides contain the same type symbol, and there is a rule for each type symbol occurring in the type rules.*

If $f_i(\bar{\tau}_i)$ is one of the alternatives in the type rule for $h(\bar{T})$, then the mapping $\bar{\tau}_i \to h(\bar{T})$ can be considered the type signature of the function symbol $f_i$. As in Mercury [6], a function symbol can occur in several type rules, hence can have several type signatures; we say its type is *overloaded*.

The instance relation – $t(\bar{\tau})$ being an instance of $t(\bar{T})$ – is a partial order over the set of types $\mathcal{T}$. However, we want to define a more expressive partial order, denoted $\leq$ (and $<$ when strict). While $list(T_1) \leq list(list(T_2))$ in the instance relation, we also want, in Example 4, that $stream(T) < list(T)$ as the former has less alternatives than the latter. More subtly, with $stream(T, A) \longrightarrow [T|A]$, we want $stream(T, A) < stream(T)$. With $stream1(T) \longrightarrow [T|stream2(T)]$ and $stream2(T) \longrightarrow [T|stream1(T)]$, we want $stream1(T) < stream(T)$ and $stream2(T) < stream(T)$ as strict relationships. The following definition defines a preorder achieving this.

**Definition 2 ($\leq$ Type inclusion).** *A type $\tau$ is included in a type $\sigma$ ($\tau \leq \sigma$) if there exist (i) a function $\rho$ mapping non variable types used in $\tau$ to non variable types used in $\sigma$ such that $\tau\rho = \sigma$ and (ii) a type substitution $\theta$ such that for each type $t(\bar{\tau})$ in the domain of $\rho$ we can show that $t(\bar{\tau})$ is included in $t(\bar{\tau})\rho$.*

*To verify (ii), assuming $s$ is the type symbol in $t(\bar{\tau})\rho$ and the type rules for $t$ and $s$ are $t(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_m(\bar{\tau}_m)$ and $s(\bar{S}) \longrightarrow g_1(\bar{\sigma}_1); \ldots; g_n(\bar{\sigma}_n)$ respectively, we check that there exists a $\overline{\sigma}$ such that for each alternative $f_i(\overline{\tau_i})$ in the type rule for $t(\bar{T})$ there is an alternative $g_j(\overline{\sigma_j})$ such that $f_i = g_j$ and for each $k$, either $\tau_{i,k}\{\bar{T}/\bar{\tau}\}\rho\theta = \sigma_{j,k}\{\bar{S}/\bar{\sigma}\}$ or $\tau_{i,k}\{\bar{T}/\bar{\tau}\}\rho\theta \leq \sigma_{j,k}\{\bar{S}/\bar{\sigma}\}$.*

Type inclusion is a preorder; using it to define equivalence classes, one obtains a partial order (which for convenience we also denote $\leq$). For the "subtle" examples above, $stream(T, A) < stream(T)$ with $\rho$ and $\theta$ given by $\rho(stream(T, A)) = stream(T)$ and $\theta = \{A/stream(T)\}$. Similarly, $stream1(T) < stream(T)$, with $\rho(stream1(T)) = stream(T)$, $\rho(stream2(T)) = stream(T)$, and $\theta$ the empty substitution. As a final example, the type $elistlist1$ of Example 5 is equivalent to the instance $list(list(elem))$ of the polymorphic type $list(T)$. In one direction, $\rho(elistlist1) = list(list(elem))$, $\rho(elist2) = list(elem)$, and $\theta$ is empty. In the other direction, $\rho(list(list(elem))) = elistlist1$ and $\rho(list(elem)) = elist2$ while $\theta$ is also empty. Note that $list(list(T)) < elistlist1$ with $\rho(list(list(T))) = elistlist1$, $\rho(list(T)) = elist2$ and $\theta = \{T/elem\}$.

*Typing Judgements* A *predicate signature* is of the form $p(\bar{\tau})$ and declares a type $\tau_i$ for every argument of predicate $p$. A type environment $E$ for a program $\mathcal{P}$ is a set of typings $X : \tau$, one for every variable $X$ in $\mathcal{P}$, and predicate signatures $p(\bar{\tau})$, one for every predicate $p$ in $\mathcal{P}$, and a type definition. A typing judgement $E \vdash e : \tau$ asserts that $e$ has type $\tau$ for the *type environment* $E$ and $E \vdash e : \diamond$ asserts that $e$ is well-typed. A typing judgement is valid if it respects the typing rules of the type system. We will consider three different type systems, but they

$$\text{(Var)}\ \Gamma, X : \tau \vdash X : \tau$$

$$\text{(Term)}\ \frac{(\texttt{:- type}\ \tau \longrightarrow \ldots\ \texttt{;}\ f(\tau_1, \ldots, \tau_n)\ \texttt{;}\ \ldots) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i' \qquad \tau_i' = \tau_i\theta}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau\theta}$$

$$\text{(True)}\ \Gamma \vdash \texttt{true} : \diamond$$

$$\text{(Unif)}\ \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\texttt{=}t_2 : \diamond}$$

$$\text{(Conj)}\ \frac{\Gamma \vdash g_1 : \diamond \qquad \Gamma \vdash g_2 : \diamond}{\Gamma \vdash (g_1\texttt{,}g_2) : \diamond}$$

$$\text{(Clause)}\ \frac{p(\tau_1, \ldots, \tau_n) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i \qquad \Gamma \vdash g : \diamond}{\Gamma \vdash p(t_1, \ldots, t_n)\ \texttt{:-}\ g : \diamond}$$

$$\text{(Prog)}\ \frac{\Gamma \vdash a_i\ \texttt{:-}\ g_i : \diamond}{\Gamma \vdash \{\overline{a_i\ \texttt{:-}\ g_i}\} : \diamond}$$

**Fig. 1.** The Common Type Judgement Rules

$$\text{(MonoCall)}\ \frac{p(\tau_1,\ldots,\tau_n)\in\Gamma \qquad \Gamma\vdash t_i:\tau_i}{\Gamma\vdash p(t_1,\ldots,t_n):\diamond}$$

$$\text{(RecCall)}\ \frac{p(\tau_1,\ldots,\tau_n)\in\Gamma \qquad \Gamma\vdash t_i:\tau_i}{\Gamma\vdash p(t_1,\ldots,t_n):\diamond}$$

$$\text{(PolyCall)}\ \frac{p(\tau_1,\ldots,\tau_n)\in\Gamma \qquad \Gamma\vdash t_i:\tau_i' \qquad \tau_i'=\tau_i\theta}{\Gamma\vdash p(t_1,\ldots,t_n):\diamond}$$

$$\text{(SCCCall)}\ \frac{\Gamma\vdash t_i:\tau_i' \qquad (\texttt{:- type}\ \tau_i'\longrightarrow\ldots)\in\Gamma \qquad \Gamma'\cup\{\texttt{:- type}\ \tau_i'\longrightarrow\ldots\}\cup\{p(\tau_1',\ldots,\tau_n')\}\vdash \mathit{subprog}(p/n):\diamond}{\Gamma\vdash p(t_1,\ldots,t_n):\diamond}$$

**Fig. 2.** The Call Rules

differ only in one place, namely in the typing of predicate calls in rule bodies. Figure 1 shows the typing rules for all the other language constructs, common to all type systems. The Var rule states that a variable is typed as given in the type environment. The Term rule constructs the type of a compound term (the quantifier $\forall i$ is omitted); the other rules state the well-typing of atoms and that a program is well-typed when all its parts are.

The different ways of well-typing a call are given in Figure 2. For the monomorphic analysis, the well-typing of a call is identical to that of the predicate in the environment (MonoCall rule). For the other two analyses, this only holds for the recursive calls (RecCall rule). The polymorphic analysis requires that the type of a non-recursive call is an instance (under type substitution $\theta$) of the type of the predicate (PolyCall rule), while the SCC based analysis (SCCCall rule) requires that the well-typing of the call in $\Gamma$ —which has predicate signature $p(\tau_1',\ldots,\tau_n')$— is such that there exists a typing environment (that can be different from $\Gamma$) with the following properties: the subprogram defining the predicate ($\mathit{subprog}(p/n)$) is well-typed in $\Gamma'$ and the predicate signature of $p/n$ is $p(\tau_1',\ldots,\tau_n')$ itself. Note that this implies that there exists a polymorphic type signature for $p/n$ such that $p(\tau_1',\ldots,\tau_n')$ is equivalent to an instance of it; however, that polymorphic type can be different for different calls.

In all three analyses, we are interested in *least* solutions. To define formally the notion of least solution, we define a partial order on predicate signatures.

**Definition 3 ($\preceq$, preorder on predicate signatures).** *A predicate signature $p(\bar\tau)$ is smaller than a predicate signature $p(\bar\sigma)$ ($p(\bar\tau)\preceq p(\bar\sigma)$) iff there is a mapping $\rho$ from the types in $\bar\tau$ to larger types (for all $i$, $\tau_i\leq\tau_i\rho$) that preserves the type variables – it can possibly introduce extra type variables – such that $p(\bar\tau)\rho\theta=p(\bar\sigma)$ for some type substitution $\theta$.*

This preorder can be extended to a partial order over equivalence classes. For convenience, we denote the latter also as $\preceq$, and write $\prec$ when the order is strict. With $p(\bar\tau_T)$ denoting the predicate signature of predicate $p$ in a well-typing $T$, we can now also define a partial order over (equivalence classes) of well-typings:

**Definition 4 ($\preceq$ Partial order on well-typings).** *A well-typing $T_1$ of a program is smaller than well-typing $T_2$ of the same program ($T_1 \preceq T_2$) if, for each predicate $p/n$, $p(\bar{\tau}_{T_1}) \preceq p(\bar{\tau}_{T_2})$.*

We can now say that a well-typing of a program is a *least well-typing* if it is smaller than any other well-typing of the same program.

For example, consider the polymorphic types of `app/3` in Examples 2 and 4. We have $app(list(T), stream(T), stream(T)) \prec app(list1(T), list2(T), list2(T))$ using the mapping $\rho(list(T)) = list1(T)$ and $\rho(stream(T)) = list2(T)$ and $\theta$ the identity mapping to satisfy Definition 3. Also, $app(list1(T), list2(T), list2(T)) \prec app(list(T), list(T), list(T))$, with $\rho(list1(T)) = list(T)$, $\rho(list2(T)) = list(T))$.

Note that $app(list(T), list(T), list(T)) \preceq app(list1(T), list2(T), list2(T))$ is not the case. The latter well-typing is "better" than the former because it expresses that there can be no aliasing between the backbones of the lists in the first argument and those in the second argument. Note that `app/3` has many other well-typings such as $app(list(list(T)), stream(list(T)), stream(list(T)))$ and $app(list(list(T)), list(list(T)), list(list(T)))$. All of them are larger than the well-typing $app(list(T), stream(T), stream(T))$. For the former, $\rho$ is the identity mapping and $\theta = \{T/list(T)\}$; for the latter, $\rho(list(T)) = list(T)$ and $\rho(stream(T)) = list(T)$ and $\theta = \{T/list(T)\}$.

Each type judgement gives rise to a least set of constraints from which a well-typing can be derived. Adding more constraints results in a larger well-typing.

**Theorem 1.** *There exists a* least *well-typing under each of the three type judgements introduced above.*

Observe that any well-typing satisfying the MonoCall rule also satisfies the PolyCall rule; furthermore any well-typing satisfying the latter also satisfies the SCCCall rule, hence also the following holds.

**Theorem 2.** *Let $T_{Mono}$, $T_{Poly}$ and $T_{SCC}$ be the least well-typings of a program $P$ obtained with respectively the monomorphic, the polymorphic and the SCC-based analysis. Then $T_{SCC} \preceq T_{Poly} \preceq T_{Mono}$.*

*Example 7.* : The following program's three well-typings are all distinct:

```
r(f(X)).                p(Y) :- r(Y).          q(Z) :- r(Z).
                        p(a).                  q(f(b)).

% Monomorphic analysis              | % Polymorphic analysis
:- type t1 ---> a ; f(t2).          | :- type t3(T) ---> a ; f(T).
:- type t2 ---> b.                  | :- type t4    ---> b.
:- pred r(t1).                      | :- pred r(t3(T)).
:- pred p(t1).                      | :- pred p(t3(T)).
:- pred q(t1).                      | :- pred q(t3(t4)).
-------------------------------------------------------------------
% SCC-based analysis
```

```
:- type t5(T) ---> f(T).                :- pred r(t5(T)).
:- type t6(T) ---> a ; f(T).            :- pred p(t6(T)).
:- type t7    ---> f(t8).               :- pred q(t7).
:- type t8    ---> b.
```

$$\begin{array}{ccccc} r(t5(T)) & \prec & r(t3(T)) & \prec & r(t1) \\ p(t6(T)) & = & p(t3(T)) & \prec & p(t1) \\ q(t7) & \prec & q(t3(t4)) & = & q(t1) \\ \hline T_{SCC} & \prec & T_{Poly} & \prec & T_{Mono} \end{array}$$

## 3 The Monomorphic Type Analysis

The monomorphic type system is simple. It requires that all calls to a predicate have exactly the same typing as the signature (rule MonoCall in Figure 2).

The monomorphic type inference (first described in [1]) consists of three phases: (1) Derive *type constraints* from the program text. (2) Normalize (or solve) the type constraints. (3) Extract *type definitions* and *type signatures* from the normalized constraints. A practical implementation may interleave these phases. In particular, (1) may be interleaved with (2) via incremental constraint solving. We discuss the three phases in more detail below.

*Phase 1: Type Constraint Derivation* For the purpose of constraint derivation we assume that a distinct type $\tau$ is associated with every occurrence of a term. In addition, every defined predicate $p$ has an associated type signature $p(\overline{\tau})$ and every variable $X$ an associated type $\tau$; these are respectively denoted as $pred(p(\overline{\tau}))$ and $var(X) : \tau$. The associated type information serves as the initial assumption for the type environment $E$; initially all types are unconstrained.

Now we impose constraints on these types based on the program text. For the monomorphic system, we only need two different kinds of type constraint: $\tau_1 = \tau_2$: the two types are (syntactically) equal, and $\tau \supseteq f(\overline{\tau})$: the type definition of type $\tau$ contains a case $f(\overline{\tau})$.

Figure 3 shows what constraints are derived from various language constructs. The unlisted language constructs do not impose any constraints.

*Phase 2: Type Constraint Normalization* In the constraint normalization phase we rewrite the bag of derived constraints (the constraint store) to propagate all available information. The normalized form is obtained as the fixed point of just three rewrite steps. The first rewrite step drops trivial equalities.

$$(\textbf{Triv})\ C \cup \{\tau = \tau\} \Longrightarrow C$$

The second rewrite step unifies equal types.

$$(\textbf{Unif})\ C \cup \{\tau_1 = \tau_2\} \Longrightarrow C[\tau_2/\tau_1] \cup \{\tau_1 = \tau_2\}$$

where $\tau_1 \in C$ and $\tau_1 \not\equiv \tau_2$. The third step collapses equal function symbols.

$$(\textbf{Coll})\ C \cup \{\tau \supseteq f(\overline{\tau_1}), \tau \supseteq f(\overline{\tau_2})\} \Longrightarrow C \cup \{\tau \supseteq f(\overline{\tau_1}), \overline{\tau_1} = \overline{\tau_2}\}$$

$$(\textsc{Var})\ \frac{X : \tau' \qquad var(X) : \tau}{\tau = \tau'}$$

$$(\textsc{Term})\ \frac{t_1 : \tau_1 \qquad \ldots \qquad t_n : \tau_n \qquad f(t_1, \ldots, t_n) : \tau}{\tau \supseteq f(\tau_1, \ldots, \tau_n)}$$

$$(\textsc{Call})\ \frac{t_i : \tau_i' \qquad pred(p(\tau_1, \ldots, \tau_n)) \qquad (a \ \texttt{:-}\ g) \in P \qquad p(t_1, \ldots, t_n) \in g}{\tau_i' = \tau_i}$$

$$(\textsc{Unif})\ \frac{t_1 : \tau_1 \qquad t_2 : \tau_2 \qquad t_1 \texttt{=} t_2 \in P}{\tau_1 = \tau_2}$$

$$(\textsc{Head})\ \frac{t_1 : \tau_1' \qquad \ldots \qquad t_n : \tau_n' \qquad pred(p(\tau_1, \ldots, \tau_n)) \qquad p(t_1, \ldots, t_n) \texttt{:-} g \in P}{\tau_i' = \tau_i}$$

**Fig. 3.** Constraint derivation rules

*Phase 3: Type Information Extraction* Type rules and type expressions are derived simultaneously from the normal form of the constraint store:

- A type $\alpha$ that does not appear as the first argument in a $\supseteq$ constraint gets as its type expression a unique type variable $A$.
- A type $\tau$ that appears on the lhs of one or more $\supseteq$ constraint is assigned a type expression $t(\ldots)$ with $t$ a unique type name and gives rise to a type rule of the form $t(\ldots) \longrightarrow \ldots$. The type expression $t(\ldots)$ has as its arguments the type variables $A_i$ occurring in the right hand side of the rule. For each constraint $\tau \supseteq f(\tau_1, \ldots, \tau_k)$, the right hand side has a case $f(exp_1, \ldots exp_k)$ where each $exp_i$ is the type expression assigned to $\tau_i$.
- A type $\tau_2$ occurring in the right hand side of an equation $\tau_1 = \tau_2$ is assigned the type expression assigned to $\tau_1$.

**Theorem 3.** *The monomorphic type inference terminates and infers a least well-typing.*

Indeed, the constraints created in Phase 1 are the minimal set of constraints to be satisfied by the well-typing; the first two rewriting steps in Phase 2 preserve the constraints, while the last one follows from the requirement that the right hand side of a type rule has distinct functor symbols and Phase 3 extracts the minimal types satisfying all constraints. Moreover, each phase terminates, hence the algorithm terminates and produces a minimal well-typing for the program.

Of particular interest is the time complexity of normalization:

**Theorem 4 (Time Complexity).** *The normalization algorithm has a near-linear time complexity $\mathcal{O}(n \cdot \alpha(n))$, where $n$ is the program size and $\alpha$ is the inverse Ackermann function.*

The $\alpha(n)$ factor follows from the **Unif** step, if implemented with the optimal union-find algorithm.

## 4 The Polymorphic Type Analysis

The polymorphic type system relaxes the monomorphic one. The type signature of non-recursive predicate calls is an instance of the predicate signature, rather than being identical to it. In [5] a type inference is described that allows this also for recursive calls. It is straightforward to adapt that approach to the current setting where recursive calls have the same signature as the predicate (only the constraint derivation phase needs to be altered). It has a complex set of rules because there is propagation of type information between predicate signature and call signature in both directions. A new case in a type rule from a type in the signature of a call is propagated to the corresponding type in the signature of the definition and in turn propagated to the corresponding types in the signature of all other calls. There, it can potentially interact with other constraints, leading to yet another case and triggering a new round of propagation. When the signature of recursive calls is allowed to be an instance of the predicate signature, it is unclear whether termination is guaranteed. Indeed, Henglein [2] showed that type checking in a similar setting is undecidable while we have strong indications [5] that type inferencing in the setting of that paper is also undecidable. The restriction on the signatures of recursive types guarantees termination but complexity remains high. Experiments indicate a cubic time complexity.

## 5 The SCC Type Analysis

The SCC type analysis reconstructs type information according to the SCCCall rule. We first present a simple, understandable, but rather naive approach for the analysis in Section 5.1, and subsequently refine it in Sections 5.2 and 5.3.

### 5.1 Naive Approach

A very simple way to implement an analysis of a program $P$ for the SCCCall rule is to apply the monomorphic analysis of Section 3 to a *transitive multi-variant* specialization (TMVS) of $P$.

**Definition 5 (Transitive Multi-Variant Specialization).** *We say that $P'$ is a transitive multi-variant specialization of $P$ if:*

- *There is at most one call to each predicate $p/n \in P'$ from outside the predicate's strongly connected component.*
- *There is an inductive equality relation between predicates of $P$ and $P'$ that implies structural equality.*

Effectively, we obtain a TMVS of $P$ by creating named apart copies of each predicate definition, one for each call to the predicate.

*Example 8.* For Example 1, we obtain:

```
p(R) :- app1([a],[b],M), app2([M],[M],R).

app1([],L,L).   app1([X|Xs],Ys,[X|Zs]) :- app2(Xs,Ys,Zs).
app2([],L,L).   app2([X|Xs],Ys,[X|Zs]) :- app2(Xs,Ys,Zs).
app([],L,L).    app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

Applying the monomorphic analysis, we obtain the type information as shown in Example 4 (for the `app/3` definition) and in Example 5 (for predicate `p/1`).

While this approach is simple, it is also highly inefficient. Observe that the size of a TMVS is worst-case exponential in $n$, the size of the original program. Hence, this approach has worst-case $\mathcal{O}(e^n \cdot \alpha(e^n))$ time complexity.

*Example 9.* The following program's TMVS exhibits the exponential blow-up:

$p_0$`(L) :- L = [a].`
$p_1$`(L) :- L = [A],` $p_0$`(A),` $p_0$`(L).`
...
$p_n$`(L) :- L = [A],` $p_{n-1}$`(A),` $p_{n-1}$`(L).`

In the TMVS of this program, there are two copies of $p_{n-1}/1$, four copies of $p_{n-2}$, ... and $2^n$ copies of $p_0/1$.

Below, we will make the analysis more efficient, but the worst case complexity remains exponential. However, in practice (see Section 6) the analysis turns out to be much better behaved than the polymorphic analysis.

### 5.2 Derive and Normalize Once

There is a clear inefficiency in the naive approach: the same work is being done repeatedly. Multiple copies of the same predicate definition are created, the same set of constraints is derived from each copy, and each copy of those constraints is normalized. This consideration leads us to propose a refined approach:

1. We analyze each predicate definition only once with Phases 1 and 2 of the monomorphic analysis.
2. We reuse the results for a whole *subprogram* multiple times.

In order to make good on the above two promises, we must traverse the program in a specific manner: SCC-by-SCC in a bottom-up fashion. In other words, we first visit the predicates that do not depend on any other predicates. Then we visit the predicates that only depend on the previous ones, and so on.

*SCC Traversal.* The strongly connected components (SCCs) of a program are sets of predicates, where each component is either a singleton containing a non-recursive predicate or a maximal set of mutually recursive predicates. There is a partial order on the SCCs; for components $s_1$ and $s_2$, $s_1 \preceq s_2$ iff some predicate in

$s_1$ depends (possibly indirectly) on a predicate in $s_2$. Hence, the refined analysis first computes the SCCs and topologically sorts them in ascending ordering wrt $\preceq$, yielding say $s_0, \ldots, s_m$. The SCCs are processed in that order. The SCCs of a program can be computed in $O(|P|)$ time [7].

*Example 10.* This means for Example 1 that we first visit `app/3` and then `p/1`: $s_0 = \{\texttt{app/3}\}$ and $s_1 = \{\texttt{p/1}\}$ . For the program of Example 9 with exponential-size TMVS, the predicate definitions of $p_i/1$ are visited in the order of increasing values of $i$: $s_i = \{p_i/1\}$.

*SCC Processing.* For each SCC $s_i$ we generate a set of constraints and, after normalisation, we derive a signature for each predicate $p/n$ in $s_i$. The set of type constraints for the clauses of $s_i$ are generated using the same rules as Phase 1 of the monomorphic analysis (see Figure 3). There is one exception: the *Call* rule is modified for non-recursive calls to NonRecCall.

Let $p(t_1, \ldots, t_n)$ be such a call to a predicate in SCC $s_j$, with $s_j \prec s_i$. Assume $\tau_1, \ldots, \tau_n$ are the argument types for the predicate signature in $s_j$ and $\tau'_1, \ldots, \tau'_n$ the types of the arguments $t_1, \ldots, t_n$ in the call. Instead of generating the equalities $\tau'_i = \tau_i$ as in Figure 3, one uses a renaming $\rho$ to give new names $\sigma_1, \ldots, \sigma_n$ to the types in $\tau_1, \ldots, \tau_n$ as well as new names to all types in the whole set $C_j$ of constraints in $s_j$. The generated constraint then consists of the renamed set of constraints $C_j\rho$ together with the equalities $\sigma_i = \tau'_i$ as described in the rule of Fig. 4.

$$\text{(NonRecCall)}\ \frac{\begin{array}{c} p/n \in s_j \qquad pred(p(\tau_1, \ldots, \tau_n)) \\ \tau_i\rho = \sigma_i \qquad t_i : \tau'_i \\ (a \texttt{ :- } g) \in P \qquad p(t_1, \ldots, t_n) \in g \end{array}}{C_j\rho \wedge \bigwedge_{i=1}^{n} \sigma_i = \tau'_i}$$

**Fig. 4.** Constraint derivation rule for non-recursive calls

Observe that, because of the copying of constraints, each call to a predicate in a lower SCC has its own type which does not interfere with calls to the same predicate elsewhere (interference is unavoidable in the polymorphic analysis).

Second, we normalize the derived constraints with the same rules as in Phase 2 of the monomorphic analysis. We call the resulting normalized constraint set $C_i$, and remember them for use in the subsequent SCCs.

Finally, we use the algorithm of Phase 3 of the monomorphic analysis to generate signatures for the predicates in $s_i$, based on the constraints $C_i$.

*Example 11.* Assume that for the program of Example 9 the constraint generation for $p_0/1$ yields for the signature $pred(p_0(\tau))$

$$C_0 \equiv \tau \supseteq \texttt{[}\tau_1 \texttt{|} \tau_2\texttt{]} \wedge \tau_1 \supseteq \texttt{a} \wedge \tau_2 \supseteq \texttt{[]}$$

Then for the analysis of $p_1$, we make two copies of $C_0$, one for the call $\mathtt{p_0(A)}$ and one for the call $\mathtt{p_0(L)}$. For instance, for the former call, we derive the constraints

$$\sigma \supseteq [\sigma_1 \,|\, \sigma_2] \wedge \sigma_1 \supseteq \mathtt{a} \wedge \sigma_2 \supseteq [] \wedge \tau' = \sigma$$

where the type of A is $\tau'$ and the substitution $\rho = \{\sigma/\tau, \sigma_1/\tau_1, \sigma_2/\tau_2\}$.

*Efficiency Improvements* Although the actual worst-cased complexity is not improved, the actual efficiency is. Not only does this SCC-based approach avoid actually generating the TMVS of a program, the normalized set of constraints $C_i$ is usually smaller than the originally generated set. Hence, the repeated copying of the normalized form is usually cheaper than repeated normalization.

We expect that an approach that shares rather than copies the constraints $C_j$ of the lower SCC $s_j$ would be even more efficient. It looks plausible to keep the set of constraints linear in the size of the program. However, the final extraction phase could in the worst case still give an exponential blow up (when each call has a signature that is an instance of a distinct polynomial predicate signature). We intend to explore this further in future work.

### 5.3 Projection

One can observe that there is no need to copy the whole constraint system $C_j$ of the SCC $s_j$ when processing a call to that SCC. For processing a non-recursive call to p/n, it suffices to copy a projection $C_j^{p/n}$: the part of $C_j$ that defines the types in the signature of $p/n$. This will usually be smaller than $C_j$.

*Example 12.* Consider the program with SCCs $s_1 = \{\mathtt{r/2}\}$, $s_2 = \{\mathtt{q/2}\}$ and $s_3 = \{\mathtt{p/2}\}$:

```
p(X) :- q(X).
q(Y) :- r(Y,Z).
r(a,b).
```

We have $C_2 \equiv (\tau_Y \supseteq \mathtt{a} \wedge \tau_Z \supseteq b)$ and signature $pred(\mathtt{q}(\tau_Y))$. However, if we project on the signature, then the second constraint of $C_2$ is dropped: $C_2^{\mathtt{q/1}} \equiv \tau_Y \supseteq \mathtt{a}$. In other words, for the call in $s_3$, we must copy half as many constraints with $C_2^{\mathtt{q/1}}$ than with $C_2$.

## 6 Evaluation

We evaluated the three algorithms on a suite of 45 small programs also used in [1]. Figure 5 displays the relative runtime (in % on a logarithmic scale) of both the polymorphic and SCC-based analyses with respect to the monomorphic analysis.

The monomorphic analysis finishes quickly, in less than 1 ms on a Pentium 4 2.00 GHz. The SCC analysis provides more accurate analysis in 1 to 3 times the

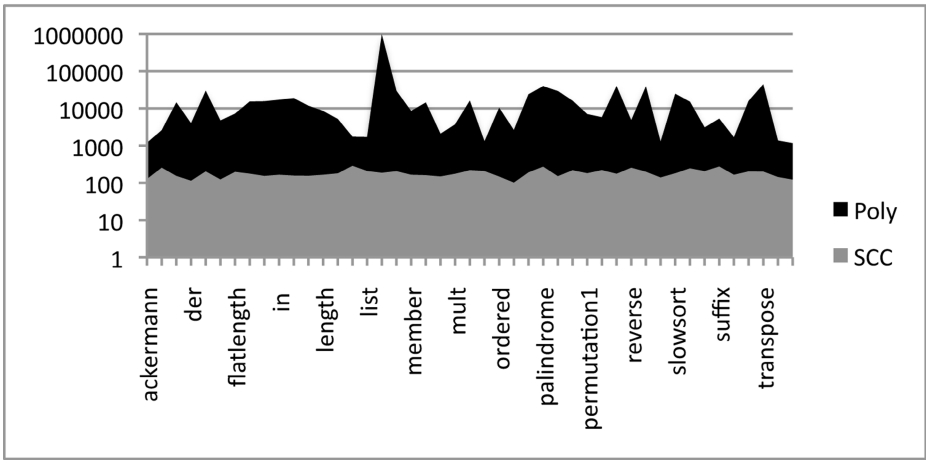



**Fig. 5.** Relative runtime (in %) wrt the monomorphic analysis

time of the monomorphic analysis. The complex polymorphic analysis lags far behind; it is easily 10 to 100 times slower.

A contrived scalable benchmark based on Example 1 shows that the monomorphic and SCC analyses can scale linearly, while the polymorphic analysis exhibits a cubic behavior. The scalable program (parameter $n$) is constructed as follows:

```
app([],L,L).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

r(R) :- app([a],[b],M1),
        app([M1],[M1],M2), ..., app([Mn],[Mn],R).
```

Below are the runtimes for the three inferences.[1]

| Program | MONO | SCC | POLY |
|---|---|---|---|
| app-1 | 0.38 ms | 0.65 ms | 12 ms |
| app-10 | 1.20 ms | 2.14 ms | 206 ms |
| app-100 | 9.60 ms | 18.10 ms | 88,043 ms |
| app-1000 | 115.80 ms | 238.05 ms | T/O |
| app-10000 | 1,402.40 ms | 2,955.95 ms | T/O |

## 7 Conclusion and Future Work

Within the framework of polymorphic well-typings of programs, it is customary to have a unique principal type signature for predicate definitions and type signatures of calls (from outside the SCC defining the predicate) that are instances of the principal type. We have presented a novel SCC-based type analysis that

[1] T/O means time-out after 2 minutes.

gives up the concept of a unique principal type and instead allows different calls to have type signatures that are instances of different well-typings of the predicate definition. This offers two advantages. Firstly, it is much more efficient than a true polymorphic analysis and is only slightly more expensive than a monomorphic one. In practice, it scales linearly with program size. Secondly, when an unexpected case appears in a type rule (which may hint at a program error), it is easy to figure out whether it is due to the predicate definition or to a particular call. This information cannot be reconstructed from the inferred types in the polymorphic and the monomorphic analyses.

In future work we plan to investigate the quality of the new analysis by performing type inference on Mercury programs where all type information has been removed and comparing the inferred types with the original ones.

## References

1. Bruynooghe, M., Gallagher, J.P., Van Humbeeck, W.: Inference of well-typing for logic programs with application to termination analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 35–51. Springer, Heidelberg (2005)
2. Henglein, F.: Type inference with polymorphic recursion. ACM Trans. Program. Lang. Syst. 15(2), 253–289 (1993)
3. Lakshman, T.L., Reddy, U.S.: Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In: Logic Programming, ISLP 1991, pp. 202–217 (1991)
4. Mycroft, A., O'Keefe, R.A.: A polymorphic type system for Prolog. Artificial Intelligence 23(3), 295–307 (1984)
5. Schrijvers, T., Bruynooghe, M.: Polymorphic algebraic data type reconstruction. In: Bossi, A., Maher, M.J. (eds.) Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, Venice, Italy, July 10-12, pp. 85–96 (2006)
6. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. Journal of Logic Programming 29(1-3), 17–64 (1996)
7. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM Journal of Computing 1(2), 146–160 (1972)

# On Negative Unfolding in the Answer Set Semantics

Hirohisa Seki

Dept. of Computer Science, Nagoya Inst. of Technology,
Showa-ku, Nagoya, 466-8555 Japan
seki@nitech.ac.jp

**Abstract.** We study negative unfolding for logic programs from a viewpoint of preservation of the answer set semantics. To this end, we consider negative unfolding in terms of nested expressions by Lifschitz et al., and regard it as a combination of the replacement of a literal by its definition (called "pre-negative unfolding") and double negation elimination. We give sufficient conditions for preserving the answer set semantics. We then consider a framework for unfold/fold transformation of locally stratified programs, which, besides negative unfolding, contains replacement rules, allowing a more general form than those proposed by Pettorossi and Proietti. A new folding condition for the correctness proof is identified, which is not required either for definite or stratified programs, but becomes necessary only in case of locally stratified programs. An extension of the framework beyond the class of locally stratified programs is also discussed.

## 1 Introduction

Since the seminal paper by Tamaki and Sato [10], various equivalence-preserving transformation rules in different semantics have been proposed (see an excellent survey [8] and references therein). Among them, *negative unfolding* is a transformation rule, which applies unfolding to a negative literal in the body of a clause. When used together with usual (positive) unfold/fold rules and replacement rules, negative unfolding is shown to play an important role in program transformation, construction (e.g., [4], [3]) and verification (e.g., [9]). One of the motivations of this paper is to better understand the properties of negative unfolding, especially from a viewpoint of preservation of the semantics of programs.

The framework for program synthesis by Kanamori-Horiuchi [4] is one of the earliest works in which negative unfolding is introduced. As a special case, their framework contains a class of stratified programs with two strata. Pettorossi and Proietti have proposed transformation rules for locally stratified logic programs [9], including negative unfolding (PP-negative unfolding for short). Unlike positive unfolding, PP-negative unfolding does not preserve the semantics of a given program in general, when applied to non-locally stratified programs. The following example is given in [3].

M. Hanus (Ed.): LOPSTR 2008, LNCS 5438, pp. 168–184, 2009.

*Example 1.* [3]

$$P_0 : \begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \end{array} \qquad P_1 : \begin{array}{l} p \leftarrow p \\ q \leftarrow not\ p \end{array}$$

Program $P_1$ is obtained by unfolding the first clause of $P_0$ by (i) first replacing $q$ by the body *not* $p$ of the clause defining $q$ (we call this operation "*pre-negative unfolding*" here), and then (ii) replacing *not not* $p$ by $p$ (*double negation elimination*). Program $P_0$ has two stable models (answer sets) $\{p\}$ and $\{q\}$, while program $P_1$ has the unique stable model (perfect model) $\{q\}$. Note that $P_0$ is not a (locally) stratified program. □

In this paper, we consider negative unfolding as a combination of (i) *pre-negative unfolding*, i.e., the replacement of an atom by its definition, and (ii) double negation elimination (DNE for short). It is shown that the properties of negative unfolding are well understood in terms of *nested expressions* by Lifschitz et al. [6]. We study some sufficient conditions of pre-negative unfolding and DNE for the preservation of the answer set semantics, and show that the class of locally stratified programs satisfies both of the conditions.

In program transformation, construction and verification, positive/negative unfolding are often utilized with the other transformation rules such as *folding* and *replacement*. We thus consider a framework for unfold/fold transformation of *locally stratified* programs. Pettorossi and Proietti [9] have already proposed program transformation rules for locally stratified programs, where their rules are based on the framework by Tamaki-Sato [10] for definite programs. Our framework given in this paper is based on the generalized one by Tamaki-Sato [11], and we show that their generalized framework can be also extended to locally stratified programs *almost* in the same way, except an *extra* condition on folding in order to preserve the answer set semantics (perfect models). Therefore, our contribution here is to show that the folding condition (FC1) in Assumption 3 in Sect. 4, which is not necessary for either definite or stratified programs, is required for *locally* stratified programs. Besides, the replacement rules considered here are more general than those by Pettorossi and Proietti [9], in the sense that non-primitive (or non-basic) predicates are allowed in the replacement rules. The set of primitive clauses can be a non-locally stratified program, as far as it satisfies the assumptions of an initial program defined below (Assumption 1). We also discuss such an extension of the framework beyond the class of locally stratified programs.

The organization of this paper is as follows. After summarizing preliminaries in Section 2, Section 3 gives negative unfolding in nested expressions, and shows some sufficient conditions for the preservation of the answer set semantics. In Section 4, a framework for unfold/fold transformation of locally stratified programs is described and an extension of the framework to a more general class than locally stratified programs is also discussed. Finally, a summary of this work is given in Section 5.

Throughout this paper, we assume that the reader is familiar with the basic concepts of logic programming, which are found in [7,1]. Some familiarity with the answer set semantics would be helpful for understanding Section 3.

## 2 Preliminaries: Nested Expressions

Nested expressions are proposed by Lifschitz et al. [6], where both the heads and bodies of the rules in a propositional program are formed from literals[1] using the operators , (conjunction), ; (disjunction), and *not* (negation as failure) that can be nested arbitrarily. For our purpose, however, it is sufficient to consider the case where the head of a rule consists of a single atom (positive literal), i.e., we do not consider either disjunction or negation as failure in the head.

The terminology below is largely borrowed from [6,12], restricted for our purpose. In this section, the words *atom* and *literal* are understood as in propositional logic. For a program with variables, the answer set semantics is given by *grounding*, that is, we consider the set of all rules obtained by all possible substitutions of elements of the Herbrand universe of the program for the variables occurring in the rules.

*Elementary formulas* are literals and the 0-place connectives $\bot$ ("false") and $\top$ ("true"). *Formulas* are built from elementary formulas using unary connectives *not* and the binary connectives , (conjunction) and ; (disjunction). A *rule* is an expression of the form: $A \leftarrow F$, where $A$ is a positive literal and $F$ is a formula, called the head and the body of the rule, respectively. A rule of the form $A \leftarrow \top$ is written as $A \leftarrow$ and is identified with formula $A$. A *program* is a set of rules.

Let $X$ be a consistent set[2] of literals. We define recursively when $X$ *satisfies* a formula $F$ (written as $X \models F$), as follows.

- For elementary $F$, $X \models F$ iff $F \in X$ or $F = \top$.
- $X \models (F, G)$ iff $X \models F$ and $X \models G$.
- $X \models (F; G)$ iff $X \models F$ or $X \models G$.
- $X \models not\ F$ iff $X \not\models F$

Furthermore, $X$ *satisfies* a rule $A \leftarrow F$ if $X \models F$ implies $X \models A$, and $X$ *satisfies* a nested program $P$ if it satisfies every rule in $P$.

The *reduct* of a formula $F$ relative to $X$ (written $F^X$) is obtained by replacing every maximal occurrence [3] in $F$ of a formula of the form *not* $G$ with $\bot$ if $X \models G$ and $\top$ otherwise. The *reduct* of a nested program $P$ relative to $X$ (written $P^X$) is obtained by replacing the body of each rule in $P$ by their reducts relative to $X$. Finally, $X$ is an *answer set* for a nested program $P$ if it is minimal among the sets of literals that satisfy $P^X$.

For example, program $\{p \leftarrow not\ not\ p\}$ has two answer sets $\phi$ and $\{p\}$, while program $\{p \leftarrow p\}$ has only one answer set $\phi$. Therefore, two negation as failure operators in a nested expression are not cancelled in general.

Lifschitz et al. have also considered several equivalent transformations in [6], where the equivalence of formulas are defined in a *strong* sense, that is, a formula

[1] Nested expressions by Lifschitz et al. [6] allow atoms possibly preceded by the classical negation sign $\neg$. In this paper, however, we do not consider classical negation.

[2] In this paper, we do not consider classical negation, thus, $X$ is simply a set of atoms.

[3] A maximal occurrence in $F$ of a formula of the form *not* $G$ is: a subformula *not* $G$ of $F$ such that there is no subformula of $F$ which (i) is of the form *not* $H$ for some $H$, and (ii) has *not* $G$ as its proper subformula.

$F$ is *equivalent* to a formula $G$ (denoted by $F \Leftrightarrow G$) if, for any consistent sets $X$ and $Y$ of literals, $X \models F^Y$ iff $X \models G^Y$. The equivalence of formulas shown by Lifschitz et al. is, among others:

1. $F, (G; H) \Leftrightarrow (F, G); (F, H)$ and $F; (G, H) \Leftrightarrow (F; G), (F; H)$.
2. $not\ (F, G) \Leftrightarrow not\ F; not\ G$ and $not\ (F; G) \Leftrightarrow not\ F, not\ G$.
3. $not\ not\ not\ F \Leftrightarrow not\ F$.

From the above equivalence, it follows that every formula can be converted to a strongly equivalent "disjunctive" normal form (called *SE-disjunctive normal form*), where a conjunct consists of atoms possibly preceded by *not* or *not not* .

The equivalence of two programs in a *strong* sense is defined as follows: program $P_1$ and $P_2$ are *strongly equivalent* if, for any consistent sets $X$ and $Y$ of literals, $X$ satisfies $P_1^Y$ iff $X$ satisfies $P_2^Y$. As an example, $\{ F \leftarrow G; H\}$ is strongly equivalent to $\{ F \leftarrow G,\ F \leftarrow H\}$.

In our framework for program transformation described below, however, the ordinary equivalence of programs is considered, i.e., whether they have the same answer sets or not, thus much weaker than strong equivalence. We can therefore utilize the above equivalent transformations in the following.

## 3 Negative Unfolding in Nested Expressions

We first recall negative unfolding rule by Pettorossi and Proietti (*PP-negative unfolding*, for short) [9][4], where transformation rules are defined for *locally stratified* programs.

In the following, the head and the body of a clause[5] $C$ are denoted by $hd(C)$ and $bd(C)$, respectively. Given a clause $C$, a variable in $bd(C)$ is said to be *existential* iff it does not appear in $hd(C)$. The other variables in $C$ are called *free* variables.

**Definition 1.** PP-Negative Unfolding
Let $C$ be a renamed apart clause in a locally stratified program $P$ of the form: $H \leftarrow G_1, not\ A, G_2$, where $A$ is an atom, and $G_1$ and $G_2$ are (possibly empty) conjunctions of literals. Let $D_1, \ldots, D_k$ with $k \geq 0$, be all clauses of program $P$, such that $A$ is unifiable with $hd(D_1), \ldots, hd(D_k)$, with mgus $\theta_1, \ldots, \theta_k$, respectively. Assume that:

(PP1) $A = hd(D_1)\theta_1 = \ldots = hd(D_k)\theta_k$, that is, for each $i$ $(1 \leq i \leq k)$, $A$ is an instance of $hd(D_i)$,
(PP2) for each $i$ $(1 \leq i \leq k)$, $D_i$ has no existential variables, and
(PP3) from $G_1, not\ (bd(D_1)\theta_1; \ldots; bd(D_k)\theta_k), G_2$, we get an equivalent disjunction $Q_1; \ldots; Q_r$ of conjunctions of literals, with $r \geq 0$, by first pushing *not* inside and then pushing ; outside.

[4] In [9], symbol $\neg$ (resp., $\vee$) is used for negation (resp., disjunction).
[5] We hereafter use "clause" instead of (program) "rule", in order to avoid confusion with "transformation rule".

By *PP-negative unfolding* wrt *not* $A$, we derive from $P$ the new program $P'$ by replacing $C$ by $C_1, \ldots, C_r$, where $C_i$ is the clause $H \leftarrow Q_i$, for $i = 1, \ldots, r$. □

We note that, in PP-negative unfolding, the application of DNE is built-in in the above condition (PP3). It is shown to work for locally stratified programs [9], while some care will be necessary for non-stratified programs (see Example 1).

### 3.1 Pre-negative Unfolding Rule

We first define the following *pre-negative unfolding* in nested expressions. DNE is considered in Sect. 3.2.

**Definition 2.** Pre-Negative Unfolding in Nested Expressions
Let $C$ be a renamed apart clause in program $P$ of the form:

$$C: \ H \leftarrow G_1, not\ A, G_2 \tag{1}$$

where $A$ is a positive literal, and $G_1$ and $G_2$ are (possibly empty) conjunctions of positive literals with possibly preceded by *not* or nested *not*. Let $D_1, \ldots, D_k$ ($k \geq 0$) be all clauses of program $P$ such that $A$ is unifiable with $hd(D_1), \ldots, hd(D_k)$, with mgus $\theta_1, \ldots, \theta_k$, respectively. Assume that conditions (PP1), (PP2) and (PP3) in Definition 1 hold, except that a disjunction $Q_1; \ldots; Q_r$ in Condition (PP3) is an *SE-disjunctive normal form* of $G_1, not\ (bd(D_1)\theta_1; \ldots; bd(D_k)\theta_k), G_2$.

Then, by *pre-negative unfolding wrt not* $A$ (of $C$ in $P$), we derive from $P$ the new program $P'$ by replacing $C$ by $C_1, \ldots, C_r$, where $C_i$ is the clause $H \leftarrow Q_i$, for $i = 1, \ldots, r$.

Similarly, when an unfolded atom is in nested form, that is,

$$C: \ H \leftarrow G_1, not\ not\ A, G_2 \tag{2}$$

then, the derived program $P'$ by *pre-negative unfolding wrt not* $A$ (of $C$ in $P$), we derive from $P$ the new program $P'$ by replacing $C$ by $C_1, \ldots, C_r$, where $C_i$ is the clause $H \leftarrow Q_i$, for $i = 1, \ldots, r$, and $Q_1; \ldots; Q_r$ is an SE-disjunctive normal form of $G_1, not\ not\ (bd(D_1)\theta_1; \ldots; bd(D_k)\theta_k), G_2$. □

The following proposition shows that pre-negative unfolding preserves the answer set of a program under suitable conditions.

**Proposition 1.** Let $P_0$ be a program and $P_1$ be the derived program by pre-negative unfolding from $P_0$. Then, the answer set $AS(P_0)$ is equivalent to $AS(P_1)$, if the following conditions hold:

1. pre-negative unfolding is *not self-unfolding*, that is, in clause (1) of Definition 2, $A$ is not an instance of $H$, or
2. when self-unfolding is applied, it is nested *not*, i.e., unfolded clause in Definition 2 is of the form (2), and $A$ is a variant of $H$. □

*Remark 1.* Due to the page limitation, all the proofs are omitted and will appear in the full version. Instead, we give some comments on the above proposition. The proof of $AS(P_0) \subseteq AS(P_1)$ is done without any condition in the above proposition, by simply showing that, for $X \in AS(P_0)$, $P_0^X = P_1^X$, i.e., the reduct of $P_0$ relative to $X$ is equivalent to that of $P_1$. To show the converse $AS(P_0) \supseteq AS(P_1)$, some conditions are necessary, as the following example shows:

*Example 2.* Let $P_0$ be

$$C_1 : \ p \leftarrow not\ q$$
$$C_2 : \ q \leftarrow not\ p$$
$$C_3 : \ p \leftarrow not\ p$$

Then, by pre-negative unfolding wrt *not* $p$ of $C_3$ in $P_0$, we derive new program $P_1 = (P_0 \setminus \{C_3\}) \cup \{p \leftarrow not\ not\ q, not\ not\ p\}$. $P_0$ has one answer set $\{p\}$, while $P_1$ has two answer sets $\{p\}$ and $\{q\}$. Therefore, $AS(P_0) \subseteq AS(P_1)$, but $AS(P_0) \not\supseteq AS(P_1)$. We note that, pre-negative unfolding of $C_3$ is done using itself, i.e., it is *self-unfolding* and *not* $p$ in $C_3$ is not nested *not*. □

*Example 3.* We reconsider Example 1. After pre-negative unfolding is applied to $P_0$ in Example 1, consider the following "intermediate" program:

$$P_{0.5} : \ p \leftarrow not\ not\ p$$
$$q \leftarrow not\ p$$

As the pre-negative unfolding applied is *not* self-unfolding, it holds from Proposition 1 that $AS(P_0)$=$AS(P_{0.5})$ = {{p},{q}}. □

### 3.2 Double Negation Elimination in Nested Expressions

Example 3 shows that the failure to preserve the answer set in transformation from $P_0$ to $P_1$ is due to *double negation elimination (DNE)* in nested expressions. We now consider when DNE preserves the answer set semantics in nested expressions. To do that, we prepare some definitions and notations.

Let $P$ be a program and $p$ a predicate. The definition of $p$ in $P$, denoted by $Def(p, P)$, is defined to be the set of all clauses of $P$ whose head predicate is $p$. *Predicate dependency* is defined as usual: a predicate $p$ *depends on* a predicate $q$ in $P$ iff either (i) there exists in $P$ a clause of the form: $p \leftarrow B$ such that predicate $q$ occurs in $B$ (either positively or negatively) or (ii) there exists a predicate $r$ such that $p$ depends on $r$ in $P$ and $r$ depends on $q$ in $P$. The *extended definition* of predicate $p$ in program $P$, denoted by $Def^*(p, P)$, is the set of clauses of the definition of $p$ and the definitions of all the predicates on which $p$ depends in $P$.[6]

[6] This notation is due to [9].

*DNE* rule is defined as follows:

**Definition 3.** Double Negation Elimination (DNE) in Nested Expressions
Let $P$ be a set of clauses and $C$ be a clause in $P$ of the form:

$$C:\ H \leftarrow G_1, not\ not\ A, G_2 \tag{3}$$

where $H$ and $A$ are positive literals, and $G_1, G_2$ are (possibly empty) conjunctions of positive literals with possibly preceded by *not* or nested *not*.

Then, by applying *DNE* to *not not A* (of $C$ in $P$), we derive from $P$ the new program $P' = (P \setminus \{C\}) \cup \{H \leftarrow G_1, A, G_2\}$ □

**DNE in Nested Expressions: Propositional Case**
We first consider propositional case for the ease of understanding.

**Proposition 2.** Equivalence of DNE: Propositional Case
Let $P_0$ be a propositional program and $C = h \leftarrow G_1, not\ not\ a, G_2$ be a clause in $P_0$. Let $P_1$ be the derived program from $P_0$ by applying DNE to *not not a* of $C$, i.e., $P_1 = (P \setminus \{C\}) \cup \{h \leftarrow G_1, a, G_2\}$. Then, $AS(P_0) = AS(P_1)$, if predicate $a$ does not depend on predicate $h$ in $P_0$. □

*Remark 2.* In Example 3, we note that $Def^*(p, P_{0.5})$=$\{p \leftarrow not\ not\ p\}$ and $p$ depends on itself, thus the condition of the proposition does not hold obviously.

From Proposition 2, we can apply DNE rule to non-locally stratified programs, if the condition of the proposition is satisfied as follows:

*Example 4.* DNE Rule Applied to a Non-Locally Stratified Program

$$\begin{array}{lll} P_0:\ p \leftarrow not\ q, not\ r & P_1:\ p \leftarrow not\ q, not\ not\ s & P_2:\ p \leftarrow not\ q, s \\ \quad q \leftarrow r & \quad q \leftarrow r & \quad q \leftarrow r \\ \quad r \leftarrow not\ s & \quad r \leftarrow not\ s & \quad r \leftarrow not\ s \\ \quad s \leftarrow not\ r & \quad s \leftarrow not\ r & \quad s \leftarrow not\ r \end{array}$$

By applying pre-negative unfolding wrt *not r* of the first clause in $P_0$, followed by DNE rule to *not not s*, we have $P_2$. In $P_1$, $s$ does not depend on $p$. The answer set $AS(P_0)$ of $P_0$ is $\{\ \{p, s\}, \{q, r\}\ \}$, which is equivalent to $AS(P_2)$. □

**DNE in Nested Expressions with Variables**
Next, we consider DNE rule with variables. In the proof of Proposition 2, we use a set of clauses $Def^*(a, P_0)$ as an "invariant" through the transformation of DNE. For such an approach to work in this case, introducing some *well-founded ordering (wfo)* on a clause will be necessary to guarantee the existence of such an invariant through the transformation. The following example will be of help to understand the situation.

*Example 5.* Adapted from [8]

$$P_0:\ p(X) \leftarrow not\ not\ p(s(X)) \qquad P_1:\ p(X) \leftarrow p(s(X))$$

We assume that the Herbrand universe has a constant "0". Then, $P_0$ has two answer sets $\phi$ and $\{p(0), p(s(0)), \ldots\}$, while $P_1$ has one answer set $\phi$. We note that there exists no well-founded ordering $\succ$ such that $p(t) \succ p(s(t))$ for each grounded clause in $P_0$. □

The above-mentioned considerations lead to the condition in the following Proposition.

**Proposition 3.** Equivalence of DNE with Variables
Let $P_0$ be a program and $C = H \leftarrow G_1, not\ not\ A, G_2$ be a clause in $P_0$. Let $P_1$ be the derived program from $P_0$ by applying DNE to *not not* $A$ of $C$, i.e., $P' = (P \backslash \{C\}) \cup \{H \leftarrow G_1, A, G_2\}$. Then, $AS(P_0) = AS(P_1)$, provided that there exists a well-founded ordering $\succ$ such that, for every *ground instance* $h \leftarrow l_1, \ldots, l_k$ $(0 \leq k)$ of a clause $C' = H \leftarrow L_1, \ldots, L_k$ in $P_0$, (i) $h \succeq l_i$ $(1 \leq i \leq k)$, and (ii) when DNE rule is applied to $L_i = not\ not\ A$ $(1 \leq i \leq k)$ of $C'$, $h \succ a$, where $l_i = not\ not\ a$. □

We note that the above condition coincides with that of Proposition 2 in propositional case. In locally stratified programs, a wfo is given for stratified negation, while, in Proposition 3, it is given for *nested not* to which DNE is applied, thus a program is not necessarily locally stratified (see Example 4).

For locally stratified programs, we can see that the conditions in *both* Proposition 1 and Proposition 3 are satisfied, when applying pre-negative unfolding, followed by DNE. We thus have the following corollary.

**Corollary 1.** Let $P$ be a locally stratified program. Then, PP-negative unfolding preserves the answer set of $P$, or its unique perfect model $PERF(P)$. □

## 4 A Framework for Unfold/Fold Transformation of Locally Stratified Programs

In this section, we consider a framework for unfold/fold transformation of *locally stratified* programs, where, in addition to negative unfolding, we have the other transformation rules such as *folding* and *replacement*. The framework proposed by Pettorossi and Proietti [9] is based on the original framework by Tamaki-Sato [10] for definite programs, while our framework given here is based on the generalized one by Tamaki-Sato [11]. As in [10,11], a suitable well-founded measure on the set of ground literals is necessary for the correctness proof of transformation. Since the well-founded measure $\mu$ in Definition 6 is defined using the stratum of a ground literal, it is shown that an *extra* folding condition (FC1) (Assumption 3 below), which is not required either for definite or stratified programs, becomes necessary only for *locally* stratified programs. To the best of our knowledge, such a folding condition (FC1) is not found in the literature. Besides, the replacement rules defined in the following are more general than those by Pettorossi and Proietti [9], in the sense that non-primitive (or non-basic) predicates are allowed in the rules.

Notwithstanding these differences, as our framework is mostly common with those by Tamaki-Sato [11], Pettorossi-Proietti [9] and Fioravanti et al. [3], we only

give terminology necessary to explain the differences below. See [11], [9] and [3] for further definitions not given here. Some familiarity with the correctness proof of unfold/fold transformation by Tamaki-Sato [10,11] would be helpful for understanding this section.

We divide the set of the predicate symbols appearing in a program into two disjoint sets: *primitive* predicates and *non-primitive* predicates.[7] We call an atom (a literal) with primitive predicate symbol a *primitive atom* (*primitive literal*), respectively. A clause with primitive (resp., non-primitive) head atom is called *primitive* (resp., *non-primitive*). We assume that every primitive clause in an initial program remains *untransformed* at any step in the transformation sequence defined below (Definition 9).

We denote by $ground(P)$ the set of ground instances of the clauses in program $P$ with respect to the Herbrand universe of $P$. A ground instance of some clause $C$ in $P$ is called an *inference by* $P$, and $C$ is called the *source clause* of the inference.

A *local stratification* is a total function $\sigma$ from the Herbrand base $HB(P)$ of program $P$ to the set $W$ of countable ordinals. We assume that $\sigma$ satisfies the following: For every primitive atom $A \in HB(P)$, $\sigma(A) = 0$. For a ground atom $A \in HB(P)$, $\sigma(not\ A) = \sigma(A) + 1$ if $A$ is non-primitive, and $\sigma(not\ A) = 0$ otherwise. For a conjunction of ground literals $G = l_1, \ldots, l_k$ $(k \geq 0)$, $\sigma(G) = 0$ if $k = 0$ and $\sigma(G) = max\{\sigma(l_i) : i = 1, \ldots, k\}$ otherwise.

### 4.1 Transformation Rules

We assume that an initial program, from which an unfold/fold transformation sequence starts, satisfies the following conditions.

**Assumption 1.** Initial Program
An *initial* program $P_0$ is divided into two disjoint sets of clauses, $P_{pr}$ and $P_{np}$, where $P_{pr}$ ($P_{np}$) is the set of primitive (non-primitive) clauses in $P_0$, respectively, and non-primitive predicates do not appear in $P_{pr}$. Moreover, $P_{pr}$ and $P_{np}$ satisfy the following conditions:

1. $P_{np}$ is a *locally stratified* program, with a local stratification $\sigma$. Moreover, we assume that $\sigma$ satisfies the following condition: For every ground non-primitive atom $A$,

$$\sigma(A) = sup(\{\sigma(l) \mid \text{literal } l \text{ appears in the body of a clause in } ground(Def(A, P_0))\}) \quad (4)$$

   where, for a set $S$ of countable ordinals, $sup(S)$ is the least upper bound of $S$ w.r.t $<$ (in $W$).
2. Each predicate symbol $p$ in $P_0$ is assigned a non-negative integer $i$ $(0 \leq i \leq I)$, called the *level* of the predicate symbol, denoted by $level(p) = i$. We define the *level of an atom (or literal)* $A$, denoted by $level(A)$, to be the level of

[7] In [9], primitive (non-primitive) predicates are called as *basic* (*non-basic*), respectively.

its predicate symbol, and the *level of a clause* $C$ to be the level of its head. For every primitive (resp. non-primitive) predicate symbol $p$, we assume that $level(p) = 0$ (resp., $1 \leq level(p) \leq I$). Moreover, we assume that every predicate symbol of a *positive* literal in the body of a clause in $P_0$ has a level not greater than the level of the clause. □

*Remark 3.* From the above assumption, an initial program consists of $I + 1$ *layers*, with the upward reference prohibited, though the recursion within a layer is allowed.[8] Technically, the notion of level is introduced to specify the condition of folding in [11], while a local stratification is utilized to define a well-founded measure for the correctness proof of the transformation. □

The transformation rules *positive* unfolding and folding are the same as those in [11], while we use PP-negative unfolding in Definition 1. We only recall the definition of folding in [11], in order to specify the folding conditions given later. The definition of a *molecule* and some notations are given in Appendix.

**Definition 4.** Folding, Reversible Folding
Let $P$ be a program and $A$ be an atom. A molecule $M$ is said to be a $P$-*expansion* of $A$ (by a clause $D$) if there is a clause $D: A' \leftarrow M'$ in $P$ and a substitution $\theta$ of free variables of $A'$ such that $A'\theta = A$ and $M'\theta = M$.[9]

Let $C$ be a clause of the form: $B \leftarrow \exists X_1 \ldots X_n(M, N)$, where $M$ and $N$ are molecules, and $X_1 \ldots X_n$ are some free variables in $M$. If $M$ is a $P$-expansion of $A$ (by a clause $D$), the result of *folding* $M$ of $C$ *by* $P$ is the clause: $B \leftarrow \exists X_1 \ldots X_n(A, N)$. The clause $C$ is called the *folded clause* and $D$ the *folding clause*.

The folding operation is said to be *reversible* if $M$ is the only $P$-expansion of $A$ in the above definition. □

To state assumptions on replacement, we need the following definitions.

**Definition 5.** Proof (with stratum $\sigma(A)$)
Let $P$ be a locally stratified program with a local stratification $\sigma$, $A$ be a ground atom true in $PERF(P)$. A finite ground SLS-derivation $T$ with its root $\leftarrow A$ is called a *proof* of $A$ (with stratum $\sigma(A)$) by $P$, if the computation rule always selects a *positive* non-primitive ground literal with stratum $\sigma(A)$, if any, in a goal, i.e., every node $v$ of $T$ satisfies the following conditions: (i) if $v$ is labeled with a goal $G$ with a selected *positive* non-primitive literal $B$ whose stratum is $\sigma(A)$, then $v$ has its child node whose label is the resolvent of $G$ with a ground clause in $ground(Def(B, P))$, and (ii) the leaf of $T$ is labeled with either □ (empty clause) or $\leftarrow l_1, \ldots, l_n$ $(n \geq 1)$, where $PERF(P) \models l_1, \ldots, l_n$ and $l_i$ $(n \geq i \geq 1)$ is either a negative literal, or primitive, or $\sigma(l_i) < \sigma(A)$.

[8] In [10], an initial program has two layers, where each predicate is classified as either *old* or *new*. Moreover, it has no primitive predicates.

[9] Two molecules $M$ and $N$ are considered to be identical, denoted by $M = N$, if $M$ is obtained from $N$ through permutation of conjuncts and renaming of existential variables. When two or more molecules are involved, they are assumed to have disjoint sets of variables, unless otherwise stated.

The definition of proof is extended from a ground atom to a conjunction of ground literals, i.e., a ground molecule, in a straightforward way. Let $L$ be a ground molecule and $T$ be a proof of $L$ by $P$. Then, we say that $L$ has a proof $T$ by $P$. $L$ is also said to be *provable* if $L$ has some proof by $P$.

For a *closed* molecule $M$, a proof of any ground existential instance[10] of $M$ is said to be a *proof of* $M$ by $P$. □

The following definition of the well-founded measure $\mu$ is a natural extension of that in [11] for definite programs, where $\mu$ is simply a weight-tuple. On the other hand, we consider as $\mu$ a pair of a stratum and a weight-tuple for locally stratified programs.

**Definition 6.** Weight-Tuple, Well-founded Measure $\mu$
Let $P_0$ be an initial program with $I+1$ layers with a local stratification $\sigma$, and $A$ be a ground atom. Let $T$ be a proof of $A$ (with stratum $\sigma(A)$) by $P_0$, and let $w_i$ $(1 \leq i \leq I)$ be the number of selected non-primitive positive literals of $T$ with level $i$. Then, the *weight-tuple* of $T$ (with stratum $\sigma(A)$) is an $I$-tuple $\langle w_1, \ldots, w_I \rangle$.

We define the *well-founded measure* $\mu(A)$ as follows:

$$\mu(A) = inf(\{\langle \sigma(A), w \rangle \mid w \text{ is the weight-tuple of a proof of } A \text{ (with stratum } \sigma(A))\}), \quad (5)$$

where $inf(S)$ is the minimum of set $S$ under the lexicographic ordering[11] over $W \times N^I$, where $W$ is the set of countable ordinals and $N$ is the set of natural numbers. For a ground molecule $L$, $\mu(L)$ is defined similarly. For a *closed* molecule $M$, $\mu(M) \stackrel{\text{def}}{=} inf(\{\langle \sigma(M'), w \rangle \mid w$ is the weight-tuple of a proof of $M'$, where $M'$ is a ground existential instance of $M\}$). □

Note that the above defined measure $\mu$ is *well-founded* over the set of ground molecules which have proofs by $P_0$. By definition, for a ground primitive atom $A$ true in $PERF(P_0)$, $\mu(A) = \langle 0, \langle 0, \ldots, 0 \rangle \rangle$.

**Definition 7.** truncation of weight-tuple and $\mu$
For $i$ $(1 \leq i \leq I)$, the $i$-th *truncation* of the weight-tuple $\langle w_1, \ldots, w_I \rangle$ is defined to be $\langle w_1, \ldots, w_i \rangle$. For a *closed* molecule $M$, the $i$-th *truncation* of the well-founded measure $\mu(M)$, denoted by $\mu_i(M)$, is defined by replacing $w$ by $\langle w_1, \ldots, w_i \rangle$ in the definition of $\mu(M)$ in Definition 6. □

**Definition 8.** Replacement Rule
A *replacement rule* $R$ is a pair $M_1 \Rightarrow M_2$ of molecules, such that $Vf(M_1) \supseteq Vf(M_2)$, where $Vf(M_i)$ is the set of free variables in $M_i$. Let $C$ be a clause of the form:$A \leftarrow M$. Assume that there is a substitution $\theta$ of free variables of $M_1$ such that $M$ is of the form: $\exists X_1 \ldots X_n(M_1\theta, N)$ for some molecule $N$ and some variables $X_1 \ldots X_n$ $(n \geq 0)$ in $Vf(M_1\theta)$. Then, the result of *applying* $R$ to $M_1\theta$ in $C$ is the clause: $A \leftarrow \exists X_1 \ldots X_n(M_2\theta, N)$.

[10] The definition of an *existential instance* of a molecule is given in Appendix.
[11] We use the inequality signs $>, \leq$ to represent this lexicographic ordering.

A replacement rule $M_1 \Rightarrow M_2$ is said to be *correct* w.r.t. an initial program $P_0$, if, for every ground substitution $\theta$ of free variables in $M_1$ and $M_2$, it holds that $M_1\theta$ has a proof by $P_0$ iff $M_2\theta$ has a proof by $P_0$. □

We are now in a position to state our assumptions on replacement rules.

**Assumption 2.** Replacement Rules
We assume that every replacement rule $R = M_1 \Rightarrow M_2$ is correct w.r.t. an initial program $P_0$, and

(RC0) $R$ is *consistent* with the well-founded measure $\mu$, that is, for every ground substitution $\theta$ for *free* variables of $M_1$ and $M_2$ so that $M_1\theta$ and $M_2\theta$ are provable by $P_0$, $\mu(M_1\theta) \geq \mu(M_2\theta)$ holds. Furthermore,
(RC1) $R$ is *consistent* with a local stratification $\sigma$, that is, for every ground substitution $\theta$ for free variables of $M_1$, $sup(\sigma(M_1\theta)) \geq sup(\sigma(M_2\theta))$, where, for a closed molecule $M$, $sup(\sigma(M))$ is defined to be $sup(\{\sigma(M') \mid M'$ is a ground existential instance of $M\})$. □

*Remark 4.* The replacement rule by Pettorossi-Proietti [9] is a special case of ours, where all of the literals appearing in $M_1 \Rightarrow M_2$ are primitive. In this case, Assumption 2 is trivially satisfied, since $\mu(M_1) = \mu(M_2) = \langle 0, \langle 0, \ldots, 0\rangle\rangle$ and $\sigma(M_1) = \sigma(M_2) = 0$ by definition. □

We can now define a transformation sequence as follows:

**Definition 9.** Transformation Sequence
Let $P_0$ be an *initial* program and $\mathcal{R}$ be a set of replacement rules satisfying Assumption 2. A sequence of programs $P_0, \ldots, P_n$ is said to be a *transformation sequence* with the input $(P_0, \mathcal{R})$, if each $P_n (n \geq 1)$ is obtained from $P_{n-1}$ by applying to a *non-primitive* clause in $P_{n-1}$ one of the following transformation rules: (i) positive unfolding, (ii) PP-negative unfolding, (iii) *reversible* folding *by* $P_0$ and (iv) some replacement rule in $\mathcal{R}$. □

We note that every primitive clause in $P_0$ remains *untransformed* at any step in a transformation sequence.

**Proposition 4.** Preservation of an Initial Local Stratification $\sigma$
Let $P_0$ be an initial program with a local stratification $\sigma$ and $P_0, \ldots, P_n$ $(n \geq 1)$ be a transformation sequence. Then, $P_n$ is locally stratified w.r.t. $\sigma$. □

*Remark 5.* The proof is basically similar to [3] except when folding and replacement are applied. In case of folding, $\sigma$ is preserved due to the definition of stratum (4) in Assumption 1 and the reversibility of folding, while the preservation of $\sigma$ through replacement follows from (RC1) in Assumption 2. □

### 4.2 The Correctness Proof of Unfold/fold Transformation

We need some assumptions on folding, which are the most "intricate" part in the correctness proof by Tamaki-Sato [10,11].

**Definition 10.** Descent Level of a Clause
Let $C$ be a clause appearing in a transformation sequence starting from an initial program $P_0$ with $I + 1$ layers. The *descent level* of $C$ is defined inductively as follows:

1. If $C$ is in $P_0$, the descent level of $C$ is the level of $C$ in $P_0$.
2. If $C$ is first introduced as the result of applying positive unfolding to some clause $C'$ in $P_i$ $(0 \leq i)$ of the form: $H \leftarrow G_1, A, G_2$ at a positive literal $A$, where $G_1$ and $G_2$ are (possibly empty) conjunctions of literals, then the descent level of $C$ is defined as follows:
   (a) The descent level of $C$ is the same as that of $C'$, if $A$ is primitive. Otherwise,
   (b) let $D_1, \ldots, D_k$ with $k \geq 1$, be all clauses of program $P_i$ , such that $A$ is unifiable with $hd(D_1), \ldots, hd(D_k)$. If it holds that, (i) for any ground substitution $\tau$ of variables in $C'$, $\sigma(H\tau) = \sigma(A\tau)$, (ii) for any ground substitution $\tau_j$ of variables in $D_j$ $(1 \leq j \leq k)$, $\sigma(hd(D_j)\tau_j) = \sigma(bd(D_j)\tau_j)$, and (iii) $C$ is the result of applying $D_j$ to $C'$, then the descent level of $C$ is the *smaller* of the descent level of $C'$ and that of $D_j$.
   (c) Otherwise, the descent level of $C$ is that of $C'$.
3. If $C$ is first introduced as the result of applying PP-negative unfolding to some clause $C'$, then the descent level of $C$ is that of $C'$.
4. If $C$ is first introduced as the result of folding, or applying some replacement rule to, some submolecule of the body of some clause $C'$, then the descent level of $C$ is that of $C'$. □

*Remark 6.* In the above definition, condition 2 for positive unfolding is different from that of the original one in [11], due to the difference of the well-founded measure $\mu$ in Definition 6, which is a pair consisting of a stratum and a weight-tuple. Another difference from that of [11] is condition 3 for PP-negative unfolding, which is introduced for dealing with locally stratified programs. The other conditions remain unchanged from [11]. We note that, from Condition 3 for PP-negative unfolding together with the following assumption (FC0) below, folding a clause to which only PP-negative unfolding is applied is prohibited. □

**Assumption 3.** Folding Conditions
In the transformation sequence, suppose that a clause $C$ is folded using a clause $D$ as the folding clause, where $C$ and $D$ are the same as those in Definition 4, i.e., $C$ is of the form: $B \leftarrow \exists X_1 \ldots X_n(M, N)$, $D = A' \leftarrow M'$ and $\theta$ is a substitution of free variables of $A'$ such that $D\theta = A \leftarrow M$. Then, we assume that, whenever folding is applied, the following conditions are satisfied:

(FC0) If there exists a ground substitution $\tau_0$ of variables of $C$ such that $\sigma(B\tau_0) = \sigma(M\tau_0)$, then the descent level of $C$ is *smaller* than the level of $D$. Moreover,
(FC1) for any ground substitution $\tau$ of free variables of $A$, the stratum of any ground existential instance of $M\tau$ is assumed to be equal to $\sigma(A\tau)$. □

When restricted to definite programs, the If-condition in (FC0) and (FC1) are always satisfied, the above assumption thus coincides with that of [11]. On the other hand, the condition (FC1) is newly introduced for the correctness proof in case of locally stratified programs, and its necessity is explained in Remark 7 below. The following *$\mu$-completeness* is crucial in the correctness proof in [11], since it is a sufficient condition for the correctness.

**Definition 11.** $\mu$-inference, $\mu$-complete
Let $P_0, \ldots, P_n, \ldots$ be a transformation sequence, and $\mu$ be the well-founded measure for $P_0$. An inference $A \leftarrow L$ by $P_n$ is called a *$\mu$-inference*, if $\mu_i(A) > \mu_i(L)$ holds, where $i$ is the descent level of the source clause of the inference.

$P_n$ is said to be *$\mu$-complete* if for every ground atom $A$ true in $PERF(P_0)$, there is a $\mu$-inference $A \leftarrow L$ by $P_n$ such that $PERF(P_0) \models L$. □

**Proposition 5.** Correctness of Transformation
Let $P_0, \ldots, P_n, \ldots$ $(n \geq 0)$ be a transformation sequence under Assumption 1, Assumption 2 and Assumption 3. Then, $PERF(P_n) = PERF(P_0)$. □

*Remark 7.* The proof is done in line with [11] by showing that (i) $P_0$ is $\mu$-complete, and (ii) $\mu$-completeness is preserved through the transformation, i.e., if $P_n$ is $\mu$-complete, then so is $P_{n+1} (n \geq 0)$. We thus sketch the necessity of folding condition (FC1) in case of locally stratified programs. Assume that $P_n$ is $\mu$-complete and let $A$ be an arbitrary ground atom true in $PERF(P_0)$, $A \leftarrow L$ be a $\mu$-inference by $P_n$ and $C$ be its source clause. Suppose further that $C = A' \leftarrow \exists X_1 \ldots X_m(M, N) \in P_n$ is folded into $C' = A' \leftarrow \exists X_1 \ldots X_m(B, N) \in P_{n+1}$, where $M$ is a unique $P_0$-expansion of $B$. There is a substitution $\theta$ of the variables of $A'$ and variables $X_1 \ldots X_m$ such that $A'\theta = A$ and $L$ is an existential instance of the closed molecule $M\theta, N\theta$. Let $K$ be a submolecule of $L$ which is an existential instance of $M\theta$. [12] We consider the case where $\sigma(A) = \sigma(K)$.

Let $i$ be the descent level of $C$, and $j$ the level of $B$. We note that $i < j$ from (FC0). From $\mu$-completeness, it holds that[13]

$$\mu_i(A) > \mu_i(L) = \mu_i(K) \oplus \mu_i(L - K) \tag{6}$$

From the definition of $\mu$ for a closed molecule (Definition 6), we have that $\mu_{j-1}(K) \geq \mu_{j-1}(M\theta)$. We can prove that $A \leftarrow B\theta, (L - K)$ is a $\mu$-inference by $P_{n+1}$, if the inequality (6) still holds when $\mu_i(K)$ is replaced by $\mu_i(B\theta)$ in it. In definite programs or stratified programs, we have that $\mu_{j-1}(M\theta) = \mu_{j-1}(B\theta)$ (thus $\mu_i(K) \geq \mu_i(B\theta)$) from the definition of $\mu_i$ and the fact that the stratum of *any* ground existential instance of $M\theta$ is equal to that of $B\theta$, regardless

[12] $L$ is thus of the form: $L = K, L_r$ for some submolecule $L_r$. We denote $L_r$ by $L - K$.
[13] For conjunctions $L_1, L_2$ of ground literals, $\mu(L_1) \oplus \mu(L_2)$ is defined to be $\langle m, w \rangle$, where $m = max\{\sigma(L_1), \sigma(L_2)\}$, and $w$ is the weight-tuple of a proof of $L_1, L_2$ with stratum $m$. The definition of its $i$-th truncation is defined similarly.

of its ground instantiation. However, it is not the case for locally stratified programs, and the condition (FC1) is therefore a sufficient condition for reconciling it. □

Our framework for unfold/fold transformation will be applicable to an initial program with a non-locally stratified $P_{pr}$, while keeping the other conditions in Assumption 1 the same. See Example 4, where $r$ and $s$ (resp. $p$ and $q$) are assumed to be primitive (resp. non-primitive). Suppose that $AS(P_{pr}) \neq \emptyset$. From Splitting Set Theorem by Lifschitz and Turner [5], it follows that, for every answer set $X$ of $P_{pr}$, there exists an answer set $X \cup Y$ of $P_0$, where $Y$ is a set of ground non-primitive atoms. Then, the only modifications we need for our transformation rules are the assumptions of replacement rules (Assumption 2), i.e., in place of considering a perfect model $PERF(P_0)$, a replacement rule should be correct w.r.t. *every* answer set of $P_0$, and the condition (RC0) should hold w.r.t. *every* answer set of $P_0$. Then, the proof of correctness of this extended framework is similarly done.

## 5 Conclusion

We have studied negative unfolding for logic programs, by considering it as a combination of pre-negative unfolding and double negation elimination (DNE). We have analyzed both transformation rules in terms of nested expressions, giving sufficient conditions for the preservation of the answer set semantics.

We then proposed a framework for unfold/fold transformation of locally stratified programs, which, besides negative unfolding, contains replacement rules, allowing a more general form than those proposed by Pettorossi and Proietti [9]. We have identified a new folding condition (FC1), which is not necessary for either definite or stratified programs, but is required only in case of locally stratified programs. Since our correctness proof is based on the well-founded measure $\mu$, which is defined in terms of the stratum of a ground literal, it seems inevitable to impose such an extra condition on folding. We leave for future research further investigation on a simpler sufficient condition on folding for practicality of the transformation rules.

Negative unfolding has been studied in other semantics for preservation of equivalence, such as Clark completion and 3-valued semantics ([8] for survey). Although our interest in this paper is on negative unfolding w.r.t. the answer set semantics, it will be interesting to study negative unfolding in the other semantics. In Answer Set Programming, equivalent transformations have been discussed in a sense stronger than our ordinary equivalence: strong equivalence ([6], [12]) and *uniform* equivalence (e.g., [2]), and others. One of the main purposes in Pettorossi-Proietti [9] is the use of unfold/fold transformation for verification, i.e., proving properties of the perfect model of a locally stratified program. We hope that our results will be a basis for verification in the answer set semantics beyond the class of locally stratified programs.

**Acknowledgement.** The author would like to thank anonymous reviewers for their constructive and useful comments on the previous version of the paper.

## References

1. Apt, K.R.: Introduction to Logic Programming. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 493–576. Elsevier, Amsterdam (1990)
2. Eiter, T., Fink, M., Tompits, H., Traxler, P., Woltran, S.: Replacements in Non-Ground Answer-Set Programming. In: Proc. of the 10th Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR 2006), pp. 340–351. AAAI Press, Menlo Park (2006)
3. Fioravanti, F., Pettorossi, A., Proietti, M.: Transformation Rules for Locally Stratified Constraint Logic Programs. In: Bruynooghe, M., Lau, K.-K. (eds.) Program Development in Computational Logic. LNCS, vol. 3049, pp. 291–339. Springer, Heidelberg (2004)
4. Kanamori, T., Horiuchi, K.: Construction of Logic Programs Based on Generalized Unfold/Fold Rules. In: Proc. the 4th Intl. Conf. on Logic Programming, pp. 744–768 (1987)
5. Lifschitz, V., Turner, H.: Splitting a Logic Program. In: Proc. the 11th Intl. Conf. on Logic Programming, pp. 23–38 (1994)
6. Lifschitz, V., Tang, L.R., Turner, H.: Nested Expressions in Logic Programs. Annals of Mathematics and Artificial Intelligence 25, 369–389 (1999)
7. Lloyd, J.W.: Foundations of Logic Programming, 2nd edn. Springer, Heidelberg (1987)
8. Pettorossi, A., Proietti, M.: Transformation of Logic Programs: Foundations and Techniques. J. of Logic Programming 19/20, 261–320 (1994)
9. Pettorossi, A., Proietti, M.: Perfect Model Checking via Unfold/Fold Transformations. In: Palamidessi, C., Moniz Pereira, L., Lloyd, J.W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Sagiv, Y., Stuckey, P.J. (eds.) CL 2000. LNCS (LNAI), vol. 1861, pp. 613–628. Springer, Heidelberg (2000)
10. Tamaki, H., Sato, T.: Unfold/Fold Transformation of Logic Programs. In: Proc. 2nd Int. Conf. on Logic Programming, pp. 127–138 (1984)
11. Tamaki, H., Sato, T.: A Generalized Correctness Proof of the Unfold/Fold Logic Program Transformation, Technical Report, No. 86-4, Ibaraki Univ., Japan (1986)
12. Turner, H.: Strong Equivalence Made Easy: Nested Expressions and Weight Constraints. Theory and Practice of Logic Programming 3(4&5), 609–622 (2003)

## Appendix

**Definition 12.** Molecule [11]
An existentially quantified conjunction $M$ of the form: $\exists X_1 \dots X_m(A_1, \dots, A_n)$ $(m \geq 0, n \geq 0)$ is called a *molecule*, where $X_1 \dots X_m$ are distinct variables called *existential variables* and $A_1, \dots, A_n$ are literals. The set of other variables in $M$ are called *free variables*, denoted by $Vf(M)$. A molecule without free variables is said to be *closed*. A molecule without free variables nor existential variables is said to be *ground*. A molecule $M$ is called an *existential instance* of a molecule $N$,

if $M$ is obtained from $N$ by eliminating some existential variables by substituting some terms for them.[14] □

The conventional representation of a clause $C : A \leftarrow A_1, \ldots, A_n$ $(n \geq 0)$, where $A_1, \ldots, A_n$ are a conjunction of literals, is equivalent to the following representation with explicit existential quantification: $A \leftarrow \exists X_1 \ldots X_m(A_1, \ldots, A_n)$, where $X_1 \ldots X_m$ $(m \geq 0)$ are variables of $A_1, \ldots, A_n$ not appearing in $A$.

[14] The variables in the substituted terms, if any, becomes free variables of $M$.

# Author Index