Proceedings of the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2012)

Elvira Albert
Complutense University of Madrid

Report CW 625, September 2012

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

Proceedings of the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2012)

Elvira Albert
Complutense University of Madrid

Report CW 625, September 2012

Department of Computer Science, KU Leuven

Abstract

This book contains the papers presented at the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2012) held on September 18-20, 2012 in Leuven. LOPSTR 2012 is co-located with PPDP 2012, the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming.

Keywords : Logic-based Programming, Synthesis, Transformation.
CR Subject Classification : I.2.2, F.3.1, F.4.1

Preface

This book contains the papers presented at the 22nd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2012) held on September 18-20, 2012 in Leuven. LOPSTR 2012 is co-located with PPDP 2012, the 14th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming.

Previous LOPSTR symposia were held in Odense (2011), Hagenberg (2010), Coimbra (2009), Valencia (2008), 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). Information about the conference can be found at: http://costa.ls.fi.upm.es/lopstr12/.

The aim of the LOPSTR series is to stimulate and promote international research and collaboration in logic-based program development. LOPSTR traditionally solicits contributions, in any language paradigm, in the areas of specification, synthesis, verification, analysis, optimization, specialization, security, certification, applications and tools, program/model manipulation, and transformational techniques. 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 this feedback in the published papers. The LOPSTR 2012 post-proceedings will be published in the Lecture Notes in Computer Science series of Springer-Verlag.

In response to the call for papers, 27 contributions were submitted from 19 different countries. The Programme Committee decided to accept 14 full papers and 2 extended abstracts, basing this choice on their scientific quality, originality, and relevance to the symposium. Each paper was reviewed by at least three Program Committee members or external referees. In addition to the 16 contributed papers, this volume includes the abstracts of the invited talks by two outstanding researchers: one LOPSTR invited talk by Tom Schrijvers (University of Ghent, Belgium) and a joint PPDP-LOPSTR invited talk by Jürgen Giesl (RWTH Aachen, Germany).

I want to thank the Program Committee members, who worked diligently to produce high-quality reviews for the submitted papers, as well as all the external reviewers involved in the paper selection. I am very grateful the LOPSTR 2012 Conference Co-Chairs, Daniel De Schreye and Gerda Janssens, and the local organizers for the great job they did in preparing the conference. I also thank Andrei Voronkov for his excellent EasyChair system that automates many of the tasks involved in chairing a conference. Finally, I gratefully acknowledge the sponsors of LOPSTR: The Association for Logic Programming and Fonds voor Wetenschappelijk Onderzoek Vlaanderen.

September 18, 2012 Elvira Albert
Madrid

Table of Contents

Program Committee

Elvira Albert	Complutense University of Madrid
Sergio Antoy	Dept of Computer Science, Portland State University
Demis Ballis	University of Udine
Henning Christiansen	Roskilde University
Michael Codish	Ben-Gurion University
Daniel De Schreye	Dept. Computer Science, KULeuven
Esra Erdem	Sabanci University
Maribel Fernandez	King's College London, Dept. of Computer Science
Carsten Fuhs	RWTH Aachen
John Gallagher	Roskilde University
Robert Glück	DIKU, Dept. of Computer Science, University of Copenhagen
Miguel Gomez-Zamalloa	Complutense University of Madrid
Rémy Haemmerlé	Technical University of Madrid
Geoff Hamilton	Dublin City University
Reiner Hähnle	Technical University of Darmstadt
Gerda Janssens	Katholieke Universiteit Leuven
Isabella Mastroeni	Universita' di Verona - Dipartimento di Informatica
Kazutaka Matsuda	The University of Tokyo
Paulo Moura	University of Beira Interior, Portugal
Johan Nordlander	Lulea University of Technology
Andrey Rybalchenko	TUM
Kostis Sagonas	Uppsala University
Francesca Scozzari	Università di Chieti-Pescara
Valerio Senni	DISP, University of Rome Tor Vergata, Roma (Italy)
German Vidal	MiST, DSIC, Universitat Politecnica de Valencia

Additional Reviewers

A
Amato, Gianluca
B
Bertolissi, Clara
Bubel, Richard
C
Chitil, Olaf
D
De Angelis, Emanuele
F
Falaschi, Moreno
Fioravanti, Fabio
G
Gorogiannis, Nikos
Grossniklaus, Michael
H
Hentschel, Martin
K
König, Arne
L
Lagoon, Vitaly
Lee, Joohyung
M
Montenegro, Manuel
N
Nicholson, Colin Jay
Nigam, Vivek
Nogueira, Vitor
O
Oetsch, Johannes
P
Popeea, Corneliu
Porto, António
Proietti, Maurizio
R
Riesco, Adrian
Ruemmer, Philipp
S
Sneyers, Jon
Stroeder, Thomas
T
Theil Have, Christian

Torella, Luca
Y
York, Bryant

Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs (Abstract)*

Jürgen Giesl[1], Thomas Ströder[1], Peter Schneider-Kamp[2], Fabian Emmes[1], and Carsten Fuhs[3]

[1] LuFG Informatik 2, RWTH Aachen University, Germany
[2] Dept. of Mathematics and Computer Science, University of Southern Denmark
[3] Dept. of Computer Science, University College London, UK

There exist many powerful techniques to analyze *termination* and *complexity* of *term rewrite systems* (TRSs). Our goal is to use these techniques for the analysis of other programming languages as well. For instance, approaches to prove termination of definite logic programs by a transformation to TRSs have been studied for decades. However, a challenge is to handle languages with more complex evaluation strategies (such as Prolog, where predicates like the *cut* influence the control flow).

We present a general methodology for the analysis of such programs. Here, the logic program is first transformed into a *symbolic evaluation graph* which represents all possible evaluations in a finite way. Afterwards, different analyses can be performed on these graphs. In particular, one can generate TRSs from such graphs and apply existing tools for termination or complexity analysis of TRSs to infer information on the termination or complexity of the original logic program.

More information can be found in the full paper [1].

References

1. J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting — a general methodology for analyzing logic programs. In *Proc. PPDP '12*. ACM Press, 2012.

* Supported by the DFG under grant GI 274/5-3, the DFG Research Training Group 1298 (*AlgoSyn*), and the Danish Council for Independent Research, Natural Sciences.

Search Combinators

Tom Schrijvers

Ghent University, Belgium,
tom.schrijvers@ugent.be

Abstract

The ability to model search in a constraint solver can be an essential asset for solving combinatorial problems. However, existing infrastructure for defining search heuristics is often inadequate. Either modeling capabilities are extremely limited or users are faced with a general-purpose programming language whose features are not tailored towards writing search heuristics. As a result, major improvements in performance may remain unexplored.

This talk introduces *search combinators*, a lightweight and solver-independent method that bridges the gap between a conceptually simple modeling language for search (high-level, functional and naturally compositional) and an efficient implementation (low-level, imperative and highly non-modular). By allowing the user to define application-tailored search strategies from a small set of primitives, search combinators effectively provide a rich *domain-specific language* (DSL) for modeling search to the user. Remarkably, this DSL comes at a low implementation cost to the developer of a constraint solver.

Acknowledgments This talk is based on joint work with Guido Tack, Pieter Wuille, Horst Samulowitz and Peter Stuckey, part of which appeared in the proceedings of CP 2011 [1].

References

1. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.: Search Combinators. In: Principles and Practice of Constraint Programming, 17th International conference, Proceedings. Springer (2011)

A declarative pipeline language for big data analysis

Henning Christiansen, Christian Theil Have,
Ole Torp Lassen and Matthieu Petit

Research group PLIS: Programming, Logic and Intelligent Systems
Department of Communication, Business and Information Technologies
Roskilde University, P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: {henning,cth,otl,petit}@ruc.dk

Abstract. We introduce BANpipe – a logic-based scripting language designed to model complex compositions of time consuming analyses. Declarative semantics are described together with alternative operational semantics facilitating goal directed execution, parallel execution, change propagation and type checking.

1 Introduction

Computations for biological sequence processing are often complex compositions of time consuming analyses, including calls to external resources and databases. The expected output, intermediate results and original input data are often huge files. To facilitate such computations, we introduce a new declarative scripting language called BANpipe. The language supports complex pipelines of Prolog programs, PRISM [10] models and other types of programs through rules which specify dependencies between computations.

BANpipe is a general pipeline programming language, but it is designed to support a special kind of annotation pipelines which we call Bayesian Annotation Networks (BANs) [3, 7]. A Bayesian Network is a directed acyclic graph where the nodes are conditional probability distributions and the edges represent conditional dependencies. A BAN is a Bayesian Network where nodes are instead (probabilistic) annotation programs and edges are input/output dependencies between programs. Inference in BANs is performed iteratively by evaluating each program at a time and using its output annotation as input for dependent programs. This is not only similar to the way forward analysis takes place in Bayesian networks, but also a nice fit to the pipeline paradigm.

Existing pipeline scripting languages, however, are not designed for the integration of Prolog and PRISM programs. As consequence, integration would have to rely on shell commands rather than providing a smooth integration at the language level. An other reason for designing a new language is that existing pipeline languages lack the desired balance between declarativity and expressiveness, cf. section 7. Declarative pipeline languages typically sacrifice expressiveness, resulting in verbose languages with limited capabilities for modeling

dynamic aspects of pipelines. Inversely, languages which favor expressiveness are often locked into a procedural semantics which exclude possibilities for automated parallelism, change propagation and management of result files.

BANpipe rules express dependencies between symbolically represented files that are automatically mapped to the underlying filesystem. The symbolic filenames may include logic variables which enables advanced control mechanisms, leading to compactly expressed pipelines. The declarative semantics of the language allow for useful extensions.

Execution of a pipeline script is goal directed, where only the desired result is specified and the system then executes programs necessary to achieve the result as entailed by the dependencies in the script. Computations entailed by multiple goals are only performed once and subsequently shared by the goals. The language enables incremental change propagation, where all depending files are recomputed recursively after a change to a component in the pipeline.

The language also lends itself to parallel execution; programs which can run in parallel are inferred from the conditional independencies in a script.

Additionally, many aspects of scripts can be statically checked, e.g., the language is extended with a type system which can be used to infer compatibility between file formats of intermediate files.

2 Syntax and informal semantics of BANpipe

The BANpipe language consists of scripts for controlling collections of programs that work on data files. We add another layer on top of the traditional file system, so that an arbitrary ground Prolog term can be used for identifying a file. We assume a *(local) file environment* that maps such file identifiers into files, via their real file name in the local file system or a remote resource. The syntax is embedded in Prolog so we inherit its notions of terms and variables[1] (written with capital letters). Our implementation use a convention that a Prolog constant whose name indicates a protocol is treated as an URL, and the files for terms are determined through the local file environment. For example:

- `'file:///a/b/c/datafile'`: refers to the file with the real name `datafile` in the `/a/b/c` directory in the local file system,
- `'http://a.b.c/file'`: refers to a file referenced using the http protocol.
- `f(7)`: may refer to a file in the local file system.

For ease of usage, we refer to Prolog terms expected to denote files as *file names*.

The programs referred to in BANpipe scripts are placed in modules, and a program defines a function from zero or more file to one or more files; a program may take options as additional arguments that modify the function being calculated. Programs are referred to in the body of BANpipes *dependency rules*, exemplified as follows.

```
file1, file2 <- m::prog([file3,file4],op1,op2).        (1)
```

[1] Terms with variables may become ground through a substitution as result of execution a script.

Here `prog` is a program in module `m`, taking two files `file3` and `file4`, plus options `op1` and `op2` as input. The rules explains how two output files `file1` and `file2` depend on `file3`, `file4`, namely being the result of applying the function (or *task*) given by `m::prog([-,-],op1,op2)`.

The sets of input files, output files and options are fixed for a given program.

File names in rules can be parameterized as shown in this rule:

```
f(N) <- m::prog(g(N)).                                    (2)
```

For any ground instance of `N`, this rule explains the dependency between two files, e.g., between `f(7)` and `g(7)` or `f(h(a))` and `g(h(a))`. Rules can be recursive as shown in the following example.

```
f(0) <- file::get('file:///data').                        (3)
f(N) <- N > 0, N1 is N-1 | m::prog(f(N1)).                (4)
```

Here, rule (3) applies a built-in module that takes care of simple file handling including the `get` facility that provides a copy of a file as shown; rule (4) includes a *guard*, which may precede the program call and is used for rule selection and for instantiating variables not given by the matching in the head. The recursion works as expected, and the evaluation of a `query f(2)` involves the calculation of files named `f(2)`, `f(1)` and `f(0)` from the local file with the real name `data`.

A BANpipe *script* is a sequence of rules defined by the following syntax, perhaps extended with definitions of Prolog predicates to be used in rule guards.

$$\langle \text{rule} \rangle ::= \langle \text{head} \rangle \texttt{ <- } \langle \text{body} \rangle \tag{5}$$

$$\langle \text{head} \rangle ::= \langle \text{file} \rangle_1 \texttt{,} \ldots \texttt{,} \langle \text{file} \rangle_m \quad m \geq 1 \tag{6}$$

$$\langle \text{file} \rangle ::= \textit{any Prolog term, as described above} \tag{7}$$

$$\langle \text{body} \rangle ::= \{ \langle \text{guard} \rangle \texttt{|} \} \ \langle \text{program call} \rangle \tag{8}$$

$$\langle \text{guard} \rangle ::= \textit{sequence of one or more Prolog calls} \tag{9}$$

$$\langle \text{program call} \rangle ::= \langle \text{module} \rangle \texttt{::} \langle \text{program name} \rangle \texttt{(} \langle \text{file} \rangle_1 \texttt{,} \ldots \texttt{,} \langle \text{file} \rangle_n \texttt{,} \textit{options} \texttt{)} \quad n \geq 0 \tag{10}$$

The following syntactic restrictions must hold for any BANpipe script.

- File names given as URLs are only allowed in program calls and cannot occur in rule heads.
- When a rule head contains file names with variables, any file name in that head must contain the same variables.

We assume that any predicate call in the guard of rule r is terminating, whenever the variables in the head of rule r are ground.

The selection of a rule for the evaluation of a query (a ground file name) must be unique. We capture the essential properties in the following definition and explain afterwards how such a selection function is implemented in practice.

Definition 1. *A selection function for a BANpipe script S is a partial function σ_S from non-URL ground Prolog terms to ground instances of rules of S such that if*

$$\sigma_S(f) = \big(f_1^{out}, \ldots, f_n^{out} \texttt{ <- } guard \mid m :: prog(f_1^{in}, \ldots f_m^{in})\big) \tag{11}$$

then guard evaluates to true, and it holds that

$$f = f_i^{out} \;\; for\ some\ i = 1, \ldots, n, \tag{12}$$

$$\sigma_S(f_i^{out}) = \sigma_S(f) \;\; for\ all\ i = 1, \ldots, n. \tag{13}$$

Any such instance is called a selection instance *for S.*

To simplify notation later, we may leave out the guard when referring to a selected instance, as it has made its duty for testing and variable instantiation, once the selection is effectuated.

Condition (12) states that the chosen rule is actually relevant for f, and condition (13) indicates that whenever a rule is applied, it calculates the unique results for all files mentioned in its head, independently of which request for a file that triggered the rule.

In the implemented system, the rules are checked in the order they appear in the script and the file names in their heads from left to right. If such a head file name unifies with the given f, and the guard succeeds, the rule is a candidate for selection. However, if the execution of a guard leads to a runtime error or does not instantiate all remaining variables in the rule, the search stops and no rule is selected. Condition (13) of def. 1 is undecidable, but it is straightforward to define sufficient conditions that can be checked syntactically; we do not consider this topic further here.

The evaluation of a query Q can be done in a standard recursive way, which will be described in more detail in section 4.

2.1 Defining programs and modules

As mentioned, the tasks activated from a BANpipe script are defined by programs that are grouped into named modules. How these modules are structured is not important for the understanding of the BANpipe script language, so we give here just a brief overview.

A module m must contain a designated interface file that defines each task through a Prolog predicate of the following form,

$$task\texttt{([}in\text{-}file_1, \ldots, in\text{-}file_n\texttt{],}opts\texttt{,[}out\text{-}file_1, \ldots, out\text{-}file_m\texttt{])} \tag{14}$$

that will be matched by a program call $m\texttt{::}task\texttt{(}\cdots\texttt{)}$ in a script as described above. In accordance with the precise semantics specified below, the file names (whether URLs or arbitrary Prolog ground terms) encountered by the script are mapped into references to actual files, which then are given to task predicates that access the files through standard input/output built-ins.

The interface file may contain all the code that implements the tasks but, typically, a module contains a number of source files, which may be shared by the different tasks. The execution of a task is done by the PRISM system, which is an extension to B-Prolog, and thus PRISM probabilistic inference and ordinary Prolog code is readily available. Calls to programs in other languages or web services are facilitated as well.

The system includes a sort of dynamic types that are specified in the interface files and not visible in the BANpipe scripts. This is described in more details in section 5.

3 Declarative semantics of BANpipe

Raw data and results of analyses are represented as data files. The language relies on no assumptions about the detailed structure of those files. We assume an unspecified domain

$$\mathit{DataFile} \tag{15}$$

including a $\perp$ element, which has the intuitive interpretation of a recognized unsuccessful result (rather that no results). Notationwise, we consider a tuple $\langle \perp, \ldots, \perp \rangle$ equivalent with $\perp$.

Program calls in a script denote *tasks* that are mappings from a (perhaps empty) sequence of data files into another sequence of data files. Thus

$$\mathit{Task} = \sum_{i=0,1,\ldots;j=1,2,\ldots} \mathit{Task}_{i,j} \tag{16}$$

$$\mathit{Task}_{i,j} = \mathit{DataFile}^i \to \mathit{DataFile}^j \tag{17}$$

Tasks are assumed to be strict in the sense that if any component of an input argument is $\perp$, the result is $\perp$. A task may also result in $\perp$ reflecting a runtime error or a Prolog failure.

Definition 2. *A* program semantics *is a function $[\![-]\!]$ from triples of module name, program name, and ground values for possible option parameters into tasks. For module mod, program prog (with n input and m output files), and option values opts, this function is indicated as*

$$[\![\mathit{mod} :: \mathit{prog}(\mathit{opts})]\!] \in \mathit{Task}_{n,m}. \tag{18}$$

Ground file names are used as synonyms for variables ranging over the domain *DataFile*; for a ground file name f, the corresponding unique variable, called a *file variable*, is denoted $\widehat{f}$, and this notation is extended to sets, $\widehat{F} = \{\widehat{f} \mid f \in F\}$; whenever f is an URL, $\widehat{f}$ is called an *URL variable*. (Partial) answers to queries are represented below as substitution for file variables into *DataFile* and are typically indicated by the letter Φ with possible subscripts. We recognize a special form of *URL substitutions* for URL variables only. For ease of notation, an URL substitution is assumed to provide a value for any URL variable which might be

$\bot$. The notation Φ_0 typically refers to an URL substitution. A substitution Φ is considered equivalent to the set of equations $\{f \doteq d \mid \Phi(f) = d\}$.[2]

The declarative meaning of a BANscript is given by a recursive systems of equations defined as follows.

Definition 3. *Given a BANscript S, a* defining equation for *a non-URL file name f is of the form*

$$\langle \widehat{f_1^{out}}, \ldots, \widehat{f_m^{out}} \rangle \doteq [\![mod :: prog(opts)]\!] \langle \widehat{f_1^{in}}, \ldots, \widehat{f_n^{in}} \rangle \tag{19}$$

where $f = \widehat{f_i^{out}}$ for some $i = 1, \ldots, m$, and S has a selection instance for f,

$$f_1^{out}, \ldots, f_m^{out} \texttt{<-} m :: prog(f_1^{in}, \ldots, f_m^{in}). \tag{20}$$

Given such S and $[\![-]\!]$, the defining set of equations for a query Q, *denoted $Eq(Q, S)$ is defined as the smallest set E of defining equations such that*

- *E contains a defining equation for any $q \in Q$,*
- *for any equation in E whose righthand side contains a non-URL variable $\widehat{f}$, E contains a defining equation for f.*

We say that a BANscript S is well-behaved *for a query Q if $Eq(Q, S)$ exists, is finite, and contains no circularities.*[3]

Notice that $Eq(Q, S)$ is defined independently of program semantics, so this definition is equally relevant for a standard semantics (i.e., the intended computations on real data files) as for different abstract semantics reflecting different program properties.

The solution to a set of equations is given as usual, as a substitution that maps variables to values, such that the left and right hand sides of each equation become identical when all functions are evaluated. Whenever a script S is well-behaved for a query Q, and Φ_0 is an URL substitution for the URL variables of $Eq(Q, S)$, there exists a unique solution for $Eq(Q, S) \cup \Phi_0$. To prove this, first convert each equation on tuples into equations of the form $\widehat{f_i} \doteq \ldots$ by projections, and then notice that all variables in the righthand sides can be eliminated in a finite number of steps. Condition (13) of def. 1 ensures that the resulting set of equations with variable-free righthand sides is unique. We can thus define:

Definition 4. *Let $[\![-]\!]$ be a program semantics, S a BANscript which is well-behaved for a query Q and Φ_0 an URL substitution, and let Φ be the solution to $Eq(Q, S) \cup \Phi_0$. The* answer *to Q (with respect to S, $[\![-]\!]$ and Φ_0) is the restriction of Φ to $\widehat{Q}$; the substitution Φ is referred to as the* full answer *to Q.*

The query is failed *whenever the solution assigns $\bot$ to any variable in $\widehat{Q}$.*

[2] We use symbol "$\doteq$" to distinguish equations that are explicit syntactic objects from the normal use of "=" as meta-notation.

[3] "No circularities" can be formalized by separating variables into disjoint, indexed strata, such that for an equation $\cdots V \cdots = \cdots V' \cdots$, that the stratum number for V' is always lower than the stratum number for V.

Alternatively, this semantics could have been formulated in terms of a fixed point or a least model, which is straightforward due to the well-behavedness property.

Well-behavedness is obviously an undecidable property as an arbitrary Turing machine can be encoded through recurrence of variables in the terms that represent file names. In practice, however, we expect the recursion patterns through file names to be rather simple, so a straightforward depth-first algorithm for calculation of $Eq(Q, S)$ is sufficient. The error message "(perhaps) not well-behaved" is then issued if this algorithm exceeds a certain recursion depth, or the selection of a defining equation for a particular file name fails.

4 Operational semantics

We present a number of alternative operational semantics for BANpipe as abstract algorithms. A script is executed in a state that contains a substitution mapping file variables into the *DataFile* domain, and which grows during the execution of a query.

4.1 Bottom-up operational semantics with memoization

The following algorithm defines an operational semantics that works in a bottom-up fashion, calculating all involved files from scratch. It ensures that any intermediate file needed to obtain the final results is evaluated exactly once, even if used in different program calls.

Algorithm 1: Bottom-up operational semantics for BANscript
Input: *A query Q, a BANscript S, program semantics $[\![-]\!]$ and initial substitution Φ_0;*
Output: *A substitution;*

$\Phi := \Phi_0$;
while $Eq(Q, S)$ *contains an equation*
$\langle \widehat{f_1^{out}}, \ldots, \widehat{f_m^{out}} \rangle \doteq [\![P]\!] \langle \widehat{f_1^{in}}, \ldots, \widehat{f_n^{in}} \rangle$
for which $\Phi(f_i^{out})$ is undefined for all $i = 1, \ldots, m$, and $\Phi(f_j^{in})$ is defined for all $j = 1, \ldots, n$
do $\Phi := \Phi[\widehat{f_1^{out}}/df_1, \ldots, \widehat{f_m^{out}}/df_m]$
where $\langle df_1, \cdots, df_m \rangle = [\![P]\!] \langle \Phi(\widehat{f_1^{in}}), \ldots, \Phi(\widehat{f_n^{in}}) \rangle$;
return Φ;

Theorem 1. *Given a program semantics $[\![-]\!]$, a BANscript S which is well-behaved for a query Q and an URL substitution Φ_0, Algorithm 1 returns the full answer Φ to Q. The solution to Q is a found as a the restriction of Φ to $\widehat{Q}$.*

Having the algorithm to return the full substitution produced, makes it possible to use it also for incremental maintenance of solutions, as we will see below in section 4.2.

Sketch of proof. Each step performed in the while loop in Algorithm 1 corresponds to a variable elimination step in $Eq(Q,S) \cup \Phi_0$. Furthermore, each such step that processes an equation of the form $\langle \widehat{f_1^{out}}, \ldots, \widehat{f_m^{out}} \rangle \doteq [\![P]\!] \langle \widehat{f_1^{in}}, \ldots, \widehat{f_n^{in}} \rangle$, will bind variables $\widehat{f_1^{out}}, \ldots, \widehat{f_m^{out}}$ to their final values in the resulting solution. □

This algorithm provides an abstract operational semantics for BANscripts which can be transformed into a running implementation by adding suitable data structures for representing the defining equations and the file environment (appearing in the algorithm as substitutions).

Algorithm 1 can also be applied for symbolic program executions in which the standard program semantics is replaced by one that calculates program properties, but the evaluation of the guards and patterns of recursion will be the same. Such symbolic executions are expected to run essentially faster than runs with a standard semantics. This principle is used below for predicting change propagation, section 4.2, and type inference, section 5.

4.2 Operational semantics for incremental change propagation

Our BANpipe language is intended for time consuming computations and will be used by researchers in an experimental style, with frequent modifications of the involved programs and data files. We describe here an extension of the operational semantics that accomodates such changes, and which reuses previous results where possible.

It involves an alternative program semantics for measuring change propagation, based on the domain $DataFile^{prop} = \{changed, unchanged, \bot\}$ and program semantics $[\![-]\!]^{prop}$ defined as follows: whenever the program *prog* (with n input and m output files) in module *mod* has been modified, or one of its input arguments (x_i below) has the value *changed*, we set

$$[\![mod :: prog(opts)]\!]^{prop} \langle x_1, \ldots, x_n \rangle = \langle \underbrace{changed, \ldots, changed}_{m \text{ times}} \rangle; \qquad (21)$$

otherwise (i.e., program not modified, input = $\langle unchanged, \ldots, unchanged \rangle$), the program call returns $\langle unchanged, \ldots, unchanged \rangle$.

We consider the difference between two substitutions Φ^{before} and Φ^{after}, intended to represent correct values for all file variables before and after the modification. This is characterized by a *propagation substitution* defined as follows.

$$Diff(\Phi^{before}, \Phi^{after})(\widehat{f}) = \begin{cases} unchanged & \text{whenever } \Phi^{before}(\widehat{f}) = \Phi^{after}(\widehat{f}) \\ changed & \text{otherwise} \end{cases} \qquad (22)$$

Changes in the set of URL files (i.e., where the ultimate input comes from) is characterized by a propagation substitution for URL variables.

We can now define an algorithm that predicts which files that need to be re-evaluated to obtain a consistent state following particular modifications of the programs and URL files. It is used as a helper in the algorithm for incremental re-evaluation later.

Algorithm 2: Change prediction for BANscript
Input: *A query Q, a BANscript S and URL change substitution Φ_0^{prop};*
Output: *A substitution of variables into $\{changed, unchanged\}$;*

Φ^{prop} := **run** *Algorithm 1 for Q, S, $[\![-]\!]^{prop}$ and Φ_0^{prop};*
return Φ^{prop};

We state the following weak correctness statement for Algorithm 2.

Theorem 2 (Soundness of the change prediction algorithm). *Let Φ_0^{before} and Φ_0^{after} be URL substitutions for the same set of variables into DataFile, and assume two program semantics $[\![-]\!]^{before}$ and $[\![-]\!]^{after}$. Let, furthermore,*

Φ_1^{before} be the full answer for Q wrt S, $[\![-]\!]^{before}$ and Φ_0^{before},
Φ_1^{after} the full answer for Q wrt S, $[\![-]\!]^{after}$ and Φ_0^{after}, and
Φ_1^{prop} the result of running Algorithm 2 for Q, S and $Diff(\Phi_0^{before}, \Phi_0^{after})$.

Then it holds for any ground file name f, that if $\Phi_1^{prop}(\widehat{f}) = unchanged$, then $\Phi_1^{before}(\widehat{f}) = \Phi_2^{before}(\widehat{f})$.

Sketch of proof. According to theorem 1, Φ_1^{before} (resp. Φ_1^{after}) can be characterized as the result of running Algorithm 1 for Q, S, $[\![-]\!]^{before}$ and Φ_0^{before} (resp. $[\![-]\!]^{after}$ and Φ_0^{after}). We can thus construct three synchronized runs of algorithm 1, calculating Φ_1^{before}, Φ_1^{after} and Φ_1^{prop} selecting the same equations in the same order. The theorem is easily shown by induction over these runs. □

Due to the sort of complex programs and data involved in our intended applications, we find it unlikely that an output of a program accidentally happens to be the same after a modification of input data or program text. It can be shown that algorithm 2 is optimal under this assumption, in the sense that if $\Phi_1^{prop}(\widehat{f}) = changed$, we will indeed have $\Phi_1^{before}(\widehat{f}) \neq \Phi_2^{before}(\widehat{f})$.

We can now give the algorithm for incremental maintenance of the solution for a given query and a script, when input data and programs called are modified. It uses algorithm 2 to identify which files that must be recomputed; their values are set to undefined in the current file substitutions, and a run of Algorithm 1 starting from this substitution leads to correctly updated file substitutions, i.e., the full answer for the query under the new circumstances, with as few program calls as possible.

Algorithm 3: Incremental maintenance for BANscript
Input: *A query Q, a BANscript S, a substitution Φ_1 produced by Algorithm 1 from some Q, S, some $[\![-]\!]^{before}$ and Φ_0^{before}, and an URL substitution Φ_0^{after};*
Output: *A substitution;*

$\Phi^{prop} :=$ **run** *Algorithm 2 for Q, S and Diff$(\Phi_0^{before}, \Phi_0^{after})$;*
$\Phi_2 := \Phi_1 \setminus \{ (\widehat{f}/\Phi_1(\widehat{f})) \mid \Phi^{prop}(\widehat{f}) = changed\}$;
$\phi_3 :=$ **run** *Algorithm 1 for Q, S, $[\![-]\!]^{after}$ and Φ_2;*
return Φ_3;

We leave out the correctness statement, which is straightforward to formulate and follows easily from the previous theorems.

4.3 A parallel operational semantics

BANpipe scripts are obvious candidates for parallel execution, as we can illustrate with this fragment of a script.

```
f0 <- m0::p0(f1,f2,f3).
f1 <- m1::p1('file:///data').
f2 <- m2::p2('file:///data').
f3 <- file::get('http://server/remoteFile').
```

Here `f1` and `f2` can be computed independently in parallel, and at the same time `f3` can be downloaded from the internet. When they all have finished, `m0::p0` can start running, taking as input the files thus produce, but not before.

Here we describe the structure of a parallel operational semantics as modification of Algoritm 1. We assume a *task manager* that maintains a queue of defining equations waiting ready to be executed. Whenever the sufficient resources are available, e.g., a free processor core plus a suitable chunk of memory, it can take an arbitrary equation from the queue and start its evaluation in a new process. When the processing of an equation is finished, it sends a signal about this, returning the result. Seen from the calling control algorithm, the task manager can receive messages of the form

enqueue(e), e being a defining equation,

and sends messages back of the form

finished($e, df_1, df_2, \ldots$), e being a defining equation, $df_1, df_2, \ldots \in DataFile$.

Such a message should guarantee that the task referred to in e has been applied correctly in order to produce the resulting file values $df_1, df_2, \ldots$ according to the standard semantics $[\![-]\!]$. A parallel operational semantics can now be given by the following abstract algorithm.

Algorithm 4: Parallel operational semantics for BANscript
Input: *A query Q, a BANscript S, program semantics $[\![-]\!]$ and initial substitution Φ_0;*
Output: *A substitution;*

$\Phi := \Phi_0$;
$E := Eq(Q, S)$; // *Equations not yet enqueued*
$F := \emptyset$; // *Equations that have been processed*
while $F \neq Eq(Q, S)$ **do**
  while *there is an* $e \in E$ *of the form*
  $$\langle \widehat{f_1^{out}}, \ldots, \widehat{f_m^{out}} \rangle \doteq [\![P]\!] \langle \widehat{f_1^{in}}, \ldots, \widehat{f_n^{in}} \rangle$$
  for which $\Phi(f_i^{out})$ *is undefined for all* $i = 1, \ldots, m$, *and* $\Phi(f_j^{in})$ *is defined for all* $j = 1, \ldots, n$
  do enqueue(e); $E := E \setminus \{e\}$;
  await message finished($e', df_1, df_2, \ldots$);
  $\Phi := \Phi[\widehat{f'^{out}_1}/df_1, \ldots, \widehat{f'^{out}_{m'}}/df_{m'}]$
  where $e' = \big(\langle \widehat{f'^{out}_1}, \ldots, \widehat{f'^{out}_{m'}} \rangle \doteq \cdots\big)$;
  $F := F \cup \{e'\}$;
return Φ;

Correctness is straightforward as this algorithm performs exactly the same file assignments as algorithm 1. We refrain from a formal exposition.

This algorithm is implemented in our system for a multicore computer, but it should also work for other architectures such as grids and clusters. We are considering an enhanced implementation that combines Algorithms 3 and 4 such that each time a resource is reported modified, the maximum number of processors are put to work to restore consistency. This may involve stopping active processes, or removing them from the queue, when input files are outdated.

5 Types and type inference for BANpipe scripts

The system includes a dynamic type system such that, for a given program call, the output files are assigned types based on the types of the input files. These types are programmer-defined and may not indicate anything about the internal structure of the file. It is up to the programmer to associate a meaning with the types, and they are only used by the system for checking the overall sensibility of a script. Types are not visible in a script, but are managed through optional declarations in the interface files (cf. section 2.1) and are checked separately by a symbolic execution of the program as explained in the following.

A type can be any Prolog term. An URL file has a default type `file`, which may be coerced into a more specific type by a variant of the `file::get` task, illustrated as follows.

```
f <- file::get(['http::/server/file.html'],type(text(html))).
```

A task specified as in (14) may have an associated, polymorphic type declaration of the following form, specifying requirements for its input and output files.

$$\texttt{type}(prog, \texttt{([}Type_1^{in}, \ldots, Type_n^{in}\texttt{],}options\texttt{,[}Type_1^{out}, \ldots, Type_m^{out}\texttt{])} \qquad (23)$$

Each $Type_i^{in/out}$ and *options* are Prolog terms, possibly with variables, and any variable in a $Type_j^{out}$ must occur in some $Type_k^{in}$ or *options*. Thus, if the types for n actual input files plus actual option values simultaneously unifies with $Type_1^{in}, \ldots, Type_n^{in}, opts$, unique ground instances are created for $Type_1^{out}, \ldots, Type_m^{out}$, which are then assigned as types for m output files.

Correct typing of a well-behaved BANscript S with respect to a given query Q are formalized through a type semantics $[\![-]\!]^{type}$. For each task with n input and m output files, assuming a type declaration as in (23) above, we define,

$$\begin{gathered}[\![mod\texttt{::}prog(opts)]\!]^{type}\langle x_1, \ldots, x_n\rangle = \langle Type_1^{out}, \ldots, Type_m^{out}\rangle\rho \\ \text{where } \rho \text{ is the unifier of } \langle x_1, \ldots, x_n, opts\rangle \\ \text{and } \langle Type_1^{in}, \ldots, Type_n^{in}, options\rangle\end{gathered} \qquad (24)$$

If the mentioned unifier does not exists, the result is instead $\bot$.

We can now use Algorithm 1 with this semantics for type checking a BANpipe script with respect to a given query as a symbolic execution of the program. The initial substitution Φ_0^{type} maps any URL variable to the type `file`.

Algorithm 5: Type inference for BANscript
Input: *A query Q, a BANscript S;*
Output: *A substitution of variables into types;*

Φ^{type} := **run** *Algorithm 1 for Q, S, $[\![-]\!]^{type}$ and Φ_0^{type};*
return Φ^{type};

If, in the resulting substitution, any file is mapped to $\bot$, we say that type checking of S failed for Q; otherwise type checking succeeded. Taking $[\![-]\!]^{type}$ as definition of correct typing, correctness of this algorithm is a consequence of theorem 1.

6 Examples

We exemplify BANpipe using examples drawn from biological sequence analysis and machine learning. In the first example we present a simple gene prediction pipeline and in the second example we show how such a pipeline can be extended with recursive rules used to implement self-training.

6.1 A basic gene prediction pipeline

The following is an example of a simple gene prediction pipeline, corresponding to experiments previously reported [3]. The premise is to train a gene finder expressed as a PRISM model using some of the known genes of the *Escherischia Coli* genome (the training set) and verify its prediction accuracy on a different set of known genes (the test set). First we get some initial data files; namely a genome sequence (`fasta_seq`) and a list of reference genes (`genes_ptt`),

```
fasta_seq <- file::get(['ftp://ftp.ncbi.nih.gov/.../NC_000913.fna']).
genes_ptt <- file::get(['ftp://ftp.ncbi.nih.gov/.../NC_000913.ptt']).
```

The fetched files are parsed into suitable format (Prolog facts),

```
genome <- fasta::parse([fasta_seq]).
genes(reference) <- ptt::parse([genes_ptt]).
```

We then extract all open reading frames (`orfs`) from the genome, and divide them into a training set and a test set,

```
orfs <- sequence::extract_orfs([genome]).
orfs(training_set), orfs(test_set) <- file::random_split([orfs],seed(42)).
```

Slightly simplified, open reading frames are subsequences of the genome which may contain genes. The `random_split` program divides the `orfs` randomly into the two files `orfs(training_set)` and `orfs(test_set)`. The process is deterministic due to the `seed(42)` option, i.e. it will split `orfs` in the same way if it is rerun. Next, we extract known genes corresponding to each set,

```
genes(Set) <- ranges::intersect([genes(reference),orfs(Set)]).
```

The `ranges` module contains tasks which deal with a files containing particular facts which besides representing sub-sequences also includes their positions in a genome. The `intersect` task finds all facts from `genes(reference)` where the represented sub-sequences are completely overlapped[4] by a member of `orfs(Set)`. It is used here to find the reference genes that belong to some `Set`, i.e., either the `training_set` or the `test_set`. This concludes the preparation of data files and we turn to the rules for the gene finder,

```
params <- genefinder::learn([genes(training_set)]).
predictions <- genefinder::predict([orfs(test_set), params]).
report <- accuracy::measures([genes(test_set), predictions]).
```

The `genefinder` module contains a PRISM based gene finder and the `learn` task bootstraps and invokes PRISMs machine learning procedure from the facts in the file `genes(traning_set)`. The resulting `params` file is a parameterization for the PRISM model. Using this parameterization, the task `predict` probabilistically predicts which orfs in `orfs(test_set)` represent genes, resulting in the file `predictions`. Finally, the accuracy of the `predictions` are evaluated with regards to the reference genes, `genes(test_set)`, by the task `accuracy::measures` which calculates, e.g., sensitivity and specificity.

[4] For the biologically inclined; we require this overlap to be in the same reading frame.

6.2 Self-training

Self-training has been demonstrated to yield improved gene prediction accuracy [1]. A self-training gene finder can be expressed by mutually recursive rules,

```
known_genes <- ...
self_learn(1) <- genefinder::learn([known_genes]).
self_learn(N) <- N > 1, N1 is N-1 | genefinder::learn([predict(N1)]).
predict(N) <- genefinder::predict([self_learn(N)]).
```

To elaborate: `known_genes` (obtained somehow) is the starting point for training and `self_learn(1)` is the parameter file resulting from training on `known_genes`. The second `self_learn(N)` rule is the recursive case, which learns parameters from the predictions of the the previous iteration. The goal `predict(N)` produces a set of gene predictions based on the parameters of obtained from `self_learn(N)`. For instance, the goal `predict(100)` corresponds to predictions after 100 iterations of self-training.

It is straight-forward to extend this example to more advanced self-training, e.g., co-training [2] where multiple models generate training data for each other.

7 Related work

Computational pipelines are ubiquitous. The classic example is Unix pipes, which feed the output of one program into another. Declarative pipeline languages with non-procedural semantics goes back at least to the make utility [5].

The importance of pipelines for biological sequence analysis has been acknowledged [9] and there are a variety of biological pipeline languages to choose from, e.g., EGene[4], BioPipe[6], DIYA[12], and SKAM[8]. The first three are configured using a static XML format with limited expressivity. DIYA targets annotation of bacterial genomes which has also been a motivating case for us. SKAM is a Prolog based pipeline with a syntax that resembles makefiles, but which borrows Prologs expressive power and provides built-in support for iteration. Dependencies are realized through nesting of functors, but are specified between tasks rather than files.

The family of concurrent logic languages [11] have syntactical and semantic similarities to BANpipe. Rules have guards and the successful execution of a guard implies a committed choice to evaluate the body of a rule. In *flat* variants of the these languages, guards are restricted to a predefined set of predicates as opposed to arbitrary used defined predicates. BANpipe allows user defined predicates, but for the semantics to be well-defined, these are subject to obvious restrictions, e.g. they should terminate. BANpipe rules have a single goal in the rule body, whereas concurrent logic languages typically have conjunctions of goals. These languages execute in parallel and synchronize computations by suspension of unification, which may be subject to certain restrictions. For instance, in Guarded Horn Clauses [13], the guard is not allowed to bind variables in the head and the body may not bind variables in the guard. BANpipe have more restricted assumptions; the guard never binds variables in the head and the body never binds variables in the head or the guard.

8 Conclusions

BANpipe is a declarative logic-language for modeling of computational pipelines with time consuming analyses. We gave abstract and operational semantics for BANpipe, which was extended for change propagation and parallelism and type checking. The language was illustrated with examples inspired from gene finding.

Our implementation of BANpipe supports all the concepts and extensions presented in this paper. The language is implemented as an interpreted domain specific language in B-Prolog/PRISM and the implementation is mature enough to handle pipelines of considerable size. Future versions may support other Prolog systems and adapt to a distributed setting by realizing tasks as web services.

Acknowledgement This work is part of the project "Logic-statistic modeling and analysis of biological sequence data" funded by the NABIIT program under the Danish Strategic Research Council.

References

1. Lomsadze A. Besemer J. and Borodovsky M. Genemarks: a self-training method for predicition of gene starts in microbial genomes. implications for finding sequence motifs in regulatory regions. *Nucleic Acids Research*, 29:2607–2618, 2001.
2. A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 92–100. ACM, 1998.
3. H. Christiansen, C. T. Have, O. T. Lassen, and M. Petit. Bayesian Annotation Networks for Complex Sequence Analysis. In *Technical Communications of the 27th International Conference on Logic Programming (ICLP'11)*, volume 11 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 220–230. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011.
4. A. M. Durham, A. Y. Kashiwabara, F. T. G. Matsunaga, P. H. Ahagon, F. Rainone, L. Varuzza, and A. Gruber. Egene: a configurable pipeline generation system for automated sequence analysis. *Bioinformatics*, 21(12):2812–2813, 2005.
5. S. I. Feldman. Make – A program for maintaining computer programs. *Software – Practice and Experience*, 9(3):255–265, March 1979.
6. S. Hoon, K.K. Ratnapu, J.M. Chia, B. Kumarasamy, X. Juguang, M. Clamp, A. Stabenau, S. Potter, L. Clarke, and E. Stupka. Biopipe: A flexible framework for protocol-based bioinformatics analysis. *Genome Research*, pages 1904–1915, 2003.
7. O. T. Lassen. *Compositionality in probabilistic logic modelling for biological sequence analysis*. PhD thesis, Roskilde University, 2011.
8. C. Mungall. Skam - skolem assisted makefiles. `http://skam.sourceforge.net/`.
9. W. S. Noble. A quick guide to organizing computational biology projects. *PLoS Comput Biol*, 5(7):e1000424, 07 2009.
10. T. Sato and Y. Kameya. Prism: a language for symbolic-statistical modeling. In *International Joint Conference on Artificial Intelligence*, volume 15, pages 1330–1339, 1997.
11. E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412, September 1989.
12. A. C. Stewart, B. Osborne, and T. D. Read. Diya: a bacterial annotation pipeline for any genomics lab. *Bioinformatics*, 25:962–963, 2009.
13. K. Ueda. Guarded horn clauses. Technical Report TR-103, ICOT, Tokyo, 1985.

Semantic code clones in logic programs (extended abstract)

Céline Dandois * and Wim Vanhoof

University of Namur - Faculty of Computer Science
21 rue Grandgagnage, 5000 Namur (Belgium)
{cda,wva}@info.fundp.ac.be

Abstract. In this extended abstract, we study what is a semantic clone pair in a logic program. In contrast with our earlier work, that focused on simple syntactic equivalence for defining clones, we propose a more general approximation based on transformation rules. This new definition captures a wider set of clones, and it allows to formally define the conditions under which a number of refactorings can be applied.

1 Introduction

Code duplication, also called *code cloning*, occurs intuitively when two or more source code fragments have an identical or sufficiently similar computational behaviour, independent of them being textually equal or not. Those fragments are described as *duplicated* or *cloned*. Clone detection has received a substantial amount of attention in recent years [16], but there is no standard definition for what constitutes a clone and the latter's definition is often bound to a particular detection technique [15]. Not unsurprisingly, most definitions and the associated detection techniques are based on somehow comparing the syntactical structure of two code fragments as a rough approximation of their semantics. Some examples are the recent abstract syntax-tree based approaches for Erlang [7] and Haskell [2], as well as our own work [4] in the context of logic programming. These syntax-based approaches suffer from a number of limitations. On the one hand, syntactical equivalence is too basic a characterisation for defining, *in a simple and uniform way*, the conditions under which a number of refactorings that aim at removing duplication from a logic program [18, 22] can be applied. On the other hand, looking at syntactical equivalence only, they are unable to classify as cloned slightly different computations that yield nevertheless the same result such as, in a logic programming setting, the goals X *is* 10 and X *is* $5 * 2$. Or, in other words, they cannot capture *semantic equivalence* of code fragments, even in cases where such equivalence could rather simply be established.

As a somewhat contrived but prototype example, representative of various real code clones, consider the following predicate definitions:
$p([], I, J, 0, 0) \leftarrow$
$p([H|T], 1, 1, H, H) \leftarrow$

* F.R.S.-FNRS Research Fellow

$p([H|T], 1, J, S, P) \leftarrow J > 1,\ J1\ is\ J-1,\ p(T, 1, J1, S1, P1),\ S\ is\ S1+H,\ P\ is\ P1*H$
$p([H|T], I, J, S, P) \leftarrow I > 1,\ J \geq I,\ I1\ is\ I-1,\ J1\ is\ J-1,\ p(T, I1, J1, S, P)$

$q([], 0) \leftarrow$
$q([H|T], S) \leftarrow even(H),\ q(T, S1),\ add(S1, H, S)$
$q([3|T], S) \leftarrow q(T, S1),\ add(S1, 3, S)$

$add(X, Y, Z) \leftarrow Z\ is\ X + Y \qquad\qquad even(N) \leftarrow 0\ is\ N\ mod\ 2$

An atom of the form $p(L, I, J, S, P)$ succeeds if S and P represent, respectively, the sum and the product of the elements in the list L between the positions I and J, while $q(L, S)$ succeeds if S is the sum of all elements in the list L, these being either even or the value 3. While syntactically very different, it is clear that $p/5$ and $q/2$ share some common functionality that may be worthwhile to exploit in a refactoring setting. Indeed, both predicates compute the sum of (a subset of) the elements of a list, possibly verifying a special condition. Obviously, this functionality is hard to detect based on the result of syntactically comparing the atoms in the definitions of $p/5$ and $q/2$, as would be the case by [4].

In this work, which is work in progress, we define the notion of a *semantic* clone pair in a logic programming setting. While semantic equivalence is in general undecidable, we provide an approximation of our definition – based on well-known program transformations – that allows to formally capture the relation between cloned code fragments such as the functionality shared by $p/5$ and $q/2$ from above, thereby going far beyond syntactical equivalence and, as such, substantially extending our previous work on the subject [4].

2 Defining semantic clones

In what follows, we consider an extended form of logic language including higher-order constructs [10, 3], and we assume the reader to be familiar with the basic logic programming concepts [1]. To simplify the presentation of the different ideas, we restrict ourselves to *definite programs*, i.e. programs without negation. Inspired from [3], the syntax that we will use is based on a countably infinite set $\mathcal{V}$ of *variables* and a countable set $\mathcal{N}$ of *relation names*. The latter set contains what is called functors and predicate names in first-order logic (and by the way, it eliminates the distinction between those entities), including built-in ones such as the unification operator $= /2$. The set $\mathcal{T}$ of *terms* satisfies the following conditions: $\mathcal{V} \cup \mathcal{N} \subseteq \mathcal{T}$ and $\forall x, t_1, \ldots, t_n (n \geq 1) \in \mathcal{T} : x(t_1, \ldots, t_n) \in \mathcal{T}$. This allows to use partial terms, as $add(X)(2, Z)$. A (positive) *atom* is a term. A *clause* is of the form $H \leftarrow B_1, \ldots, B_s$, where H, the *head* of the clause, is an atom of the particular form $h(t_1, \ldots, t_n)$ where $h/n \in \mathcal{N}$ and $t_1, \ldots, t_n (n \geq 0) \in \mathcal{T}$, and $B_1, \ldots, B_s$, the *body* of the clause, is a conjunction of atoms, also called a *goal*. A *predicate* is defined by a sequence of clauses sharing the same predicate symbol and arity. Finally, a *program* is formed by a set of predicates.

Besides, we will use some auxiliary functions. First, *dom* returns the domain of a given function. Then, let G be a subgoal from a clause $H \leftarrow L, G, R$ where

L and R are possibly empty subgoals, $vars(G)$ returns the set of variables of G and $exvars(G)$ returns the set of variables of G that are shared with L and R.

We define a *code fragment* as either (1) a predicate definition, or (2) a subsequence of clauses belonging to some predicate definition, or (3) a subgoal in some clause body. Given a code fragment F, we define the *predicate definition relative to* F, denoted $rp(F)$, as follows: if F is a predicate definition or a subsequence of clauses, then $rp(F)$ is F itself ; if, however, F is a goal, $rp(F)$ is defined by $relp(exvars(F)) \leftarrow F$ where $relp$ is a unique relation name not used in the program at hand. According to the notation F/n, the natural n represents the *arity* of the code fragment F and equals the arity of $rp(F)$.

Furthermore, we extend the original notion of variance. Given two atoms $B \equiv p(t_1, \ldots, t_m)$ and $B' \equiv q(t'_1, \ldots, t'_n)$ $(m \leq n)$, and an injective argument mapping $\phi : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$, we say that B is a *ϕ-variant* of B' if and only if, after rewriting, the atoms $r(t_1, \ldots, t_m)$ and $r(t'_{\phi(1)}, \ldots, t'_{\phi(m)})$ (where r is a new relation name) are variants. Given predicates p/m and q/n $(m \leq n)$ defined by the same number of clauses, and an injective argument mapping $\phi : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$, we say that p/m is a *ϕ-variant* of q/n if and only if their clause heads and recursive calls are pairwise ϕ-variants and the rest of their clause bodies are pairwise variants.

Our main definitions will be based on the operational semantics of logic programming [1, 10, 3], and more precisely, on the concept of computed answer substitutions [8, 1]. The computed answer substitution (CAS) semantics may be expressed as the following function [12]: $CAS : P^+ \times Q^+ \rightarrow (\mathfrak{P}(Subst), \leq)$, where P^+ is the set of definite programs, Q^+ is the set of definite queries, and $(\mathfrak{P}(Subst), \leq)$ is the powerset of the set of substitutions ordered by set inclusion. In other terms, we have: $CAS(P, Q) = \{\theta \mid P \vdash Q\theta\}$. This semantics is *relevant*, which means that, given a program and a query, its value comes only from the query and the clauses on which the query depends [12]. An empty set stands for a failed query. Note that for SLD-resolution, we adopt the standard clause and atom selection rules of Prolog, respectively top-down and left-to-right.

We also define the notion of *CAS semantic ϕ-equivalence* of two queries: given a predicate p/m and a predicate q/n $(m \leq n)$ from a program P, an injective argument mapping $\phi : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$, a call to p of the form $p(t_1, \ldots, t_m)$ and a call to q of the form $q(t'_1, \ldots, t'_n)$, we say that $CAS(P, p(t_1, \ldots, t_m)) = \{\theta_1, \ldots, \theta_k\}$ is *ϕ-equivalent* to $CAS(P, q(t'_1, \ldots, t'_n)) = \{\theta'_1, \ldots, \theta'_{k'}\}$, denoted $CAS(P, p(t_1, \ldots, t_m)) \simeq_\phi CAS(P, q(t'_1, \ldots, t'_n))$, if and only if either $k = k' = 0$ and $p(t_1, \ldots, t_m)$ is a ϕ-variant of $q(t'_1, \ldots, t'_n)$, or $1 \leq k = k'$ and $\forall i (1 \leq i \leq k) : p(t_1, \ldots, t_m)\theta_i$ is a ϕ-variant of $q(t'_1, \ldots, t'_n)\theta'_i$. This means that, according to their argument mapping, both queries generate the same procedural behavior.

In order to apprehend the notion of a semantic clone pair, let us now define the following relation between code fragments: given two code fragments F_1/m and F_2/n $(m \leq n)$ from a program P, we say that F_1 *computes a subrelation of* F_2 *with respect to an injective argument mapping* $\phi : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$, denoted $F_1 \subseteq_\phi F_2$, if and only if for each call to $rp(F_1)$, say of the form

$p(t_1, \ldots, t_m)$, there exists a call to $rp(F_2)$, say of the form $q(t'_1, \ldots, t'_n)$, such that $CAS(P, p(t_1, \ldots, t_m)) \simeq_\phi CAS(P, q(t'_1, \ldots, t'_n))$. Intuitively, $F_1 \subseteq_\phi F_2$ means that all goals provable by the relation represented by F_1 can be proved, at least in one manner, by the relation represented by F_2. Note that, on the one hand, F_1 and F_2 are not necessarily code fragments of the same granularity (a predicate definition may compute a subrelation of a goal for example) and, on the other hand, they may belong to two different predicates or to the same predicate. Besides, there may be cross-calls during the resolution of the calls to $rp(F_1)$ and $rp(F_2)$, i.e. F_1 may depend (possibly indirectly) on F_2 and vice versa.

As an illustration, consider the well-known predicate definitions of $rev1(L1, L2)$, implementing naive reverse, and of $rev2(L1, Acc, L2)$, implementing reverse with an accumulator. We may state that $rev1$ computes a subrelation of $rev2$ with respect to $\phi = \{(1,1),(2,3)\}$ since for each call to $rev1$, say of the form $rev1(L1, L2)\sigma$ for a certain substitution σ, there exists a call to $rev2$ of the form $rev2(L1, [], L2)\sigma$, such that both queries fail or result in the same computed answer substitution. Reconsidering the example from the introduction, neither $p \subseteq_{\{(1,1),(4,2)\}} q$ nor $q \subseteq_{\{(1,1),(2,4)\}} p$. But it is possible to define a third predicate, say g, such that $p \subseteq_{\phi_1} g$ and $q \subseteq_{\phi_2} g$, disclosing consequently the relationship between p and q. This is exactly the basic idea of our definition of a code clone pair.

Definition 1. *Two code fragments F_1 and F_2 form a* code clone pair *if and only if there exists a code fragment F and two injective argument mappings ϕ_1 and ϕ_2 such that $F_1 \subseteq_{\phi_1} F$ and $F_2 \subseteq_{\phi_2} F$.*

Intuitively, if F_1 and F_2 form a code clone pair, it means that there exists a code fragment that can be seen as generalizing both F_1 and F_2, that can be used to compute both the relations computed by F_1 and F_2. Back to our running example, such a possible generalization of p and q is this predicate definition:
$g(C, [], I, J, 0, 0) \leftarrow$
$g(C, [H|T], 1, 1, H, H) \leftarrow C(H)$
$g(C, [H|T], 1, J, S, P) \leftarrow C(H),\ J > 1,\ J1\ is\ J - 1,\ g(T, 1, J1, S1, P1),\ S\ is\ S1 + H,\ P\ is\ P1 * H$
$g(C, [H|T], I, J, S, P) \leftarrow C(H),\ I > 1,\ J \geq I,\ I1\ is\ I - 1,\ J1\ is\ J - 1,\ g(T, I1, J1, S, P)$

The predicate definition of g corresponds to the definition of p to which a test was added on the elements of the input list by means of the atom $C(H)$ where C represents a relation name given as argument to g. Regarding the semantics, we effectively have that g allows to compute both p and q since a generic call of the form $p(L, I, J, S, P)$ is equivalent to the call $g(test\_p, L, I, J, S, P)$ and a call $q(L, S)$ is equivalent to the call $g(test\_q, L, 1, n, S, \_)$ where n stands for the length of the list, and the predicates $test\_p$ and $test\_q$ are defined as follows:
$test\_p(X) \leftarrow$ $\qquad$ $test\_q(X) \leftarrow even(X)$
$\qquad\qquad\qquad\qquad\quad$ $test\_q(3) \leftarrow$

Returning to the definition of a code clone pair, two particular cases may be deduced. First, if $F_1 \subseteq_\phi F_2$, then F_1 and F_2 form a code clone pair, given that by definition $F_2 \subseteq_{id} F_2$ (with id representing the identical argument mapping).

The predicates $rev1$ and $rev2$ belong to this category. Secondly, if we have both $F_1 \subseteq_\phi F_2$ and $F_2 \subseteq_{\phi^{-1}} F_1$, then F_1 and F_2 are fully semantically equivalent, in the sense that every call to $rp(F_1)$ could be replaced by a call to $rp(F_2)$, and vice versa, taking into account some manipulation of the arguments. For example, this could be the case for two predicates sorting a list in increasing order – one implementing quicksort, the other implementing bubble sort – or for two predicates representing the relations "bigger than" and "smaller than".

As the examples show, the proposed notion of a code clone pair captures a larger subset of cloned code fragments than would any definition based on syntactical comparison, such as [4]. Moreover, the notion of a code clone pair is not, in general, computable, but, to the best of our knowledge, this is the first attempt of formal definition for logic programs. As we will argue in the next section, it can also be approximated by using well-known program transformations.

3 Towards an approximation of semantic clones

Even if *computing* code clone pairs in the sense outlined above is beyond the scope of the present work, a first step consists in providing an approximation of code clone pairs that is verifiable by program analysis. In what follows, we consider a generic set $\mathcal{R}$ of program transformation rules which must be *partially correct with respect to the CAS semantics*, i.e. such that given a program P and for all queries Q, $CAS(P,Q) \subseteq CAS(P',Q)$, where P' results from the application of the transformation rule on P [12]. When $CAS(P,Q) = CAS(P',Q)$, the rule is said *totally correct*, i.e. semantics-preserving [12]. Some well-known program transformation rules satisfy this condition of being totally correct: the *predicate definition introduction* (R1), *unfolding* (R2) and *folding* (more precisely, the version *in-situ folding*) (R3) rules [12]. The fourth rule that we will consider is based on the notion of *theta-subsumption*, taken from the inductive logic programming field [14]. To introduce it, we first need to present a particular notation: given two clauses C_1 and C_2, the formula $C_1\theta \subseteq C_2$ means that, modulo the substitution θ, the head of C_1 can be mapped to the head of C_2 and the body atoms of C_1 can be mapped into a subset of the body atoms of C_2. The definition of θ-subsumption is the following:

Definition 2. *A clause C_1* θ-subsumes a clause *C_2, denoted $C_1 \succeq_\theta C_2$, if and only if there exists a substitution θ such that $C_1\theta \subseteq C_2$. We call C_1 a* generalization of *C_2 (and C_2* a specialization of *C_1)* under θ-subsumption*. A clause* θ-subsumes a set of clauses *if and only if it θ-subsumes each clause in the set.*

As an approximation of logical implication (in general undecidable for first-order languages), θ-subsumption is sound ($C_1 \succeq_\theta C_2 \Rightarrow C_1 \models C_2$) but incomplete with respect to logical implication (if we exclude tautologies and self-recursive clauses, $C_1 \succeq_\theta C_2 \Leftrightarrow C_1 \models C_2$) [9]. Moreover, θ-subsumption is decidable but NP-complete [5]. An example is provided by the clause $parent(X,Y) \leftarrow mother(X,Y),\ mother(X,Z)$ which θ-subsumes the clause $parent(A,max) \leftarrow mother(A,max),\ male(max)$ with $\theta = \{X/A, Y/max, Z/max\}$.

We may now define our fourth program transformation rule (R4).

Definition 3. *This rule is divided in three subrules. Given a program P, we can transform it into a program P' by one of the following rules:*

- deletion of θ-subsumed clauses (R4.1) *: we obtain P' by deleting from P a set of clauses which is θ-subsumed by another clause in P.*
- deletion of θ-subsuming clauses (R4.2) *: we obtain P' by deleting from P a clause which θ-subsumes a set of other clauses in P.*
- introduction of θ-subsumed clauses (R4.3) *: we obtain P' by replacing in P a clause which θ-subsumes a set of other (new) clauses by the latter set.*

The rule R4.1 is inspired by a simpler form considering a single clause instead of a set of clauses [12]. This rule does not preserve CAS semantics, however a particular case of this rule, called *deletion of duplicate clauses*, does [12]: all the occurrences of a clause C in a program are replaced by a single occurrence of C. The rule R4.2 is the exact inverse of R4.1 and, with the rule R4.3, they both constitute program specialization rules under θ-subsumption. In R4.3, the creation of the θ-subsumed clauses results from the application of a *refinement operator*, which basically executes two operations on a clause: applying a substitution θ and adding one or several atoms (according to the language of the program) [9].

Finally, the last rule, called *purification* (R5), is inspired from works addressing the merging of two different predicates [2, 20, 11]. Intuitively, it is aiming at suppressing from a clause body one of its atoms which does not influence the resolution of the subgoal remaining to the right of the given atom.

Definition 4. *Let P be a logic program and $C \equiv H \leftarrow B_1, \ldots, B_s$ a clause in P. Suppose that there exists a non-failing atom B_k ($1 \leq k \leq s$) such that the new variables that it introduces (i.e. the set $vars(B_k) \setminus exvars(B_k)$) do not appear in the atoms to the right of B_k that are not recursive calls. If we* purify *C* w.r.t. *B_k, we derive the clause $C' \equiv H \leftarrow B_1, \ldots, B_{k-1}, B_{k+1}, \ldots, B_s$ and we get a program P' from P by replacing C by C'.*

For the purpose of this work, we will fill the set $\mathcal{R}$ with the five basic program transformation rules presented above (other instanciations of $\mathcal{R}$ could of course be possible), and these rules will serve to modify only a single chosen predicate of the original program. To formalize this idea, we borrow from [12] the notion of *transformation sequence*, adapted to our work: given a logic program P and a predicate p defined over the language of P, an *$\mathcal{R}$-transformation sequence related to $P \cup \{p\}$* is a finite sequence of programs $P_0 \cup \{p_0\}, \ldots, P_k \cup \{p_k\}$ such that $P_0 \cup \{p_0\} = P \cup \{p\}$ and $\forall i (0 \leq i < k) : P_{i+1} \cup \{p_{i+1}\}$ is obtained from $P_i \cup \{p_i\}$ by applying the transformation rule R1 or by applying on the clauses of p_i a transformation rule from $\mathcal{R} \setminus \{\text{R1}\}$. The notion of an $\mathcal{R}$-transformation sequence is quite related to the idea of *partial evaluation* [17, 6]. In fact, it extends the idea of "specializing" a program by applying transformation rules other than unfold.

Finally, the above notion of $\mathcal{R}$-transformation sequence in association with the notion of ϕ-variant allow us to approximate the notion of a subrelation:

Definition 5. *Given two code fragments F_1/m and F_2/n $(m \leq n)$ from a program P, we say that* F_1 computes an $\mathcal{R}$-subrelation *of* F_2 with respect to an injective argument mapping $\phi : \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$, *denoted* $F_1 \subseteq^{\mathcal{R}}_{\phi} F_2$, *if and only if there exists an $\mathcal{R}$-transformation sequence of the form $P_0 \cup \{rp(F_2)_0\}, \ldots, P_k \cup \{rp(F_2)_k\}$, such that $rp(F_1)$ is a ϕ-variant of $rp(F_2)_k$.*

At the level of code fragments, demonstrating $F_1 \subseteq^{\mathcal{R}}_{\phi} F_2$ boils thus down to deriving from the predicate definition relative to the more general code fragment F_2 a new predicate of which the predicate definition relative to the more specific code fragment F_1 is a ϕ-variant. If F_1 computes an $\mathcal{R}$-subrelation of F_2, then F_1 computes a subrelation of F_2, but the converse is not necessarily true. This is for example the case for the predicates $rev1$ and $rev2$: $rev1$ is not an $\mathcal{R}$-subrelation of $rev2$ because the way of computing the reverse list is inherently different in both predicate definitions, they implement two different algorithms which can possibly not be derived from one another by the transformation rules from $\mathcal{R}$.[1]

Example 1. Returning to our running example, by rule R1, we may introduce the definition of $test\_p/1$. By rule R4.3 and $\theta = \{C/test\_p\}$, we obtain:
$g_1(test\_p, [], I, J, 0, 0) \leftarrow$
$g_1(test\_p, [H|T], 1, 1, H, H) \leftarrow test\_p(H)$
$g_1(test\_p, [H|T], 1, J, S, P) \leftarrow test\_p(H),\ J > 1,\ J1\ is\ J-1,\ g_1(test\_p, T, 1, J1, S1, P1),\ S\ is\ S1+H,\ P\ is\ P1 * H$
$g_1(test\_p, [H|T], I, J, S, P) \leftarrow test\_p(H),\ I > 1,\ J \geq I,\ I1\ is\ I-1,\ J1\ is\ J-1,\ g_1(test\_p, T, I1, J1, S, P)$

Then, by R2 w.r.t. the atom $test\_p(H)$, we obtain:
$g_2(test\_p, [], I, J, 0, 0) \leftarrow$
$g_2(test\_p, [H|T], 1, 1, H, H) \leftarrow$
$g_2(test\_p, [H|T], 1, J, S, P) \leftarrow J > 1,\ J1\ is\ J-1,\ g_2(test\_p, T, 1, J1, S1, P1),\ S\ is\ S1+H,\ P\ is\ P1 * H$
$g_2(test\_p, [H|T], I, J, S, P) \leftarrow I > 1,\ J \geq I,\ I1\ is\ I-1,\ J1\ is\ J-1,\ g_2(test\_p, T, I1, J1, S, P)$

We have that p is a $\{(1,2),(2,3),(3,4),(4,5),(5,6)\}$-variant of the above predicate, thus an $\mathcal{R}$-subrelation of g, thus a subrelation of g. Indeed, to each call to p corresponds a call to g, instance of one of the clause heads of g_2.

Example 2. Considering the same example, by R4.3 and $\theta = \{I/1\}$, we obtain:
$g_0(C, [], 1, J, 0, 0) \leftarrow$
$g_0(C, [H|T], 1, 1, H, H) \leftarrow C(H)$
$g_0(C, [H|T], 1, J, S, P) \leftarrow C(H),\ J > 1,\ J1\ is\ J-1,\ g_0(T, 1, J1, S1, P1),\ S\ is\ S1+H,\ P\ is\ P1 * H$
$g_0(C, [H|T], 1, J, S, P) \leftarrow C(H),\ 1 > 1,\ J \geq 1,\ I1\ is\ 1-1,\ J1\ is\ J-1,\ g_0(T, I1, J1, S, P)$

By R3 w.r.t. the atom $1 > 1$, we delete the last clause. By R5, we successively delete from the third clause the atoms $P\ is\ P1 * H$, $J1\ is\ J-1$ and $J > 1$:

[1] In the case of $rev1$ and $rev2$ the two *could* be proven equivalent but this requires using properties other than the rules in $\mathcal{R}$ [19, 13].

$g_2(C, [], 1, J, 0, 0) \leftarrow$
$g_2(C, [H|T], 1, 1, H, H) \leftarrow C(H)$
$g_2(C, [H|T], 1, J, S, P) \leftarrow C(H),\ g_2(T, 1, J1, S1, P1),\ S\ is\ S1 + H$

By R4.3 and the empty substitution, we may replace the second clause by $g(C, [H], 1, 1, H, H) \leftarrow C(H),\ g([], 1, 0, 0, 0),\ H\ is\ 0 + H$. This new clause is θ-subsumed by the third clause and may thus be deleted by R4.1. Only the first and third clauses remain at this point. Next, by rule R1, we introduce the definition of $test\_q/1$. Finally, by R4.3 and $\theta = \{C/test\_q\}$, then by rule R.2 w.r.t. the atom $test\_q(H)$, we obtain:
$g_7(test\_q, [], 1, J, 0, 0) \leftarrow$
$g_7(test\_q, [H|T], 1, J, S, P) \leftarrow even(H),\ g_7(test\_q, T, 1, J1, S1, P1),\ S\ is\ S1 + H$
$g_7(test\_q, [3|T], 1, J, S, P) \leftarrow g_7(test\_q, T, 1, J1, S1, P1),\ S\ is\ S1 + 3$

The predicate q is effectively a $\{(1, 2), (2, 5)\}$-variant of this above predicate, thus computes an ($\mathcal{R}$-)subrelation of g.

We can now also approximate the notion of a code clone pair as follows:

Definition 6. *Two code fragments F_1 and F_2 form an* $\mathcal{R}$-clone pair *if and only if there exists a code fragment F and two injective argument mappings ϕ_1 and ϕ_2 such that $F_1 \subseteq^{\mathcal{R}}_{\phi_1} F$ and $F_2 \subseteq^{\mathcal{R}}_{\phi_2} F$.*

Example 3. Reconsidering our running examples, we can conclude from the proofs in examples 1 and 2 that the predicates p and q form an $\mathcal{R}$-clone pair.

4 Relevance and ongoing work

Let us first examine the relevance of our definition of an $\mathcal{R}$-clone pair in the context of refactorings that aim at removing duplicated (cloned) code from a program. Some basic refactorings exist in the literature of logic programming. The "deletion of θ-subsumed clauses" and "deletion of duplicate clauses" rules are two of them [12]. In the same work [12], another program transformation rule, "deletion of duplicate goals", allows to replace a goal (G, G) in the body of a clause by the goal G. Serebrenik et al. [18] present two refactorings for Prolog programs. The first one aims at identifying identical subsequences of goals in different predicate bodies, in order to extract them into a new predicate. The detection phase is said to be comparable to the problem of determining longest common subsequences. The second refactoring aims at eliminating copy-pasted predicates, with identical definitions. However, to limit the search complexity, only predicates with identical names in different modules are considered duplicated. Vanhoof and Degrave [21, 22] expose the idea of a more complex refactoring: generalization of two predicate definitions into a higher-order one. Those refactorings are frequently studied for other programming languages [15], and we may point out in particular another kind of generalization realized thanks to a merging operator [2].

Our definition of an $\mathcal{R}$-clone pair allows to embody all those refactorings in a single simple schema. Given two code fragments F_1 and F_2 from a program P that form an $\mathcal{R}$-clone pair via the code fragment F, we could refactor them using the following procedure:

- add $rp(F)$ to the program P as a new predicate
- replace F_1 by an equivalent call to $rp(F)$ of this manner, assuming that $F_1 \subseteq^{\mathcal{R}}_{\phi} F$ and the predicate name of $rp(F)$ is g/k: if F_1 is a goal, then it simply becomes the call to $rp(F)$; if F_1 is a sequence of clauses or a predicate definition of name p/m, then it is replaced by a new clause $p(t_1, \ldots, t_m) \leftarrow g(t'_1, \ldots, t'_k)$ where $\forall i(1 \leq i \leq m) : t_i = t'_{\phi(i)}$.
- replace F_2 by an equivalent call to $rp(F)$ as explained just above.

Example 4. Returning to our running example, the predicates p and q defined in the introduction may be refactored by adding the definitions of $g/6$, $test\_p/1$ and $test\_q/1$ (see Section 2) to the program and replacing the definition of p by the single clause $p(L, I, J, S, P) \leftarrow g(test\_p, L, I, J, S, P)$ and the definition of q by $q(L, S) \leftarrow g(test\_q, L, 1, MAXINT, S, \_)$.

As for some concluding remarks, note that the definition of an $\mathcal{R}$-clone pair generalizes the definition of a purely syntactically structural clone as the one in our previous work [4] by considering ϕ-variance with respect to an $\mathcal{R}$-transformation sequence. Indeed, a purely syntactical clone pair between code fragments F_1 and F_2 corresponds to an $\mathcal{R}$-clone pair where the code fragment F is either F_1 or F_2 and where no transformation rule has to be applied.

The price to pay for the generality of the definition relies of course in the complexity needed for verifying whether two code fragments are related by an $\mathcal{R}$-subrelation. Also note that even if the verification could be completely automated, doing so might turn out to be far from trivial since deciding which transformation rule to apply and how to parametrize it may reveal to be quite complex. An interesting topic of ongoing work is how to *find* a generalization of two code fragments suspected of forming an $\mathcal{R}$-clone pair. Developing an algorithm for detecting (a relevant subset) of $\mathcal{R}$-clone pairs is an important step in identifying semantical clones [16].

5 Aknowledgements

We warmly thank the anonymous reviewers which helped us to improve our work and our paper by their thought-provoking and enriching remarks.

References

1. Apt, K.: Logic programming. In: Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics, pp. 493–574. Elsevier (1990)
2. Brown, C., Thompson, S.: Clone detection and elimination for Haskell. In: Proceedings of the 2010 SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'10). pp. 111–120. ACM (2010)

3. Chen, W., Kifer, M., Warren, D.: HiLog: A foundation for higher-order logic programming. Journal of Logic Programming 15(3), 187–230 (1993)
4. Dandois, C., Vanhoof, W.: Clones in logic programs and how to detect them. In: Proceedings of the 22st International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2011) (2011)
5. Kapur, D., Narendran, P.: NP-completeness of the set unification and matching problems. In: Proceedings of the 8th International Conference on Automated Deduction. pp. 489–495 (1986)
6. Leuschel, M.: Advanced Techniques for Logic Program Specialisation. Ph.D. thesis, Katholieke Universiteit Leuven (1997)
7. Li, H., Thompson, S.: Clone detection and removal for Erlang/OTP within a refactoring environment. In: Proceedings of the 2009 SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'09). pp. 169–178. ACM (2009)
8. Lloyd, J.W.: Foundations of Logic Programming (Second Edition). Springer-Verlag (1987)
9. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. Journal Of Logic Programming 19(20), 629–679 (1994)
10. Nadathur, G., Miller, D.: Higher-order logic programming. In: Handbook of logic in Artificial Intelligence and logic programming. vol. 5, pp. 499–590. Oxford University Press (1998)
11. Pettorossi, A., Proietti, M.: Transformation of logic programs: Foundations and techniques. The Journal of Logic Programming 19-20, 261–320 (1994)
12. Pettorossi, A., Proietti, M.: Transformation of logic programs. In: Handbook of Logic in Artificial Intelligence and Logic Programming, vol. 5, pp. 697–787. Oxford University Press (1998)
13. Pettorossi, A., P.M.S.V.: Constraint-based correctness proofs for logic program transformations. Tech. Rep. 24, IASI-CNR (2011)
14. Plotkin, G.D.: Automatic Methods of Inductive Inference. Ph.D. thesis, Edinburgh University (1971)
15. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Tech. rep. (2007)
16. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Science of Computer Programming 74(7), 470–495 (2009)
17. Salin, D.: Mixtus: An automatic partial evaluator for full Prolog. New Generation Computing 12, 7–15 (1993)
18. Serebrenik, A., Schrijvers, T., Demoen, B.: Improving Prolog programs: Refactoring for Prolog. Theory and Practice of Logic Programming (TPLP) 8, 201–215 (2008), other version consulted: `https://lirias.kuleuven.be/bitstream/123456789/164765/1/technical_note.pdf`
19. Seres, S., Spivey, J.M.: Higher-order transformation of logic programs. In: Selected Papers form the 10th International Workshop on Logic Based Program Synthesis and Transformation (LOPSTR'00). pp. 57–68. Springer-Verlag (2001)
20. Tamaki, H., Sato, T.: Unfold/fold transformations of logic programs. In: Proceedings of the 2nd International Conference on Logic Programming (ICLP84) (1984)
21. Vanhoof, W.: Searching semantically equivalent code fragments in logic programs. In: Proceedings of the 14th International Symposium on Logic based Program Synthesis and Transformation (LOPSTR'04). pp. 1–18. Springer-Verlag (2004)
22. Vanhoof, W., Degrave, F.: An algorithm for sophisticated code matching in logic programs. In: Proceedings of the 24th International Conference on Logic Programming (ICLP'08). pp. 785–789. Springer-Verlag (2008)

Branching Preserving Specialization for Software Model Checking

Emanuele De Angelis[1], Fabio Fioravanti[1],
Alberto Pettorossi[2], and Maurizio Proietti[3]

[1] University 'G. D'Annunzio',
Viale Pindaro 42, I-65127 Pescara, Italy
{deangelis,fioravanti}@sci.unich.it
[2] DISP, University of Rome Tor Vergata, Rome, Italy
pettorossi@disp.uniroma2.it
[3] IASI-CNR, Viale Manzoni 30, I-00185 Rome, Italy
maurizio.proietti@iasi.cnr.it

Abstract. We present a method for verifying properties of imperative programs by using techniques based on constraint logic programming (CLP). We consider a simple imperative language, called SIMP, extended with a nondeterministic choice operator and we address the problem of checking whether or not a *safety* property φ (that specifies that an *unsafe* configuration cannot be reached) holds for a SIMP program P. The operational semantics of the language SIMP is specified via an interpreter I written as a CLP program. The first phase of our verification method consists in specializing I with respect to P, thereby deriving a specialized interpreter I_P. Then, we specialize I_P with respect to the property φ and the input values of P, with the aim of deriving a program whose least model can be computed as a finite set of constrained facts. To this purpose we introduce a novel generalization strategy which, during specialization, preserves the so called branching behaviour of the predicate definitions. We have fully automated our method and we have made its experimental evaluation on some examples taken from the literature. The evaluation shows that our method is competitive with respect to state-of-the-art software model checkers.

1 Introduction

Software model checking is a body of formal verification techniques for imperative programs that combine and extend ideas and techniques developed in the fields of static program analysis and model checking (see [19] for a recent survey).

In this paper we consider a simple imperative language SIMP acting on integer variables, with nondeterministic choice, assignment, conditional, and while-do commands (see, for instance, [29]) and we address the problem of verifying *safety* properties. Basically, a safety property states that when executing a program, an unsafe configuration cannot be reached from any initial configuration. Since we consider programs that act on integer numbers, the problem of deciding whether or not an unsafe configuration is reachable is undecidable.

In order to cope with this undecidability limitation, many program analysis techniques follow approaches based on *abstraction* [4], by which the concrete data domain is mapped to an abstract domain so that reachability is preserved, that is, if a concrete configuration is reachable, then the corresponding abstract configuration is reachable. By a suitable choice of the abstract domain one can design reachability algorithms that terminate and, whenever they prove that an abstract unsafe configuration is not reachable from an abstract initial configuration, then the program is proved to be safe (see [19] for a general abstract reachability algorithm). Notable abstractions are those based on convex polyhedra, that is, conjunctions of linear inequalities (also called *constraints* here).

Due to the use of abstraction, the reachability of an abstract unsafe configuration does not necessarily imply that the program is indeed unsafe. It may happen that the abstract reachability algorithm produces a *spurious counterexample*, that is, a sequence of configurations leading to an abstract unsafe configuration which does not correspond to any concrete computation. When a spurious counterexample is found, *counterexample-guided abstraction refinement* (CEGAR) automatically refines the abstract domain so that a new run of the abstract reachability algorithm rules out the counterexample [1,3,30]. Clearly, the CEGAR technique may not terminate because an infinite number of spurious counterexamples may be found. Several techniques have been proposed to improve the termination behaviour of that technique by introducing more sophisticated refinement strategies (see, for instance, [14,16,20,32])

In order to improve the termination of the safety verification process, we propose in this paper a technique based on the *specialization of constraint logic programs*. Constraint Logic Programming (CLP) has been shown to be very suitable for the analysis of imperative programs, because it provides a very convenient way of representing symbolic program executions and also, by using constraints, program invariants (see, for instance, [16,18,27,28]). Program specialization is a program transformation technique which, given a program P and a portion in_1 of its input data, returns a specialized program P_s that is equivalent to P in the sense that when the remaining portion in_2 of the input of P is given, then $P_s(in_2) = P(in_1, in_2)$ [12,21,22]. The specialization of CLP programs has been proposed in [27] as a pre-processing phase for program analysis. This analysis is done in various steps. First, the semantics of an imperative language is defined by means of a CLP program which is the interpreter I of that language, and then, program I is specialized with respect to the program P whose safety property should be checked. The result of this specialization is a CLP program I_P and, since program specialization preserves semantic equivalence, we can analyze I_P for proving the properties of P.

Similarly to [27], also the technique proposed in this paper produces a specialized interpreter I_P. However, instead of applying program analysis techniques, we further specialize I_P with respect to the property characterizing the input values of P (that is, a precondition of P), thereby deriving a new program I'_P. The effect of program specialization is the modification of the structure of the program and the explicit addition of new constraints that denote invariants of

the computation. Through various experiments we show that by exploiting this additional knowledge, the construction of the least model of the program of I'_P terminates in many interesting cases and, thus, it is possible to verify safety properties by simply inspecting that model.

An essential ingredient of program specialization is the *generalization*, which introduces new predicate definitions representing invariants of the program P. Generalization can be used to enforce the termination of program specialization (which occurs when no new predicate definitions are generated) and, in this respect, is similar to the widening operators used in static program analysis [5]. One problem encountered with generalization strategies is that sometimes they introduce predicate definitions which are too general, thereby making specialization useless. In this paper we introduce a new generalization strategy, called the *branching preserving generalization*, whose objective is indeed to avoid the introduction of predicate definitions that are too general.

The basic idea is as follows. Given a sequence of unfolding steps performed during program specialization, we may consider a symbolic evaluation tree made out of clauses, such that every clause has as children the clauses which are generated from that clause by unfolding. If a given clause γ has n children which are generated by unfolding using clauses $\gamma_1, \ldots, \gamma_n$, and clause γ is generalized to a new predicate definition δ, then it should be the case that by unfolding δ, we get again n children and these children are due to the same clauses $\gamma_1, \ldots, \gamma_n$.

Since this generalization preserves the branching structure of the symbolic evaluation tree, it is said to be branching preserving. We will see that branching preservation can be realized by adding suitable constraints that avoid the generation of extra branches.

The paper is organized as follows. In Section 2 we describe the syntax of the SIMP language and the CLP interpreter which defines its operational semantics. In Section 3 we outline our software model checking approach by developing an example taken from [14]. In Sections 4 and 5 we describe our strategy for specializing CLP programs and, in particular, the branching preserving generalization technique. In Section 6 we report on some experiments we have performed by using a prototype implementation based on the MAP transformation system [26]. We also compare the results we have obtained using the MAP system with the results we have obtained using state-of-the-art software model checking systems such as ARMC [28], HSF(C) [13], and TRACER [17]. Finally, in Section 7 we discuss the related work and, in particular, we compare our approach with other existing methods of software model checking.

2 A CLP Interpreter for a Simple Imperative Language

The syntax of our language SIMP, a C-like imperative language, is defined by using: (i) the set *Int* of integers, ranged over by n, (ii) the set {`true`, `false`} of booleans, and (iii) the set *Loc* of locations, ranged over by x. We have also the following derived sets: (iv) *Aexpr* of arithmetic expressions, (v) *Bexpr* of boolean expressions, (vi) *Test* of tests, and (vii) *Com* of commands. The syntax of our language is as follows.

Aexpr $\ni a ::= n \mid x \mid a_0\ aop\ a_1$
Bexpr $\ni b ::=$ `true` $\mid$ `false` $\mid a_0\ rop\ a_1 \mid$ `!` $b \mid b_0\ bop\ b_1$
Test $\ni t ::=$ `nd` $\mid b$
Com $\ni c ::=$ `skip` $\mid x$ `=` $a \mid c_0$ `;` $c_1 \mid$ `if (` t `) {` c_0 `} else` $c_1 \mid$ `while (` t `) {` c `} \mid` `error`

where the arithmetic operator *aop* belongs to {`+`, `-`, `*`}, the relational operator *rop* belongs to {`<`, `<=`, `==`}, and the boolean operator *bop* belongs to {`&&`, `||`}. The constant `nd` denotes the nondeterministic choice and `error` denotes the error command. The other symbols should be understood as usual in C. We will write `if (`t`) {`c_0`}`, instead of `if (`t`) {`c_0`} else skip`.

Now we introduce a CLP program which defines the interpreter of our SIMP language. We need the following notions.

A *state* is a function from *Loc* to *Int*. It is denoted by a list of CLP terms, each of which is of the form `bn(loc(X),V)`, where `bn` is a binary constructor binding the location `X` to the value of the CLP variable `V`. We assume that the set of locations used in every command is fixed and, thus, for every command, the state has a fixed, finite length. We have two predicates operating on states: (i) `lookup(loc(X),S,V)`, which holds iff the location `X` stores the value `V` in the state `S`, and (ii) `update(loc(X),V,S1,S2)`, which holds iff the state `S2` is equal to the state `S1`, except that the location `X` stores the value `V`.

We also have the predicates `aev(A,S,V)` and `bev(B,S)`, for the evaluation of arithmetic expressions and boolean expressions, respectively. `aev(A,S,V)` holds iff the arithmetic expression `A` in the state `S` evaluates to `V`, and `bev(B,S)` holds iff the boolean expression `B` holds in the state `S`. A test `T` in a state `S` is evaluated via the predicate `tev(T,S)` defined as follows: (i) for all states `S`, *both* `tev(nd,S)` and `tev(not(nd),S)` hold, and (ii) for all boolean expressions `B`, `tev(B,S)` holds iff `bev(B,S)` holds.

A command c is denoted by a term built out of the following constructors: `skip` (nullary), `asgn` (binary) for assignment, `comp` (binary) for command composition, `ite` (ternary) for conditional, and `while` (binary) for while-do. The operator '`;`' associates to the right. Thus, for instance, the command c_0`;`c_1`;`c_2 is denoted by the term `comp(c0,comp(c1,c2))`.

A *configuration* is a pair of a command and a state. A configuration is denoted by the term `cf(c,s)`, where `cf` is a binary constructor which takes as arguments the command `c` and the state `s`. The interpreter of our SIMP language, adapted from [29], is defined in terms of a *transition relation* that relates an old configuration to either a new configuration or a new state. That relation is denoted by the predicate `tr` whose clauses are given below. `tr(cf(C,S),cf(C1,S1))` holds iff the execution of the command `C` in the state `S` leads to the new configuration `cf(C1,S1)`, and `tr(cf(C,S),S1)` holds iff the execution of the command `C` in the state `S` leads to the new state `S1`.

```
tr(cf(skip,S), S).
tr(cf(asgn(loc(X),A),S),S1) :- aev(A,S,V), update(loc(X),V,S,S1).
tr(cf(comp(C0,C1),S), cf(C1,S1)) :- tr(cf(C0,S),S1).
tr(cf(comp(C0,C1),S), cf(comp(C0',C1),S')) :- tr(cf(C0,S), cf(C0',S')).
tr(cf(ite(T,C0,_),S), cf(C0,S)) :- tev(T,S).
tr(cf(ite(T,_,C1),S), cf(C1,S)) :- tev(not(T),S).
tr(cf(while(T,C),S), cf(ite(T,comp(C,while(T,C)),skip),S)).
```

A state `s` is said to be *initial* if *initProp* holds in `s`. A configuration is said to be *initial* if its state is initial. A configuration is said to be *unsafe* if its command is `error`.

Now, we introduce a CLP program, called R, that by using a bottom-up evaluation strategy, performs in a backward way the reachability analysis over configurations. Program R checks whether or not an unsafe configuration is reachable from an initial configuration, by starting from the unsafe configurations. The semantics of program R is given by its least model, denoted $M(R)$.

Definition 1 (Reachability Program). *Given a boolean expression initProp holding in the initial states and a command com, the reachability program R is made out of the following clauses:*

```
unsafe :- initConf(X), reachable(X).
reachable(X) :- unsafeConf(X).          % unsafe configurations are reachable
reachable(X) :- tr(X,X1), reachable(X1).
initConf(cf(com,S)) :- bev(initProp,S).% initProp holds in the initial state S
unsafeConf(cf(error,S)).  % the error command defines an unsafe configuration
```

together with the clauses for the predicates `tr` *and* `bev` *and the predicates they depend upon. In the above clauses for R the terms* `initProp` *and* `com` *denote initProp and com, respectively. We will say that com is safe with respect to initProp (or com is safe, for short) iff* `unsafe` $\notin M(R)$.

3 Specialization-Based Software Model Checking

In this section we outline the method for software model checking we propose. By means of an example borrowed from [14], we argue that program specialization can prove program safety in some cases where the CEGAR method (as implemented in ARMC [28]) does not work. Let *initProp* be: `x==0 && y==0 && n>=0` and the SIMP command *com* be:

```
while (x<n) { x = x+1; y = y+1 };
while (x>0) { x = x-1; y = y-1 };
if (y>x) error
```

We want to prove that *com* is safe with respect *initProp*, that is, there is no execution of *com* with input values of `x`, `y`, and `n` satisfying *initProp*, such that the `error` command is executed. CEGAR fails to prove this safety property, because an infinite set of counterexamples is generated.

By applying the Specialization-Based Software Model Checking method we propose in this paper, we will be able to prove that *com* is indeed safe. As indicated in Section 2, we have to show that `unsafe` $\notin M(\mathrm{R})$, where R is the CLP program of Definition 1, `com` is the term:

```
comp(while(lt(loc(x),loc(n)),
  comp(asgn(loc(x),plus(loc(x),1)), asgn(loc(y),plus(loc(y),1)))),
comp(while(gt(loc(x),0),
  comp(asgn(loc(x),minus(loc(x),1)), asgn(loc(y),minus(loc(y),1)))),
ite(gt(loc(y),loc(x)),error,skip)))
```

and `initProp` is the term:

```
and(eq(loc(x),0), and(eq(loc(y),0), ge(loc(n),0)))
```

Our method consists of the three phases as specified below.

The Software Model Checking Method

Input: A boolean expression *initProp* characterizing the initial states and a SIMP command *com*.

Output: The answer *safe* iff *com* is safe.

Let R be the CLP program of Definition 1 defining the predicate `unsafe`.

(Phase 1) $\mathit{InterpreterSpecialize}(R, R_{\mathtt{com}})$;

(Phase 2) $\mathit{Specialize}(R_{\mathtt{com}}, R_{Sp})$;

(Phase 3) $\mathit{BottomUp}(R_{Sp}, M_{Sp})$;

Return the answer *safe* iff $\mathtt{unsafe} \notin M_{Sp}$.

During Phase (1), by making use of familiar transformation rules (definition introduction, unfolding, folding, removal of clauses with unsatisfiable body, and removal of subsumed clauses [7]), we compile away, similarly to [27], the SIMP interpreter by specializing program R with respect to `com`, thereby deriving the following program $R_{\mathtt{com}}$ which encodes the reachability relation associated with the interpreter specialized with respect to `com`:

```
1. unsafe :- X=1, Y=1,    N>=1,   new1(X,Y,N).
2. new1(X,Y,N) :- Y'=Y+1, X'=X+1, N>=1+X, new1(X',Y',N).
3. new1(X,Y,N) :- Y'=Y-1, X'=X-1, X>=1,   N=<X, new2(X',Y',N).
4. new1(X,Y,N) :- Y>=X+1, X=<0,   N=<X.
5. new2(X,Y,N) :- X>=1,   X'=X-1, Y'=Y-1, new2(X',Y',N).
6. new2(X,Y,N) :- X=<0,   Y>=1+X.
```

Note that: (i) the two predicates `new1` and `new2` correspond to the two while-do commands occurring in *com*, and (ii) the assignments and the conditional occurring in *com*, do not occur in $R_{\mathtt{com}}$ because by unfolding they have been replaced by suitable constraints relating the old values of `X` and `Y` to the new values of `X'` and `Y'`.

Unfortunately, the program $R_{\mathtt{com}}$ is not satisfactory for showing safety, because the bottom-up construction of the least model $M(R_{\mathtt{com}})$ does not terminate. The top-down evaluation of the `unsafe` query in $R_{\mathtt{com}}$ does not terminate either. Then, in Phase (2) we specialize program $R_{\mathtt{com}}$ by taking into account the property *initProp*, thereby deriving the specialized program R_{Sp}. During this Phase (2) the constraints occurring in the definitions of `new1` and `new2` are generalized according to a suitable generalization strategy based both on widening [5,8,11] and on the novel branching preserving generalization strategy we propose in this paper. Suitable new predicate definitions will be introduced during this Phase (2), so that at Phase (3) we can construct the least model M_{Sp} of the derived program R_{Sp} by using a bottom-up evaluation procedure. We will show that, in our example, the construction of the least model M_{Sp} terminates and we can prove the safety of the command *com* by showing that the atom `unsafe` does not belong to that model.

Phase (2) of our method makes use of the same transformation rules used during Phase (1), but those rules are applied according to a different strategy, whose effect is the propagation of the constraints occurring in $R_{\mathtt{com}}$. We start off by folding clause 1 using the following definition:

7. `new3(X,Y,N) :- X=1, Y=1, N>=1, new1(X,Y,N).`

and we get the folded clause:

1.f `unsafe:- X=1, Y=1, N>=1, new3(X,Y,N).`

We proceed by following the usual unfold-definition-fold cycle performed by specialization strategies [8,11]. Each new definition introduced during specialization determines a new node of a tree, called *DefsTree*, whose root is clause 7, which is the first definition we have introduced. (We will explain below how this tree is incrementally constructed.)

Then, we unfold clause 7 and we get:

7.1 `new3(X,Y,N) :- X=1, Y=1, N>=2, X'=2, Y'=2, new1(X',Y',N).`

7.2 `new3(X,Y,N) :- X=1, Y=1, N=1,  X'=0, Y'=0, new2(X',Y',N).`

Now, we should fold these two clauses. In order to fold clause 7.1 we first consider a definition, called the *candidate definition*, which is of the form:

8. `new4(X,Y,N) :- N>=2, X=2, Y=2, new1(X,Y,N).`

The body of the candidate definition is obtained by projecting the constraint in clause 7.1 with respect to `X'`, `Y'`, and `N`, and renaming the primed variables to unprimed variables. Since in *DefsTree* there is *an ancestor definition*, namely the root clause 7, with `new1` in the body, we apply *widening* to clause 7 and clause 8, and we get the definition:

9. `new4(X,Y,N) :- X>=1, Y>=1, N>=1, new1(X,Y,N).`

(Recall that the widening operation of two clauses $c1$ and $c2$, after replacing every equality `A=B` by the equivalent conjunction `A>=B, A=<B`, keeps the atomic constraints of clause $c1$ which are implied by the constraint of clause $c2$.) However, we do not introduce clause 9, because our *branching preserving generalization* imposes the addition of some extra constraints to the body of that clause as we now explain.

With each predicate `newk` we associate a set of constraints, called the *regions for* `newk`, which are all the atomic constraints on unprimed variables occurring in any one of the clauses for `newk` in program $R_{\mathtt{com}}$. Then, we add to the body of the candidate definition `newp(...) :- c, newk(...)` all *negated regions for* `newk` which are implied by `c`. These extra constraints ensure that there is a one-to-one correspondence between the set of clauses obtained by unfolding the candidate definition and the set of clauses obtained by unfolding the generalized definition and, in this sense, the unfolding branching is preserved by generalization.

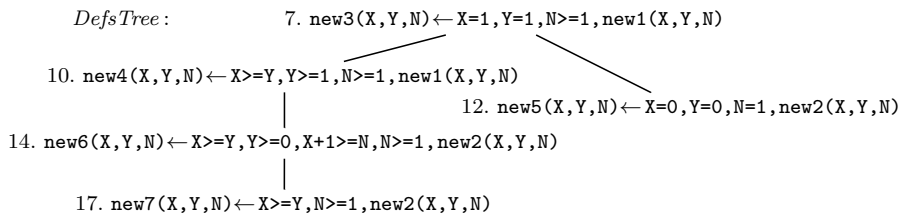
In our example, the regions for `new1` are: `Y>X`, `X=<0`, `N=<X`, `X>=1`, `N>X` and the negated regions are: `Y=<X`, `X>0`, `N>X`, `X<1`, `N=<X`. (We did some simplifications and, in particolar, we wrote `Y>X`, instead of `Y>=X+1`.) The only negated region implied by the constraint `N>=2, X=2, Y=2` occurring in the body of the candidate clause 8, is: `X>=Y`.

Thus, instead of clause 9, we introduce the following clause:

10. `new4(X,Y,N) :- X>=Y, Y>=1, N>=1, new1(X,Y,N).`

By folding clause 7.1 using clause 10 we get:
11. `new3(X,Y,N) :- X=1, Y=1, N>=2, X'=2, Y'=2, new4(X',Y',N).`
Now, it remains to fold clause 7.2 and in order to do so, we need the following candidate definition:
12. `new5(X,Y,N) :- X=0, Y=0, N=1, new2(X,Y,N).`
Clause 12 is placed in *DefsTree* as a child of clause 7, as clause 7.2 has been derived by unfolding from clause 7. We do not make any generalization of this clause, because no definition with `new2` in its body occurs as an ancestor of clause 12 in *DefsTree*. By folding clause 7.2 we get:
13. `new3(X,Y,N) :- X=1, Y=1, N=1, X'=0, Y'=0, new5(X',Y',N).`
Now, starting from the definition clause 10 and the definition clause 12, we perform some sequences of unfolding-definition-folding steps similar to that we have described above when deriving clauses 11 and 13 starting from clause 7 and, in particular, we introduce two new predicates `new6` and `new7`.

The final tree *DefsTree* of definitions introduced during specialization is as follows:



Eventually, we get the following program R_{Sp}:
1.f `unsafe :- X=1, Y=1, N>=1, new3(X,Y,N).`
11. `new3(X,Y,N):-X=1, Y=1, N>=2, X'=2, Y'=2, new4(X',Y',N).`
15. `new4(X,Y,N):-N>=X', X'>=Y', X+1=X', Y+1=Y', Y'>=2, new4(X',Y',N).`
16. `new4(X,Y,N):-N>=1,N=<1+X',X'>=Y',X-1=X', Y-1=Y', Y'>=0, new6(X',Y',N).`
18. `new6(X,Y,N):-Y-1=Y',Y'>=-1,Y'=<X',N>=1,X-1=X',X'>=0,N=<2+X',new7(X',Y',N).`
19. `new7(X,Y,N):-Y-1=Y', X-1=X', X'>=0, Y'=<X', N>=1, new7(X',Y',N).`
(Clause 13 has been removed by unfolding, because for `new5` we get an empty set of clauses.) This concludes Phase (2).

Now, we can perform Phase (3) of our method. This phase terminates immediately because there are no constrained facts in R_{Sp} and $M(R_{Sp})$ is the empty set. Thus, `unsafe` $\notin M(R_{Sp})$ and we conclude that the command *com* is safe.

One can verify that if we were to do the generalization steps of Phase (2) using the widening technique alone (without the branching preserving generalization), we could not derive a program that allows us to prove safety.

4 The Specialization Strategy

Phases (1) and (2) of the Software Model Checking Method outlined in Section 3 are realized by two applications of a general specialization strategy for CLP programs. This strategy is an adaptation of the specialization strategies presented

in [8,11]. As already mentioned in Section 3, our specialization strategy makes use of the following transformation rules: definition introduction, unfolding, clause removal, and folding which, under suitable conditions, guarantee that the least model semantics is preserved (see, for instance, [7]).

The specialization strategy is realized by the *Specialize* procedure that we now present. Initially, this procedure takes a clause of the form:

γ_0: $unsafe \leftarrow c, G$

where c is a constraint and G is a goal, and then iteratively applies the following two procedures: (i) the *Unfold* procedure, which uses the unfolding rule and the clause removal rule, and (ii) the *Generalize&Fold* procedure, which uses the definition introduction rule and the folding rule.

Procedure *Specialize*
Input: A CLP program of the form $P \cup \{\gamma_0\}$.
Output: A CLP program P_S such that $unsafe \in M(P \cup \{\gamma_0\})$ iff $unsafe \in M(P_S)$.
$P_S := \{\gamma_0\}$; $InDefs := \{\gamma_0\}$; $Defs := \emptyset$;
while there exists a clause γ in $InDefs$
do $Unfold(\gamma, \Gamma)$;
$Generalize\&Fold(Defs, \Gamma, NewDefs, \Phi)$;
$P_S := P_S \cup \Phi$; $InDefs := (InDefs - \{\gamma\}) \cup NewDefs$; $Defs := Defs \cup NewDefs$;
end-while

The *Unfold* procedure takes as input a clause γ and returns as output a set Γ of clauses derived from γ by one or more applications of the unfolding rule, which consists in replacing an atom A occurring in the body of a clause by the bodies of the clauses in P whose head is unifiable with A. The first step of the *Unfold* procedure consists in unfolding γ with respect to the leftmost atom in its body. In order to guarantee the termination of the *Unfold* procedure, an atom A is selected for unfolding only if it has not been derived by unfolding a variant of A itself. More sophisticated unfolding strategies can be applied (see [22] for a survey of techniques for controlling unfolding), but our simple strategy turns out to be effective in all our examples. At the end of the *Unfold* procedure, subsumed clauses and clauses with unsatisfiable constraints are removed.

The *Generalize&Fold* procedure takes as input the set Γ of clauses produced by the *Unfold* procedure and introduces a set *NewDefs* of *definitions*, that is, clauses of the form $newp(X) \leftarrow d(X), A(X)$, where *newp* is a new predicate symbol, X is a tuple of variables, $d(X)$ is a constraint whose variables are among the ones in X, and $A(X)$ is an atom whose variables are exactly those of the tuple X. Any such definitions denotes a set of states X satisfying the constraint $d(X)$. By folding the clauses in Γ using the definitions in *NewDefs* and the definitions introduced during previous iterations of the specialization procedure, the *Generalize&Fold* procedure derives a new set of specialized clauses. In particular, a clause of the form:

$newq(X) \leftarrow c(X), A(X)$

obtained by the *Unfold* procedure, is folded by using a definition of the form:

$newp(X) \leftarrow d(X), A(X)$

if $c(X)$ implies $d(X)$, denoted $c(X) \sqsubseteq d(X)$. If $c(X)$ implies $d(X)$, we say that $d(X)$ is a *generalization* of $c(X)$. The result of folding is the specialized clause:

$newq(X) \leftarrow c(X), newp(X)$.

The specialization strategy proceeds by applying the *Unfold* procedure followed by the *Generalize&Fold* procedure to each clause in *NewDefs*, and terminates when no new definitions are needed for performing folding steps. Unfortunately, an uncontrolled application of the *Generalize&Fold* procedure may lead to the introduction of infinitely many new definitions, thereby causing the nontermination of the specialization procedure. In the following section we will define suitable *generalization operators* which guarantee the introduction of finitely many new definitions.

5 Branching Preserving Generalization

In this section we define the branching preserving generalization strategy and the operators which are used to ensure the termination of the specialization strategy. As mentioned in the Introduction, a distinguishing property of the branching preserving generalization operator is that it generalizes the constraints occurring in a candidate definition, so that the generalized definition preserves, in the symbolic evaluation tree, the same branches which can be generated by unfolding from the candidate definition itself.

Let $\mathcal{C}$ denote the set of all linear constraints. The set $\mathcal{C}$ is the minimal set of constraints which: (i) includes all atomic constraints of the form either $p_1 \leq p_2$ or $p_1 < p_2$, where p_1 and p_2 are linear polynomials with integer coefficients, and (ii) it is closed under conjunction, also denoted by $\wedge$. An equation $p_1 = p_2$ stands for $p_1 \leq p_2 \wedge p_2 \leq p_1$. The projection of a constraint c onto a tuple X of variables, denoted $project(c, X)$, is a constraint such that $\mathcal{R} \models \forall X \ (project(c, X) \leftrightarrow \exists Y c)$, where Y is the tuple of variables occurring in c and not in X, and $\mathcal{R}$ is the structure of the real numbers.

In order to specify our branching preserving generalization operator we need the following notion. Let c be a constraint and let A be an atom. Given a program P, the *unfeasible branches* for (c, A), denoted $UnfeasibleBranches(c, A)$, is the set $\{(H_1 \leftarrow c_1, G_1), \ldots, (H_m \leftarrow c_m, G_m)\}$, of (renamed apart) clauses of P such that, for $i = 1, \ldots, m$, A and H_i are unifiable by the most general unifier ϑ_i and $(c \wedge c_i)\,\vartheta_i$ is unsatisfiable.

A branching preserving generalization operator is a generalization operator [11] which preserves the unfeasible branches, as stated by the following definition.

Definition 2 (Branching Preserving Generalization Operator $\ominus_{bp}$). Let $\precsim$ be a thin wqo on the set $\mathcal{C}$ of constraints [6]. A function $\ominus_{bp}$ from $\mathcal{C}\times\mathcal{C}\times Atoms$ to $\mathcal{C}$ is a *branching preserving generalization operator* with respect to $\precsim$ if, for all constraints c, d, for all atoms A, we have: (i) $d \sqsubseteq \ominus_{bp}(c, d, A)$, (ii) $\ominus_{bp}(c, d, A) \precsim c$, and (iii) $UnfeasibleBranches(d, A) = UnfeasibleBranches(\ominus_{bp}(c, d, A), A)$.

The branching preserving generalization strategy is realized by the following procedure *Generalize&Fold* which is an adaptation of the one in [11].

Procedure *Generalize&Fold*
Input: (i) a set *Defs* of definitions structured as a tree of definitions, called *DefsTree*, (ii) a set Γ of clauses obtained from a clause γ by the *Unfold* procedure, and (iii) a branching preserving generalization operator $\ominus_{bp}$.
Output: (i) A set *NewDefs* of new definitions, and (ii) a set Φ of folded clauses.

NewDefs $:= \emptyset$; $\Phi := \Gamma$;
while in Φ there exists a clause η: $H \leftarrow e, G_1, A, G_2$ where the predicate symbol of A occurs in the body of some clause in Γ *do*

GENERALIZE:
Let X be the set of variables occurring in A and $e_X = project(e, X)$.
1. *if* in *Defs* $\cup$ *NewDefs* there exists a clause δ: $newp(X) \leftarrow d, A$ such that $e_X \sqsubseteq d$ and $UnfeasibleBranches(d, A) = UnfeasibleBranches(e_X, A)$ (modulo variable renaming)
   then *NewDefs* := *NewDefs*
2. *elseif* there exists a clause α in *Defs* such that:
   (i) α is of the form $newq(X) \leftarrow b, A$, and (ii) α is the most recent ancestor of γ in *DefsTree* whose body contains an atom of the form A (modulo variable renaming)
   then *NewDefs* := *NewDefs* $\cup \{newp(X) \leftarrow \ominus_{bp}(b, e_X, A), A\}$
3. *else* *NewDefs* := *NewDefs* $\cup \{newp(X) \leftarrow e_X, A\}$

FOLD:
$\Phi := (\Phi - \{\eta\}) \cup \{H \leftarrow e, G_1, newp(X), G_2\}$
end-while

The proof of termination of the *Specialize* procedure of Section 4 is a variant of the proof of Theorem 3 in [10]. Since the correctness of the *Specialize* procedure directly follows from the fact that the transformation rules preserve the least model semantics [7], we have the following result.

Theorem 1 (Branching Preserving Specialization: Termination and Correctness). (i) *The Specialize procedure terminates.* (ii) *Let program* P_S *be the output of the Specialize procedure. Then* `unsafe` $\in M(P)$ *iff* `unsafe` $\in M(P_S)$.

Some generalization operators defined in terms of relations and operators on constraints such as *widening* and *convex-hull*, have been defined in [11]. Now we describe a method for deriving a branching preserving generalization operator from any generalization operator. This method has been applied for deriving the branching preserving generalization operators that we have used in our software model checking experiments reported in Section 6.

A constraint $c_1 \wedge \ldots \wedge c_n$ is *in simplified form* iff no i and j exist, with $1 \leq i < j \leq n$, such that $c_i \sqsubseteq c_j$ or $c_j \sqsubseteq c_i$. Let $\preceq$ be a thin wqo and B a finite set of (non necessarily atomic) constraints. The wqo *derived from* $\preceq$ *and* B, denoted $\preceq_B$, is a binary relation on constraints such that $c \preceq_B d$ iff *either* (i) $c \preceq d$, *or*

(ii) $c \equiv c_1 \wedge \ldots \wedge c_n$ is in simplified form and there exists $i \in \{1, \ldots, n\}$ such that $c_i \in B$ and $(c_1 \wedge \ldots \wedge c_{i-1} \wedge c_{i+1} \wedge \ldots \wedge c_n) \precsim_B d$. It can be shown that $\precsim_B$ is a thin wqo.

A *disjointedness preserving bound* $dpb(d, B)$ of a constraint d and a (finite) set B of constraints is the most general constraint such that $d \sqsubseteq dpb(d, B)$ and, for all atomic constraints b_i occurring in a constraint in B, if $d \wedge b_i$ is unsatisfiable, so is $dpb(d, B) \wedge b_i$.

We construct $dpb(d, B)$ as the conjunction of all atomic constraints r such that: (i) $d \sqsubseteq r$, and (ii) there exists a constraint $b_1 \wedge \ldots \wedge b_n$ in B and $r = neg(b_i)$, for some $i \in \{1, \ldots, n\}$, where $neg(a)$ denotes the negation of an atomic constraint a defined as follows: $neg(p < 0)$ is $-p \leq 0$ and $neg(p \leq 0)$ is $-p < 0$.

The *head constraint* of a clause γ of the form $H \leftarrow c, A$ is the constraint $project(c, X)$, where X is the tuple of variables occurring in H.

Let $\precsim$ be a thin wqo on the set $\mathcal{C}$ of constraints. Let $\ominus$ be a generalization operator with respect to $\precsim$, that is, by definition, for all constraints c and d in $\mathcal{C}$, (i) $d \sqsubseteq c \ominus d$, and (ii) $c \ominus d \precsim c$ [11]. For all constraints c, d, for all atoms A, we define $\ominus_{bp}(c, d, A)$ to be $(c \ominus d) \wedge dpb(d, B)$, where B is the set of head constraints of clauses in $UnfeasibleBranches(d, A)$.

We observe that: (i) since $c \ominus d$ and $dpb(d, B)$ are generalizations of d, also their conjunction is; (ii) by definition of $\precsim_B$, for all constraints e, if $c \ominus d \precsim e$ then $\ominus_{bp}(c, d, A) \precsim_B e$, and (iii) by definition of disjointedness preserving bound, $(c \ominus d) \wedge dpb(d, B)$ and d have the same set of unfeasible branches with respect to the atom A. Thus, we have the following result.

Proposition 1. *The operator $\ominus_{bp}$ is a branching preserving generalization operator with respect to the thin well-quasi ordering $\precsim_B$.*

6 Experimental Evaluation

In this section we present some preliminary results obtained by applying our software model checking method to some benchmark programs taken from the literature. The results show that our approach is viable and competitive with the state-of-the-art software model checkers.

Programs *ex*1, *f*1*a*, *f*2, and *interp* have been taken from the benchmark set of DAGGER [14]. Program *substring* and *tracerP* are taken from [20] and [16], respectively. Program *re*1 and *singleLoop* have been introduced to illustrate the branching preserving strategy. Finally, *selectSort* is an encoding of the Selection sort algorithm where references to arrays have been replaced by using the non-deterministic choice operator `nd` to perform array bounds checking. The source code of all the above programs is available at `http://map.uniroma2.it/smc/`.

Our model checker uses the MAP system [26] which is a tool for transforming constraint logic programs implemented in SICStus Prolog. MAP uses the `clpr` library to operate on constraints over the reals. Our model checker consists of three modules: (i) a translator which takes a property *initProp* and a command *com* and returns their accociated terms, (ii) the MAP system for CLP program

specialization which performs Phases (1) and (2) of our method, and (iii) a program for computing the least models of CLP programs which performs Phase (3) of our method.

We have also run three state-of-the-art CLP-based software model checkers on the same set of programs, and we have compared their performance with that of our model checker. In particular, we have used: (i) ARMC [28], (ii) HSF(C) [13], and (iii) TRACER [17]. ARMC and HSF(C) are CLP-based software model checkers which implement the CEGAR technique. TRACER is a CLP-based model checker which uses Symbolic Execution (SE) for the verification of safety properties of sequential C programs using approximated preconditions and postconditions as interpolants.

Program	MAP				ARMC	HSF(C)	TRACER	
	W	W_{bp}	$CHWM$	$CHWM_{bp}$			$SPost$	$WPre$
$ex1$	1.08	1.09	1.14	1.25	0.18	0.21	∞	1.29
$f1a$	∞	∞	0.35	0.36	∞	0.20	$\perp$	1.30
$f2$	∞	∞	0.75	0.88	∞	0.19	∞	1.32
$interp$	0.29	0.29	0.32	0.44	0.13	0.18	∞	1.22
$re1$	∞	0.33	0.33	0.33	∞	0.19	∞	∞
$selectSort$	4.34	4.70	4.59	5.57	0.48	0.25	∞	∞
$singleLoop$	∞	∞	∞	0.26	∞	∞	$\perp$	1.28
$substring$	88.20	171.20	5.21	5.92	931.02	1.08	187.91	184.09
$tracerP$	0.11	0.12	0.11	0.12	∞	∞	1.15	1.28

Table 1. Time (in seconds) taken for performing model checking. '∞' means 'no answer within 20 minutes', and '$\perp$' means 'termination with error'.

Table 1 reports the results of our experimental evaluation which has been performed on an Intel Core Duo E7300 2.66Ghz processor with 4GB of memory under the GNU Linux operating system.

In columns W and $CHWM$ we report the results obtained by the MAP system when using the procedure presented in Section 5 and the generalization operators $Widen$ and $CHWidenMax$ [11], respectively. In columns W_{bp} and $CHWM_{bp}$ we report the results for the branching preserving versions $Widen_{bp}$ and $CHWidenMax_{bp}$ of those generalization operators. In the remaining columns we report the results obtained by ARMC, HSF(C), and TRACER using the strongest postcondition ($SPost$) and the weakest precondition ($WPre$) as interpolants, respectively.

On the selected set of examples, we have that the MAP system with the $CHWidenMax_{bp}$ is able to verify 9 properties out of 9, while the other tools do not exceed 7 properties. Also the verification time is generally comparable to that of the other tools, and it is not much greater than that of the fastest tools.

Note that there are two examples ($re1$ and $singleLoop$) where branching preserving generalization operators based on widening and convex-hull are strictly more powerful than the corresponding operators which are not branching preserving.

We also observe that the use of a branching preserving generalization operator usually causes a very small increase of the verification time with respect to

the non-branching preserving counterparts, thus making branching preserving generalization a promising technique that can be used in practice for software verification.

7 Related Work and Conclusions

The specialization-based software model checking technique presented in this paper is an extension of the technique for the verification of safety properties of infinite state reactive systems, encoded as CLP programs, presented in [9,11]. The main novelties of the present paper are that here we consider imperative sequential programs and we propose a new specialization strategy which ensures branching preservation.

The use of constraint logic programming and program specialization for verifying properties of imperative programs has also been proposed by [27]. In that paper, the interpreter of an imperative language is encoded as a CLP program. Then the interpreter is specialized with respect to a specific imperative program to obtain a residual program on which a static analyser for CLP programs is applied. Finally, the information gathered during this process is translated back in the form of invariants of the original imperative program. Our approach does not require static analysis of CLP and, instead, we discover program invariants *during* the specialization process by means of (branching preserving) generalization operators.

The idea of branching preserving generalization is related to the technique for preserving *characteristic trees* while applying abstraction during partial deduction [24]. Indeed, a characteristic tree provides an abstract description of the tree generated by unfolding a given goal, and abstraction corresponds to generalization. However, the partial deduction technique considered in [24] is applied to ordinary logic programs (not CLP programs) and constraints such as equations and inequations on finite terms are only used in an intermediate phase.

In order to prove that a program satisfies a given property, software model checking methods try to automatically construct a conservative model (that is, a property-preserving model) of the program such that, if the model satisfies the given property, then also does the actual program. In constructing such a model a software model checker may follow two dual approaches: either (i) it may start from a coarse model and then progressively refine it by incorporating new facts, or (ii) it may start from a concrete model and then progressively abstract away from it some irrelevant facts.

Our verification method follows the second approach. Given a program P, we model its computational behaviour as a CLP program (Phase 1) by using the interpreter of the language in which P is written. Then, the CLP program is specialized with respect to the property to be verified, by using generalization operators which preserve the branching behaviour for avoiding loss of precision and, at the same time, enforcing the termination of the specialization process (Phase 2).

In order to get a conservative model of a program, different generalization operators have been introduced in the literature. In particular, in [2] the authors

introduce the *bounded widen* operator $c\nabla_B d$, defined for any given constraint c and d and any set B of constraints. This operator, which improves the precision of the *widen* operator introduced in [5], has been applied in the verification of synchronous programs and linear hybrid systems. A similar operator $c\nabla_B d$, called *widening up to* B, has been introduced in [15]. In this operator the set B of constraints is statically computed once the system to be verified is given. There is also a version of that operator, called *interpolated widen*, in which the set B is dynamically computed [14] by using the interpolants which are derived during the counterexample analysis.

Similarly to [2,5,14,15], the main objective of the branching preserving generalization operator introduced in this paper is the improvement of precision during program specialization. In particular, this generalization operator, similar to the bounded widen operator, limits the possible generalizations on the basis of a set of constraints defined by the CLP program obtained as output of Phase 1. Since this set of constraints which limits the generalization depends on the output of Phase 1, our generalization is more flexible than the one presented in [2]. Moreover, our generalization operator is more general than the classical widening operator introduced in [4]. Indeed, we only require that the set of constraints which have a non-empty intersection with the generalized constraint $c \ominus d$, are entailed by d.

Now let us point out some advantages of the techniques for software model checking which, like ours, use methodologies based on program specialization.

(1) First of all, the approach based on specialization of interpreters provides a parametric, and thus flexible, technique for software model checking. Indeed, by following this approach, given a program P written in the programming language L, and a property φ written in a logic M, in order to verify that φ holds for P, (i) we first specify the interpreter I_L for L and we specify the semantics S_M of M (as a proof system or a satisfaction relation) in a suitable metalanguage, (ii) we then specialize the interpreter and the semantics with respect to P and φ, and (iii) we finally analyze the derived specialized program (by possibly applying program specialization again, as done in this paper).

The metalanguage we used in this paper for Step (i) is CLP in which we have specified both the interpreter and the reachability relation (which defines the semantics of the reachability formula to be verified).

These features make program specialization a suitable framework for software model checking in a very agile, effective way, by easily adapting to changes to the syntax and/or semantics of the programming language under consideration and also to different logics where properties of interest are expressed.

(2) By applying suitable generalization operators we can make sure that specialization always terminates and produces an equivalent program with respect to the property of interest. Thus, we can apply a sequence of specializations, thereby refining the analysis to the desired degree of precision.

(3) Program specialization provides a uniform framework for program analysis. Indeed, as already mentioned, abstraction operators can be regarded as particular generalization operators and, moreover, specialization can be easily combined

to other program transformation techniques, such as program slicing, dead code elimination, continuation passing transformation, and loop fusion.

(4) Finally, on a more technical side, program specialization can easily accommodate polyvariant analysis [31] by introducing several specialized predicate definitions corresponding to the same program point.

Our preliminary experimental results show that our approach is viable and competitive with state-of-the-art software model checkers such as ARMC [28], HSF(C) [13] and TRACER [17].

In order to justify the claim that our approach is indeed effective in practice, we plan in the near future to perform experiments on a larger set of examples. In particular, in order to support a larger set of input specifications, we are currently working on rewriting our translator so that it can take CIL (C Intermediate Language) programs [25]. We also plan to extend our interpreter to deal with more sophisticated features of imperative languages such as arrays, pointers, and procedure calls.

Moreover, since our specialization-based method preserves the semantics of the original specification, we also plan to explore how our techniques can be effectively used in a preprocessing step before using existing state-of-the-art software model checkers for improving both their precision and their efficiency.

An extended version of this paper is available at `www.iasi.cnr.it/~proietti/`.

References

1. T. Ball and S. K. Rajamani. Boolean programs: a model and process for software analysis. MSR TR 2000-14, Microsoft Report, 2000.
2. N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and assertions. In: *Proc. CP'95*, LNCS 976, pages 589–623. Springer, 1995.
3. E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In: *Proc. CAV'00*, pages 154–169. Springer, 2000.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In: *Proc. POPL'77*, pages 238–252. ACM Press, 1977.
5. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In: *Proc. POPL'78*, pages 84–96. ACM Press, 1978.
6. N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1-2):69–116, 1987.
7. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
8. F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. In: *Proc. LOPSTR'00*, LNCS 2042, pages 125–146. Springer, 2001.
9. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying CTL properties of infinite state systems by specializing constraint logic programs. In: *Proc. VCL'01*, DSSE-TR-2001-3, pages 85–96. University of Southampton, UK, 2001.
10. F. Fioravanti, A. Pettorossi, and M. Proietti. Verifying infinite state systems by specializing constraint logic programs. R. 657, IASI-CNR, Rome, Italy, 2007.
11. F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Generalization strategies for the verification of infinite state systems. *Theo. Pract. Log. Pro.* To appear, 2012.

12. J. P. Gallagher. Tutorial on specialisation of logic programs. In: *Proc. PEPM'93*, pages 88–98. ACM Press, 1993.
13. S. Grebenshchikov, A. Gupta, N. P. Lopes, C. Popeea, and A. Rybalchenko. HSF(C): A Software Verifier based on Horn Clauses. In: *Proc. TACAS'12*. To appear, 2012.
14. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically Refining Abstract Interpretations. In: *Proc. TACAS'08*, LNCS 4963, pages 443–458. Springer, 2008. `www.cfdvs.iitb.ac.in/~bhargav/dagger.php`
15. N. Halbwachs, Y. E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11:157–185, 1997.
16. J. Jaffar, J. A. Navas, and A. E. Santosa. Symbolic execution for verification. *Computing Research Repository*, 2011.
17. J. Jaffar, J. A. Navas, and A. E. Santosa. TRACER: A Symbolic Execution Tool for Verification, 2012. `paella.d1.comp.nus.edu.sg/tracer/`
18. J. Jaffar, A. Santosa, and R. Voicu. An interpolation method for CLP traversal. In: *Proc. CP'09*, LNCS 5732, pages 454–469. Springer, 2009.
19. R. Jhala and R. Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21:1–21:54, 2009.
20. R. Jhala and K. L. McMillan. A Practical and Complete Approach to Predicate Refinement. In: *Proc. TACAS'06*, LNCS 3920, pages 459–473. Springer, 2006.
21. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.
22. M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theo. Pract. Log. Pro.*, 2(4&5):461–515, 2002.
23. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.
24. M. Leuschel and D. De Schreye. Constrained partial deduction. In: *Proc. WLP'97*, Munich, Germany, pages 116–126, 1997.
25. G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. CC'02*, LNCS 2304, pages 209–265. Springer, 2002. `kerneis.github.com/cil/`
26. The MAP transformation system. `www.iasi.cnr.it/~proietti/system.html`
27. J. C. Peralta, J. P. Gallagher, and H. Saglam. Analysis of Imperative Programs through Analysis of Constraint Logic Programs. In: *Proc. SAS'98*, LNCS 1503, pages 246–261. Springer, 1998.
28. A. Podelski and A. Rybalchenko. ARMC: The Logical Choice for Software Model Checking with Abstraction Refinement. In: *Proc. PADL'07*, LNCS 4354, pages 245–259. Springer, 2007.
29. C. J. Reynolds. *Theories of Programming Languages*. Cambridge Univ. Press, 1998.
30. H. Saïdi. Model checking guided abstraction and analysis. In: *Proc. SAS'00*, pages 377–396. Springer, 2000.
31. S. Scott and T. Wang. Polyvariant flow analysis with constrained types. In: *Proc. ESOP'00*, LNCS 1782, pages 382–396. Springer, 2000.
32. N. Sharygina, S. Tonetta, and A. Tsitovich. An abstraction refinement approach combining precise and approximated techniques. *Soft. Tools Techn. Transf.*, 14(1):1–14, Springer, 2012.
33. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In: *Proc. ILPS'95*, pages 465–479. MIT Press, 1995.

Enhancing Tree Compression in Declarative Debugging*

David Insa, Josep Silva, and César Tomás

Universitat Politècnica de València
Camino de Vera s/n, E-46022 Valencia, Spain.
{dinsa,jsilva,ctomas}@dsic.upv.es

Abstract. Declarative debugging is a semi-automatic debugging technique that allows the programer to debug a program without the need to see the source code. The debugger generates questions about the results obtained in different computations and the programmer only has to answer these questions to automatically find the bug. In this work we present a technique for declarative debugging based on tree compression. This technique is used to compress the representation of loops in a computation. We provide an algorithm to decide when to compress these loops in combination with different declarative debugging strategies.

Keywords: Declarative Debugging, Execution Trees, Tree Compression

1 Introduction

Debugging is one of the most difficult and less automated tasks of software engineering. This is due to the fact that bugs are usually hidden under complex conditions that only happen after particular interactions of software components. The programmer cannot consider all possible computations of her piece of software, and those unconsidered computations usually produce a bug. In words of Brian Kernighan, the difficulty of debugging is explained as follows:

> *"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"*
>
> The Elements of Programming Style, 2nd edition

The problems caused by bugs are highly expensive. Sometimes more than the price of the own development of the product. For instance, the NIST report

* This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02 and by the *Generalitat Valenciana* under grant PROMETEO/2011/052. David Insa was partially supported by the Spanish *Ministerio de Eduación* under FPU grant AP2010-4415.

[6] calculated that undetected software bugs produce a cost to the USA economy of $59 billion per year.

There have been many attempts to define automatic techniques for debugging, but in general poor results have been obtained. One notable exception is Declarative Debugging [8, 9]. In this work we present a technique to improve the performance of Declarative Debugging reducing the debugging time.

Declarative debugging is a semi-automatic debugging technique that automatically generates questions about the results obtained in different subcomputations. Then, the programmer answers the questions and with this information the debugger is able to precisely identify the bug in the source code. Roughly speaking, the debugger discards those parts of the source code that are associated to correct computations until a small part of the code (usually a function or procedure) is isolated. One interesting property of this technique is that the programmer does not need to see the source code during debugging. She only needs to know the actual and the intended results produced by a computation with given inputs. Therefore, if we have available a formal specification of the pieces of software that is able to answer the questions, then the technique is fully automatic.

In declarative debugging, the execution of programs is represented with a data structure called *Execution Tree* (ET) where each node represents a method execution. Moreover, declarative debugging uses a navigation strategy to traverse the ET asking the programmer about the validity of the nodes. Each node contains a particular method execution with its inputs and outputs. If the node is marked as wrong, then the bug must be in the subtree of this wrong node; and it must be in the rest of the tree if it is marked as correct. When a node is marked as wrong, and all its children are marked as correct, then the node is considered buggy, and the debugger reports the associated method as buggy.

One of the main problems of declarative debugging is that it can produce long series of questions making the debugging session too long. Moreover, declarative debugging works at the level of functions, thus the granularity level of the bug found is a function. In this work we propose a new technique that allows declarative debuggers to reduce the number of questions needed to detect a bug. This technique improves a previous technique called *Tree Compression* [3]and it is based on a transformation of the ET in such a way that the produced ET can be debugged more efficiently using the standard algorithms. Therefore the technique is conservative with respect to previous implementations and it can be integrated in any debugger as a preprocessing stage.

The rest of the paper has been organized as follows. In Section 2, we discuss the related work. Section 3 introduces some preliminary definitions that will be used in the rest of the paper. In Section 4 we explain our technique and its main applications, and we introduce the algorithms used to improve the structure of the ET. Then, in Section 5, we provide details about our implementation and some experiments carried out with real Java programs. Finally, Section 6 concludes. All the information related to our experiments, the

source code of the tool, the benchmarks, and other materials can be found at http://www.dsic.upv.es/∼jsilva/DDJ/.

2 Related Work

Reducing the number of questions asked by declarative debuggers is a well-known objective in the field, and there exist several works devoted to achieve this goal. Some of them face the problem by defining different ET transformations that modify the structure of the ET in such a way that its exploration is more efficient.

For instance, in [7], authors propose a technique to improve the declarative debugging of Maude specifications whose effect balances the ET. Having a balanced ET is very convenient when the exploration of the ET is done with strategies such as Divide and Query [9], because, after every answer, it is possible to prune almost half of the tree, thus obtaining a logarithmic number of questions with respect to the number of nodes in the ET. The technique uses a transformation that allows the user to decide whether she wants to keep the transitivity inference steps applied by the calculus. Keeping these steps balances the tree and thus less questions are asked in general. This approach is related to our technique, but it has some drawbacks: it can only be applied where transitivity inferences took place, and thus most of the parts of the tree cannot be balanced, and even in these cases the balancing will only affect two nodes. Our technique, in contrast, is applied to loops, being able to balance the whole loop.

Our technique is based on Tree Compression (TC) introduced by Davie and Chitil [3]. In particular, we define an algorithm to decide when tree compression must be applied (or partially applied) in an ET. TC is a conservative approach that transforms an ET into an equivalent (smaller) ET where the same bugs can be detected. The objective of this technique is essentially different to previous ones: it tries to reduce the size of the ET by removing redundant nodes, and it is only applicable to recursive calls. Let us explain it with an example.

Example 1. Consider the ET in Figure 1. In this ET, every node is labeled (inside) with a number that refers to a specific function, and each node has a number (outside) that indicates the number of questions needed to find the bug if this node is the buggy node. To compute this number, we have considered the navigation strategy Divide and Query [8], that always selects the node that better divides the number of nodes in the ET by the half.

For each recursive call, TC removes the child node associated to the recursive call and all children of this node become children of the parent node. Therefore, six nodes are removed, thus statically reducing the size of the tree.

Observe that the average number of questions has been reduced ($\frac{72}{17}$ vs $\frac{42}{11}$) thanks to the use of TC.

Unfortunately, TC does not always produce good results. Sometimes reducing the number of nodes causes a worst structure of the ET that is more difficult to debug and thus the number of questions is increased, producing the contrary effect to the intended one.

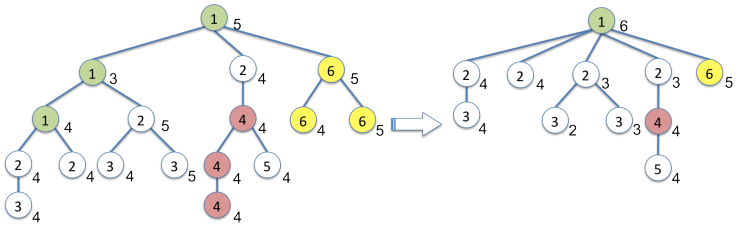


Fig. 1. Example of Tree Compression.

Example 2. Consider the ET in Figure 2. In this ET, the average number of questions needed to find the bug is $\frac{33}{9}$. Nevertheless, after compressing the recursive calls (the dark nodes), the average number of questions is augmented to $\frac{28}{7}$ (see the ET at the left). The reason is that in the new compressed ET we cannot prune any question because its structure is completely flat. The previous structure allowed us to prune some nodes because deep trees are more convenient for declarative debugging. However, if we only compress one of the two recursive calls, the number of questions is reduced to $\frac{27}{8}$.

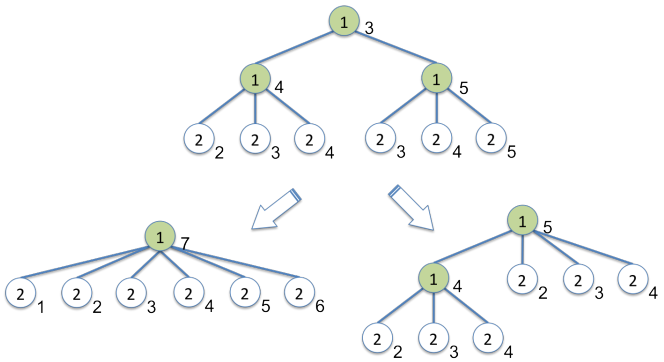


Fig. 2. Negative and positive effects of Tree Compression.

Example 2 clearly shows that TC should not be applied always to all recursive calls. From the best of our knowledge, there does not exist an algorithm to decide when to apply TC, and current implementations always compress all recursive calls [2, 4]. Our new technique solves this problem with an analysis to decide when to compress recursive calls. A similar approach to TC is declarative source debugging [1], that instead of modifying the tree implements an algorithm to prevent the debugger from selecting questions related to nodes generated by recursive calls.

Another approach which is related to ours was presented in [5]. Here, a source code (rather than an ET) transformation for list comprehensions in functional programs was introduced. Concretely, this technique transforms list comprehensions into a set of equivalent functions that implement the iteration. The produced ET can be further transformed to remove the internal nodes reducing the size of the final ET as in the TC technique. This technique is orthogonal to our technique and it can be applied before.

3 Preliminaries

Our ET transformations are based on the structure of the ET and the name of the function used in each node. Therefore, for the purpose of this work, we can provide a definition of execution tree whose nodes are labeled with a number referring to a specific function.

Definition 1 (Execution Tree). *An* execution tree *is a labeled directed tree* $T = (N, E)$ *whose nodes* N *represent method executions and are labeled with method identifiers, where the label of node* n *is referenced with* $l(n)$*. Each edge* $(n \to n') \in E$ *indicates that the method associated to* $l(n')$ *is invoked during the execution of the method associated to* $l(n)$*.*

We use as method identifiers numbers that uniquely identify each method in the source code. This simplification is enough to keep our definitions and algorithms precise and simple. Given an ET, *Simple recursion* is represented with a branch of chained nodes with the same number. *Nested recursion* happens when a branch of chained nodes with the same number is descendant of a node in a recursion. *Multiple recursion* happens when a node labeled with a number n has two or more children labeled with n.

The weight of a node is the number of nodes contained in the tree rooted at this node. We refer to the weight of node n as w_n. In the following, we will refer to the two most used navigation strategies for declarative debugging: Top-Down and Divide and Query (D&Q). In both cases, we will always implicitly refer to the most efficient version of both techniques, namely, (i) *Heaviest First* for Top-Down that always traverses the ET from the root to the leaves selecting always the heaviest node; and (ii) *Hirunkitti's Divide and Query* that always selects the node in the ET that better divides the number of nodes in the ET by the half. A comparative study of these techniques can be found in [9].

For the comparison of strategies we use function $\mathcal{Q}uestions(T, s)$ that computes the number of questions needed (as an average) to find the bug in an ET T using the navigation strategy s.

4 ET Optimization with Tree Compression

In this section we present a new technique for the optimization of ETs. We introduce an algorithm able to decide when to compress a branch of the ET in any case (with simple recursion, nested recursion, and multiple recursion). We also discuss how navigation strategies are affected by TC.

4.1 When to Apply Tree Compression

Tree compression was proposed as a general technique for declarative debugging. However, it was defined in the context of a functional language (Haskell) and with the use of a particular strategy (Hat-Delta). The own authors realized that TC can produce wide trees that are difficult to debug and, for this reason, defined heuristics that avoid asking about the same function repeatedly. Unfortunately, these heuristics are only useful when the same function is repeated in recursive loops.

Consequently, despite TC can be very useful, it can also produce bad ETs as it was shown in Example 2. Unfortunately, the authors of TC did not study how this technique works with other (more extended) strategies such as Top-Down or D&Q, so it is not clear at all when to use TC. To study this, we can consider the most general case of a simple recursion in an ET. It is shown in Figure 3 where the clouds represent possibly empty sets of subtrees and the dark nodes are the recursion chain with a length of n calls, $n \geq 2$. Here, the labels are not method identifiers. They are just used to number each node.

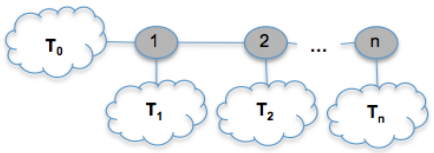


Fig. 3. Recursion chain in an ET.

It should be clear that the recursion chain can be useful to prune nodes of the tree. For instance, in the figure, if we ask for the node $n/2$, we can prune $n/2$ subtrees. Therefore, in the case that the subtrees $T_i, 1 \leq i \leq n$, are empty, then no pruning is possible—note that only the nodes in the recursion chain could be pruned, but then it would be better to prune them directly with TC—and thus TC must be used. Hence, we can conclude that *every recursive call whose only child is another recursive call must be compressed*. This result can be formally stated for Top-Down as follows.

Theorem 1. *Let T be an ET with a recursion chain $R = n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_m$ where the only child of a node n_i, $1 \leq i \leq m-1$, is n_{i+1}. And let T' be an ET equivalent to T except that nodes n_i and n_{i+1} have been compressed. Then, $\mathcal{Q}uestions(T', \text{Top-Down}) < \mathcal{Q}uestions(T, \text{Top-Down})$.*

Proof. Let us consider the two nodes that form the subchain to be compressed. For the proof we can call them $n_1 \rightarrow n_2$. Firstly, the number of questions needed to find a bug in any ancestor of n_1 is exactly the same if we compress or not the subchain. Therefore, it is enough to prove that $\mathcal{Q}uestions(T_{1c}, \text{Top-Down}) <$

$\mathcal{Q}uestions$(T_1,Top-Down) where T_1 is the subtree whose root is n_1 and T_{1c} is the subtree whose root is n_1 after tree compression.
Let us assume that n_2 has j children. Thus we call T_2 the subtree whose root is n_2, and T_{2_i} the subtree whose root is the i-th child of n_2. Then,

$$\mathcal{Q}uestions(T_2,\text{Top-Down}) = \frac{(j+1)+\sum_{i=1}^{j} |T_{2_i}|*(i+\mathcal{Q}uestions(T_{2_i},\text{Top-Down}))}{|T_2|}$$

Here, $(j+1)$ are the questions needed to find a bug in n_2. To reach the children of n_2, the own n_2 and the previous $i-1$ children must be asked first, and this is why we need to add i to $\mathcal{Q}uestions(T_{2_i},\text{Top-Down})$, Finally, $|T_x|$ represents the number of nodes in the (sub)tree T_x.
Therefore, $\mathcal{Q}uestions(T_{1c},\text{Top-Down}) = \mathcal{Q}uestions(T_2,\text{Top-Down})$
and $\mathcal{Q}uestions(T_1,\text{Top-Down}) = \frac{2+|T_2|*(i+\mathcal{Q}uestions(T_2,\text{Top-Down}))}{|T_2|+1}$
Clearly, $\mathcal{Q}uestions(T_{1c},\text{Top-Down}) < \mathcal{Q}uestions(T_1,\text{Top-Down})$, and thus the claim follows.

This theorem provides an opportunity to statically improve the structure of an ET using TC. But TC is not the panacea, and we need to identify in what cases it should be used. D&Q is a good example of strategy where TC has a negative effect.

Tree Compression for D&Q

In general, when debugging an ET with the strategy D&Q, TC should not be applied except in the case described by Theorem 1 ($T_i, 1 \leq i \leq n$, are empty). The reason is that D&Q can jump to any node of the ET without following a predefined path. This allows D&Q to ask about any node of the recursion chain without the need to ask about the previous nodes in the chain. Note that this does not happen in other strategies such as Top-Down. Therefore, D&Q has the ability to use the recursion chain as a mean to divide the iterations by the half.

Given the ET in Figure 3, the only case when D&Q cannot use the recursion chain to prune nodes is when the sets of subtrees $T_1 \ldots T_n$ are empty. In this case, TC must be used. However, if the subtrees contain at least one node, then TC should not be used. Observe in Figure 4 that, except for very small recursion chains (e.g., $n \leq 3$), D&Q can take advantage of the recursion chain to prune half of the iterations. The greater is n the more nodes pruned. Observe that even with a single child in the recursion chain nodes (e.g., $T_i, 1 \leq i \leq n$, is a single node), D&Q can prune nodes. Therefore, if we add more nodes to the subtrees Ti, then more nodes can be pruned and D&Q will behave even better.

Tree Compression for Top-Down

In the case of Top-Down-based strategies, it is not trivial at all to decide when to apply TC and when not to apply it. Considering again the ET in Figure 3, there are two factors that must be considered to decide when to apply TC:

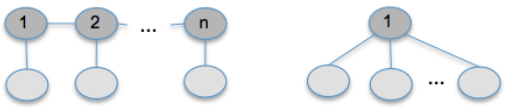


Fig. 4. TC applied to a recursive loop.

1. the length n of the recursive chain, and
2. the size of the trees $T_i, 1 \leq i \leq n$.

In order to decide in which cases TC should be applied, we provide Algorithm 1 that takes an ET and determines what nodes from the recursion chains must be compressed.

Essentially, Algorithm 1 analyzes for each recursion sequence what is the effect of applying TC, and it is finally applied only when it produces an improvement. This analysis is done little by little, analyzing each pair of parent-child (recursive) nodes in the sequence separately. Thus, it is possible that the final result is to only compress one (or several) parts of one recursion chain. For this, variable *recs* initially contains all nodes of the ET with a recursive child. Each of these nodes is processed with the loop in line 1 bottom-up (lines 2-3). That is, the nodes closer to the leaves are processed first. In order to also consider multiple recursion, the algorithm uses the loops in lines 5 and 8. These loops analyze each recursive branch independently and for each of them, they study the improvement achieved when a branch is compressed. This improvement is stored in variable *improvement*. The algorithm uses two functions whose code has been included (Cost and Compress), and three more functions whose code has not been included because they are trivial: function Children computes the set of children of a node in the ET (i.e., Children$(m) = \{n \mid (m \rightarrow n) \in E\}$); function Sort takes a set of nodes and produces an ordered sequence where nodes have been decreasingly ordered by their weights; and function Pos takes a node and a sequence of nodes and returns the position of the node in the sequence.

Given two nodes *parent* and *child* candidates to make a TC, the algorithm first sorts the children of both the *parent* and the *child* (lines 9-10) in the order in which Top-Down would ask them (sorted by their weight). Then, it combines the children of both nodes simulating a TC (line 11). Finally, it compares the average number of questions of both possibilities (line 12). The equation that appears in line 12 is one of the main contributions of the algorithm, because this equation determines when to perform TC between two nodes in a chain with the strategy Top-Down. This equation depends in turn on the formula (line 21 in function Cost) used to compute the average cost of exploring an ET with Top-Down.

Algorithm 1 Optimized Tree Compression

Input: An ET $T = (N, E)$
Output: An ET T'
Inicialization: $T' = T$ and $recs = \{n \mid n, n' \in N \wedge (n \rightarrow n') \in E \wedge l(n) = l(n')\}$

begin
1) **while** ($recs \neq \emptyset$)
2) **take** $n \in recs$ such that $\nexists n' \in recs$ with $(n \rightarrow n') \in E^+$
3) $recs = recs \backslash \{n\}$
4) $parent = n$
5) **do**
6) $maxImprovement = 0$
7) $children = \{c \mid (n \rightarrow c) \in E \wedge l(n) = l(c)\}$
8) **for each** $child \in children$
9) $pchildren = \text{Sort}(\text{Children}(parent))$
10) $cchildren = \text{Sort}(\text{Children}(child))$
11) $comb = \text{Sort}((pchildren \cup cchildren) \backslash \{child\})$
12) $improvement = \frac{Cost(pchildren) + Cost(cchildren)}{w_{parent}} - \frac{Cost(comb)}{w_{parent}-1}$
13) **if** ($improvement > maxImprovement$)
14) $maxImprovement = improvement$
15) $bestNode = child$
16) **end for each**
17) **if** ($maxImprovement \neq 0$)
18) $T' = \text{Compress}(T', parent, bestNode)$
19) **while** ($maxImprovement \neq 0$)
20) **end while**
end
return T'

function Cost($sequence$)
begin
21) **return** $\sum\{\text{Pos}(node, sequence) * w_{node} \mid node \in sequence\} + |sequence|$
end

function Compress($T = (N, E), parent, child$)
begin
22) $nodes = \text{Children}(child)$
23) $E' = E \backslash \{(child \rightarrow n) \in E \mid n \in nodes\}$
24) $E' = E' \cup \{(parent \rightarrow n) \mid n \in nodes\}$
25) $N' = N \backslash \{child\}$
end
return $T' = (N', E')$

5 Implementation

We have implemented for Java the original TC algorithm and the optimized version presented in this paper. This implementation has been integrated into

the Declarative Debugger for Java DDJ [4]. The source code of all the transformations is publicly available at: http://www.dsic.upv.es/∼jsilva/DDJ/.

All the implementation has been done in Java. After the implementation we conducted a series of experiments in order to measure the influence of both techniques in the performance of the debugger. The results obtained can be found in the URL of the implementation. From the experiments we concluded that our technique produce a reduction in the number of questions with an average of 9.72%.

6 Conclusions

One of the main problems of declarative debugging is that it can generate too many questions to find a bug. In this work we have made a step forward to solve this problem. We have introduced a technique that reduces the number of questions by improving the structure of the ET. The technique uses an algorithm to decide when to apply a transformation called tree compression. For future work we are investigating the possibility of using a transformation to expand the loops of a program, and combine this transformation with tree compression.

7 Acknowledgements

We want to thank Rafael Caballero and Adrián Riesco for productive comments and discussions at the initial stages of this work.

References

1. Miguel Calejo. *A Framework for Declarative Prolog Debugging.* PhD thesis, New University of Lisbon, 1992.
2. T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In Andrew Butterfield, editor, *Draft Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL'05)*, page 11. Tech. Report No: TCD-CS-2005-60, University of Dublin, Ireland, September 2005.
3. T. Davie and O. Chitil. Hat-delta: One Right Does Make a Wrong. In *Seventh Symposium on Trends in Functional Programming, TFP 06*, April 2006.
4. D. Insa and J. Silva. An Algorithmic Debugger for Java. *In Proc. of the 26th IEEE International Conference on Software Maintenance*, 0:1–6, 2010.
5. H. Nilsson. *Declarative Debugging for Lazy Functional Languages.* PhD thesis, Linköping, Sweden, May 1998.
6. NIST. The Economic Impacts of Inadequate Infrastructure for Software Testing. In USA National Institute of Standards and Technology, editors, *NIST Planning Report 02-3*, May 2002.
7. Adrián Riesco, Alberto Verdejo, Narciso Martí-Oliet, and Rafael Caballero. Declarative Debugging of Rewriting Logic Specifications. *Journal of Logic and Algebraic Programming*, September 2011.
8. E.Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1982.
9. Josep Silva. A Survey on Algorithmic Debugging Strategies. *Advances in Engineering Software*, 42(11):976–991, November 2011.

XACML 3.0 in Answer Set Programming

Carroline Dewi Puspa Kencana Ramli, Hanne Riis Nielson, Flemming Nielson

Department of Informatics and Mathematical Modelling
Danmarks Tekniske Universitet
Lyngby, Denmark
{cdpu,riis,nielson}@imm.dtu.dk

Abstract We present a systematic technique to transform XACML 3.0 policies in Answer Set Programming (ASP). We show that the resulting logic program has a unique answer set that directly corresponds to our formalisation of the standard semantics of XACML 3.0 from [7]. We demonstrate how our results make it possible to use off-the-shelf ASP solvers to formally verify properties of access control policies represented in XACML, such as checking the completeness of a set of access control policies.

Keywords: XACML, access control, policy language, Answer Set Programming

1 Introduction

XACML (eXtensible Access Control Markup Language) is an OASIS[1] standard that describes both a policy language and an access control decision request/response language. The policy language is used to describe access control requirements (*who can do what when*). The request/response language is used to form a query to ask whether or not a given action should be allowed (*request*), and describes the answer to the query (*response*). The response returns an answer in one of four values: *permit* (the access is granted), *deny* (the access is denied), *indeterminate* (a decision cannot be made because there is an error during evaluation of some attributes) or *not applicable* (no policy is relevant to the given request). XACML has been under development for some time – in this paper we consider the latest version 3.0 ratified by OASIS standards organisation on August 10, 2010 [8].

In XACML, policies are combined together to produce a single decision. XACML 3.0 enhances XACML 2.0's [6] existing combination algorithms in the treatment of the indeterminate value. XACML 3.0 distinguishes between the following three types of indeterminate values:

i. *Indeterminate permit* (i_p) – an indeterminate value arising from a policy which could have been evaluated to permit but not deny;
ii. *Indeterminate deny* (i_d) – an indeterminate value arising from a policy which could have been evaluated to deny but not permit;

[1] OASIS (Organization for the Advancement of Structured Information Standard) is a non-for-profit, global consortium that drives the development, convergence, and adoption of e-business standards. Information about OASIS can be found at http://http://www.oasis-open.org/.

iii. *Indeterminate deny permit* ($\mathsf{i_{dp}}$) – an indeterminate value arising from a policy which could have been evaluated as both deny and permit.

The problem with XACML is that its specification is described in natural language (c.f. [8]) and manual analysis of the overall effect and consequences of a large XACML policy set is a very daunting and time-consuming task. How can a policy developer be certain that the represented policies capture all possible requests? Can they lead to conflicting decisions for some request? Do the policies satisfy all required properties? These are complex problems that cannot be solved easily without some sort of automatised support.

To address this problem we propose a logic-based XACML analysis framework using Answer Set Programming (ASP). With ASP we model an XACML Policy Decision Point (PDP) that loads XACML policies and evaluates XACML requests against these policies. The expressivity of ASP and existence of efficient implementations of the answer set semantics, such as `clasp`[2] and `DLV`[3], provides the means for declarative specification and verification of properties of XACML policies.

Our approach is inspired by the work of Ahn *et al.* [1,2]. There are three main differences between our approach and the work of Ahn *et al.* First, while they consider XACML version 2.0, we address the newer version 3.0. Second, Ahn *et al.* produce a monolithic logic program that can be used for the analysis of XACML policies while we take a more modular approach by first modelling an XACML PDP as a logic program and then using this encoding within a larger program for property analysis. Finally, Ahn *et al.* translate the XACML specification directly to logic programming, so the ambiguities in the natural language specification of XACML are also reflected in their encodings. Due to these reasons, we base our encodings on our formalisation of XACML from [7].

We start our work by presenting the formal semantics of XACML that we introduced in [7]. Then we define a step-by-step transformation of XACML policies into logic programs. We conjecture that the unique answer set of the transformed logic program directly corresponds to our formal semantics of XACML.

Outline. We explain the syntax and semantics of XACML 3.0 in Sect. 2. Then we describe the transformation of XACML 3.0 components into a logic program $\mathcal{P}_{\text{XACML}}$ in Sect. 3. We show the relation between XACML 3.0 semantics and the answer sets of $\mathcal{P}_{\text{XACML}}$ in Sect. 4. Next, in Sect. 5, we show how to verify access control properties, such as checking the completeness of a set of policies. We end the paper with conclusions and future work.

2 XACML 3.0

In this section we show the syntax of XACML 3.0 and the semantics of XACML 3.0 components' evaluation based on Committee Specification [8]. We take Ramli *et. al* work [7] as our reference.

[2] `http://www.cs.uni-potsdam.de/clasp/`
[3] `http://www.dlvsystem.com/`

2.1 Abstract Syntax of XACML 3.0

We summarize the syntax of XACML 3.0 in Table 1. To make the notation clear we use bold font for non-terminal **symbols**, typewriter font for terminal `symbols` and *identifiers* and *values* are written in italic font. Moreover, `<XACML Component>` denotes the symbol for an XACML component. A symbol followed by the star symbol (*) indicates that there are zero or more occurences of that symbol. Similarly, a symbol followed by the plus symbol ($^+$) indicates that there are one or more occurences of that symbol. We assume that each policy has a unique identifier (ID).

Table 1. Abstraction of XACML 3.0 Components

XACML Policy Components	
`<PolicySet>`	:- *PolicySetID* = `[<Target>,`$\langle$*PolicySetID*$^*\rangle$, **CombID** `]`
	| *PolicySetID* = `[<Target>,` $\langle$*PolicyID*$^*\rangle$, **CombID** `]`
`<Policy>`	:- *PolicyID* = `[<Target>,` $\langle$*RuleID*$^+\rangle$, **CombID** `]`
`<Rule>`	:- *RuleID* = `[` **Effect**`, <Target> , <Condition> ]`
`<Condition>`	:- *propositional formulae*
`<Target>`	:- `Null` | $\bigwedge$ `<AnyOf>` $^+$
`<AnyOf>`	:- $\bigvee$ `<AllOf>` $^+$
`<AllOf>`	:- $\bigwedge$ `<Match>` $^+$
`<Match>`	:- **AttrType** (*attribute value*)
CombID	:- `po` | `do` | `fa` | `ooa`
Effect	:- `deny` | `permit`
AttrType	:- `subject` | `action` | `resource` | `environment`
XACML Request Component	
`<Request>`	:- { **Attribute**$^+$}
Attribute	:- **AttrType** (*attribute value*) | `error`(**AttrType** (*attribute value*))
	| *external state*

There are three levels of policies in XACML, namely `<PolicySet>`, `<Policy>` and `<Rule>`. `<PolicySet>` or `<Policy>` can act as the root of a set of access control policies while `<Rule>` is a single entity that describes one particular access control policy. Throughout this paper we assume that `<PolicySet>` is the root of the set of access control policies.

Both `<PolicySet>` and `<Policy>` function as containers for a sequence of `<PolicySet>`, `<Policy>` or `<Rule>`. A `<PolicySet>` contains either a sequence of `<PolicySet>` or a sequence of `<Policy>` while a `<Policy>` only can contain a sequence of `<Rule>`. Every sequence of `<PolicySet>`, `<Policy>` or `<Rule>` has an associated *combining algorithm*. There are four common combining algorithms defined in XACML 3.0, namely *permit-overrides* (`po`), *deny-overrides* (`do`), *first-applicable* (`fa`) and *only-one-applicable* (`ooa`).

A `<Rule>` describes an individual access control policy. It regulates whether an access should be *permitted* or *denied*. All `<PolicySet>`, `<Policy>` and `<Rule>` are applicable whenever their `<Target>` matches with the `<Request>`. When the `<Rule>`'s `<Target>` matches with the `<Request>`, then the applicability of the `<Rule>` is refined by its `<Condition>`.

A <Target> is a combination of <Match> elements. Each <Match> element describes an *attribute* that a <Request> should match in order to activate a policy. There are four attribute categories in XACML 3.0, namely *subject* attribute, *action* attribute, *resource* attribute and *environment* attribute. The subject attribute is the entity requesting access, e.g., a file system, a workstation, etc. The action attribute defines the type of access requested, e.g., to read, to write, to delete, etc. The resource attribute is data, service or a system component. The environment attribute can optionally provide additional information.

A <Request> contains a set of attribute values for a particular access request. A <Request> can contain additional information such as the state of the environment (e.g. the current time or current temperature) and error messages about errors that occurred during the evaluation of attribute values.

2.2 XACML 3.0 Formal Semantics

The evaluation of XACML policies starts from the evaluation of <Match> elements and continues bottom-up until the evaluation of the root <PolicySet> element. We use the $[\![\cdot]\!]$ notation to map XACML elements into their values (see the summary in Table 2). An XACML component returns an indeterminate value whenever the decision cannot be made. Usually this happens when there is an error during the evaluation process. See [7] for further explanation of the semantics of XACML 3.0.

Table 2. XACML Components' Values

XACML Components	Values
$[\![$<Match>$]\!]$, $[\![$<AllOf>$]\!]$, $[\![$<AnyOf>$]\!]$, $[\![$<Target>$]\!]$	match (m), not match (nm) and indeterminate (idt)
$[\![$<Rule>$]\!]$	permit (p), deny (d), indeterminate permit ($\mathsf{i_d}$), indeterminate deny ($\mathsf{i_p}$), not applicable (na)
$[\![$<Policy>$]\!]$, $[\![$<PolicySet>$]\!]$	permit (p), deny (d), indeterminate permit ($\mathsf{i_d}$), indeterminate deny ($\mathsf{i_p}$), indeterminate deny permit ($\mathsf{i_{dp}}$), not applicable (na)

Evaluation of <Match> into $\{\, \mathsf{m}, \mathsf{nm}, \mathsf{idt} \,\}$**.** Given a <Request> Q, the evaluation of <Match> M is as follows

$$[\![M]\!](Q) = \begin{cases} \mathsf{m} & \text{if } M \in Q \text{ and } \mathsf{error}(M) \notin Q \\ \mathsf{nm} & \text{if } M \notin Q \text{ and } \mathsf{error}(M) \notin Q \\ \mathsf{idt} & \text{if } \mathsf{error}(M) \in Q \end{cases} \tag{1}$$

Evaluation of <AllOf> into $\{\, \mathsf{m}, \mathsf{nm}, \mathsf{idt} \,\}$**.** Given a <Request> Q, the evaluation of <AllOf> $A = \bigwedge_{i=1}^{n} M_i$ is as follows

$$[\![A]\!](Q) = \begin{cases} \mathsf{m} & \text{if } \forall i : [\![M_i]\!] = \mathsf{m} \\ \mathsf{nm} & \text{if } \exists i : [\![M_i]\!] = \mathsf{nm} \\ \mathsf{idt} & \text{otherwise} \end{cases} \tag{2}$$

where each M_i is a <Match> element.

Evaluation of <AnyOf> into $\{ \mathsf{m}, \mathsf{nm}, \mathsf{idt} \}$**.** Given a <Request> Q, the evaluation of <AnyOf> $E = \bigvee_{i=1}^{n} A_i$ is as follows

$$[\![E]\!](Q) = \begin{cases} \mathsf{m} & \text{if } \exists i : [\![A_i]\!] = \mathsf{m} \\ \mathsf{nm} & \text{if } \forall i : [\![A_i]\!] = \mathsf{nm} \\ \mathsf{idt} & \text{otherwise} \end{cases} \tag{3}$$

where each A_i is a <AllOf> element.

Evaluation of <Target> into $\{ \mathsf{m}, \mathsf{nm}, \mathsf{idt} \}$**.** Given a <Request> Q, the evaluation of <Target> $T = \bigwedge_{i=1}^{n} E_i$ is as follows

$$[\![T]\!](Q) = \begin{cases} \mathsf{m} & \text{if } \forall i : [\![E_i]\!] = \mathsf{m} \text{ or } T = \texttt{Null} \\ \mathsf{nm} & \text{if } \exists i : [\![E_i]\!] = \mathsf{nm} \\ \mathsf{idt} & \text{otherwise} \end{cases} \tag{4}$$

where each E_i is a <AnyOf> element. Empty <Target>, indicated by Null always evaluated to m.

Evaluation of <Condition> into $\{ \mathsf{t}, \mathsf{f}, \mathsf{idt} \}$**.** Given a <Request> Q, the evaluation of <Condition> C is as follows

$$[\![C]\!](Q) = \mathsf{eval}(C, Q) \tag{5}$$

Note: The eval is an unspecified function that returns $\{ \mathsf{t}, \mathsf{f}, \mathsf{idt} \}$.

Evaluation of <Rule> into $\{ \mathsf{d}, \mathsf{p}, \mathsf{i_d}, \mathsf{i_p}, \mathsf{na} \}$**.** Given a <Request> Q, the evaluation of <Rule> $R = [e, T, C]$ as follows

$$[\![R]\!](Q) = \begin{cases} e & \text{if } [\![T]\!](Q) = \mathsf{m} \text{ and } [\![C]\!](Q) = \mathsf{t} \\ \mathsf{na} & \text{if } ([\![T]\!](Q) = \mathsf{m} \text{ and } [\![C]\!](Q) = \mathsf{f}) \text{ or} [\![T]\!](Q) = \mathsf{nm} \\ i_e & \text{otherwise} \end{cases} \tag{6}$$

where $e \in \{ \mathsf{p}, \mathsf{d} \}$, T is a <Target> element and C is a <Condition> element.

Evaluation of <Policy> into $\{ \mathsf{d}, \mathsf{p}, \mathsf{i_d}, \mathsf{i_p}, \mathsf{i_{dp}}, \mathsf{na} \}$**.** Given a <Request> Q, the evaluation of <Policy> $P = [T, \langle R_1, \ldots, R_n \rangle, \mathsf{CombID}]$ is as follows

$$[\![P]\!](Q) = \begin{cases} \mathsf{i_d} & \text{if } [\![T]\!](Q) = \mathsf{idt} \text{ and } \bigoplus_{\mathsf{CombID}}(\mathbf{R}) = \mathsf{d} \\ \mathsf{i_p} & \text{if } [\![T]\!](Q) = \mathsf{idt} \text{ and } \bigoplus_{\mathsf{CombID}}(\mathbf{R}) = \mathsf{p} \\ \mathsf{na} & \text{if } [\![T]\!](Q) = \mathsf{nm} \text{ or } \forall i : [\![R_i]\!](Q) = \mathsf{na} \\ \bigoplus_{\mathsf{CombID}}(\mathbf{R}) & \text{otherwise} \end{cases} \tag{7}$$

where T is a <Target> element, and each R_i is a <Rule> element. We use $\mathbf{R}$ to denote $\langle [\![R_1]\!](Q), \ldots, [\![R_n]\!](Q) \rangle$.
Note: The combining algorithm denoted by $\bigoplus_{\mathsf{CombID}}$ will be explained in Sect. 2.3.

Evaluation of <PolicySet> into $\{ \mathsf{d}, \mathsf{p}, \mathsf{i_d}, \mathsf{i_p}, \mathsf{i_{dp}}, \mathsf{na} \}$**.** Given a <Request> Q, the evaluation of <PolicySet> $PS = [T, \langle P_1, \ldots, P_n \rangle, \mathsf{CombID}]$ is as follows

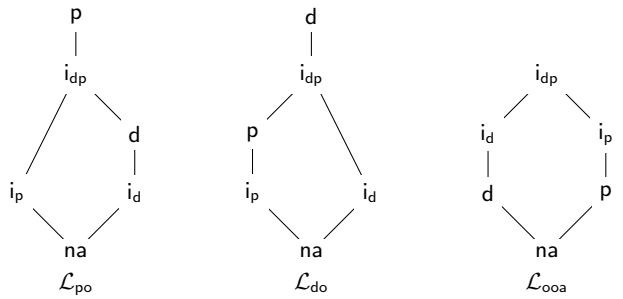


Figure 1. The lattice $\mathcal{L}_{\mathsf{po}}$ for the Permit-Overrides Combining Operator (left); the lattice $\mathcal{L}_{\mathsf{do}}$ for the Deny-Overrides Combining Operator (middle); and the lattice $\mathcal{L}_{\mathsf{ooa}}$ for the Only-One-Applicable Combining Operator (right).

$$[\![PS]\!](Q) = \begin{cases} \mathsf{i_d} & \text{if } [\![T]\!](Q) = \mathsf{idt} \text{ and } \bigoplus_{\mathsf{CombID}}(\mathbf{P}) = \mathsf{d} \\ \mathsf{i_p} & \text{if } [\![T]\!](Q) = \mathsf{idt} \text{ and } \bigoplus_{\mathsf{CombID}}(\mathbf{P}) = \mathsf{p} \\ \mathsf{na} & \text{if } [\![T]\!](Q) = \mathsf{nm} \text{ or } \forall i : [\![P_i]\!](Q) = \mathsf{na} \\ \bigoplus_{\mathsf{CombID}}(\mathbf{P}) & \text{otherwise} \end{cases} \tag{8}$$

where T is a `<Target>` element and each P_i is a `<Policy>` (or `<PolicySet>`) element. We use $\mathbf{P}$ to denote $\langle [\![P_1]\!](Q), \ldots, [\![P_n]\!](Q) \rangle$.

2.3 XACML Combining Algorithms

There are four common combining algorithms defined in XACML 3.0, namely permit-overrides (po), deny-overrides (do), first-applicable (fa) and only-one-applicable (ooa). The permit-overrides combining algorithm takes permit decision as the most priority than deny decision while the deny-overrides combining algorithm takes deny decision over permit. Likewise their names, the first-applicable combining algorithm return the first `<Rule>` (or `<Policy>` or `<PolicySet>`) that is *not* not-applicable and the only-one-applicable combining algorithm return a decision whenever only one `<Rule>` (or `<Policy>` or `<PolicySet>`) which is *not* not-applicable, otherwise it returns indeterminate value. The lattices in Figure 1 describe the permit-overrides combining algorithm, deny-overrides combining algorithm and the only-one-applicable combining algorithm. In this paper, we do not consider the deny-overrides combining algorithm since it is the mirror of the permit-overrides combining algorithm (see Fig. 1 for comparison between Lattices $\mathcal{L}_{\mathsf{po}}$ and $\mathcal{L}_{\mathsf{do}}$).

Permit-Overrides (po) Combining Algorithm. Let $\langle s_1, \ldots, s_n \rangle$ be a sequence of element of $\{ \mathsf{p}, \mathsf{d}, \mathsf{i_p}, \mathsf{i_d}, \mathsf{i_{dp}}, \mathsf{na} \}$. The *permit-overrides combining operator* is defined as follows:

$$\bigoplus_{\mathsf{po}}(\langle s_1, \ldots, s_n \rangle) = \bigsqcup_{\mathsf{po}} \{ s_1, \ldots, s_n \} \tag{9}$$

First-Applicable (fa) Combining Algorithm. Let $\langle s_1, \ldots, s_n \rangle$ be a sequence of element of $\{ \mathsf{p}, \mathsf{d}, \mathsf{i_p}, \mathsf{i_d}, \mathsf{i_{dp}}, \mathsf{na} \}$. The *first-applicable combining operator* is defined as follows:

$$\bigoplus_{\mathsf{fa}}(\langle s_1, \ldots, s_n \rangle) = \begin{cases} s_i & \exists i : s_i \neq \mathsf{na} \text{ and } \forall j : (j < i) \Rightarrow (s_j = \mathsf{na}) \\ \mathsf{na} & \text{otherwise} \end{cases} \tag{10}$$

Only-One-Applicable (ooa) Combining Algorithm. Let $\langle s_1, \ldots, s_n \rangle$ be a sequence of element of $\{ \mathsf{p}, \mathsf{d}, \mathsf{i_p}, \mathsf{i_d}, \mathsf{i_{dp}}, \mathsf{na} \}$. The *only-one-applicable combining operator* is defined as follows:

$$\bigoplus_{\mathsf{ooa}}(\langle s_1, \ldots, s_n \rangle) = \begin{cases} \mathsf{i_d} & \exists i, j : i \neq j, s_i = s_j = \mathsf{d} \text{ and} \\ & \forall k : (s_k \neq \mathsf{d}) \Rightarrow (s_k = \mathsf{na}) \\ \mathsf{i_p} & \exists i, j : i \neq j, s_i = s_j = \mathsf{p} \text{ and} \\ & \forall k : (s_k \neq \mathsf{p}) \Rightarrow (s_k = \mathsf{na}) \\ \bigsqcup_{\mathsf{ooa}} \{ s_1, \ldots, s_n \} & \text{otherwise} \end{cases} \tag{11}$$

3 Transforming XACML Components into Logic Programs

In this section we show step by step how to transform XACML 3.0 components into logic programs. We begin by introducing the syntax of logic programs (LP). Next we show the transformation of XACML component into LP starting from `<Request>` element to `<PolicySet>` element. We also present transformations for combining algorithms. The transformation of each XACML element is based on its formal semantics explained in Sections 2.2 and 2.3.

3.1 Preliminaries

We recall some basic notation and terminology that we use in the remainder of this paper.

First-Order Language. We consider an *alphabet* consisting of (finite or countably infinite) disjoint sets of variables, constants, function symbols, predicate symbols, connectives $\{ \mathbf{not}, \wedge, \leftarrow \}$, punctuation symbols $\{ \text{"("}, \text{","}, \text{","}, \text{")"}, \text{"."} \}$ and special symbols $\{ \top, \bot \}$. In this paper we use upper case letters to denote variables and lower case letters to denote constants, function and predicate symbols. Terms, atoms, literals and formulae are defined as usual. The *language* given by an alphabet consists of the set of all formulae constructed from the symbols occurring in the alphabet.

Logic Programs. A *rule* is an expression of the form

$$A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \mathbf{not}\ B_{m+1} \wedge \cdots \wedge \mathbf{not}\ B_n. \tag{12}$$

where A is either an atom or $\bot$ and each B_i, $1 \leq i \leq n$, is an atom or $\top$. $\top$ is a valid formula. We usually write $B_1 \wedge \cdots \wedge B_m \wedge \mathbf{not}\ B_{m+1} \wedge \cdots \wedge \mathbf{not}\ B_n$ simply as $B_1, \ldots, B_m, \mathbf{not}\ B_{m+1}, \ldots, \mathbf{not}\ B_n$. We refer the rule as a *constraint* when $A = \bot$.

One should observe that the body of a rule must not be empty. A rule of the form $A \leftarrow \top$ is called a *fact*.

A *logic program* (LP) is a finite set of rules. $ground(\mathcal{P})$ denotes the set of all ground instances of rules in the program $\mathcal{P}$.

3.2 XACML Components Transformation into Logic Programs

The transformation of XACML components is based on the semantics of each component explained in Sect. 2.2. First we recall the syntax of each component, then we introduce its transformation into a logic program.

<Request> Transformation. *XACML Syntax:* Let $Q = \{ A_1, \ldots, A_n \}, 1 \leq i \leq n$, be a <Request> component. The transformation of <Request>, Q, into LP $\mathcal{P}_Q$ is as follows

$$A_i \leftarrow \top.\ 1 \leq i \leq n$$

<Match> Transformation. *XACML Syntax:* Let M be a <Match> component. The transformation of <Match> M into LP $\mathcal{P}_M$ is as follows (see (1) for <Match> evaluation)

$$\begin{array}{ll} \mathsf{val}(M, \mathsf{m}) & \leftarrow M, \mathbf{not}\ \mathsf{error}(M). \\ \mathsf{val}(M, \mathsf{nm}) & \leftarrow \mathbf{not}\ M, \mathbf{not}\ \mathsf{error}(M). \\ \mathsf{val}(M, \mathsf{idt}) & \leftarrow \mathsf{error}(M). \end{array}$$

<AllOf> Transformation. *XACML Syntax:* Let $A = \bigwedge_{i=1}^{n} M_i$ be an <AllOf> component where each M_i is a <Match> component. The transformation of <AllOf> A into LP $\mathcal{P}_A$ is as follows (see (2) for <AllOf> evaluation)

$$\begin{array}{ll} \mathsf{val}(A, \mathsf{m}) & \leftarrow \mathsf{val}(M_1, \mathsf{m}), \ldots, \mathsf{val}(M_n, \mathsf{m}). \\ \mathsf{val}(A, \mathsf{nm}) & \leftarrow \mathsf{val}(M_i, \mathsf{nm}).\ (1 \leq i \leq n) \\ \mathsf{val}(A, \mathsf{idt}) & \leftarrow \mathbf{not}\ \mathsf{val}(A, \mathsf{m}), \mathbf{not}\ \mathsf{val}(A, \mathsf{nm}). \end{array}$$

<AnyOf> Transformation. *XACML Syntax:* Let $E = \bigvee_{i=1}^{n} A_i$ be an <AnyOf> component where each A_i is an <AllOf> component. The transformation of <AnyOf> E into LP $\mathcal{P}_E$ is as follows (see (3) for <AnyOf> evaluation)

$$\begin{array}{ll} \mathsf{val}(E, \mathsf{m}) & \leftarrow \mathsf{val}(A_i, \mathsf{m}).\ (1 \leq i \leq n) \\ \mathsf{val}(E, \mathsf{nm}) & \leftarrow \mathsf{val}(A_1, \mathsf{nm}), \ldots, \mathsf{val}(A_n, \mathsf{nm}). \\ \mathsf{val}(E, \mathsf{idt}) & \leftarrow \mathbf{not}\ \mathsf{val}(E, \mathsf{m}), \mathbf{not}\ \mathsf{val}(E, \mathsf{nm}). \end{array}$$

<Target> Transformation. *XACML Syntax:* Let $T = \bigwedge_{i=1}^{n} E_i$ be a <Target> component where each E_i is an <AnyOf> component. The transformation of <Target> T into LP $\mathcal{P}_T$ is as follows (see (4) for <Target> evaluation)

$$\begin{array}{ll} \mathsf{val}(T, \mathsf{m}) & \leftarrow \mathsf{val}(E_1, \mathsf{m}), \ldots, \mathsf{val}(E_n, \mathsf{m}). \\ \mathsf{val}(null, \mathsf{m}) & \leftarrow \top. \\ \mathsf{val}(T, \mathsf{nm}) & \leftarrow \mathsf{val}(E_i, \mathsf{nm}).\ (1 \leq i \leq n) \\ \mathsf{val}(T, \mathsf{idt}) & \leftarrow \mathbf{not}\ \mathsf{val}(T, \mathsf{m}), \mathbf{not}\ \mathsf{val}(T, \mathsf{nm}). \end{array}$$

<Condition> Transformation. *XACML Syntax:* We assume that the <Condition> element is a boolean formula which the evaluation of <Condition> is based on eval function. The transformation of <Condition> C into LP $\mathcal{P}_C$ is as follows

$$\mathsf{val}(C, V) \leftarrow \mathsf{eval}(C, V).$$

The possibility of eval function for "rule r1: patient only can see his or her patient record" is like following

$\mathcal{P}_{cond(r1)}$:

$$\begin{array}{ll}
\mathsf{val}(cond(r1), V) & \leftarrow \mathsf{eval}(cond(r1), V). \\
\mathsf{eval}(cond(r1), \mathsf{t}) & \leftarrow patient_id(X), patient_record_id(X), \\
 & \quad \mathbf{not}\ \mathsf{error}(patient_id(X)), \mathbf{not}\ \mathsf{error}(patient_record_id(X)). \\
\mathsf{eval}(cond(r1), \mathsf{f}) & \leftarrow patient_id(X), patient_record_id(Y), X \neq Y, \\
 & \quad \mathbf{not}\ \mathsf{error}(patient_id(X)), \mathbf{not}\ \mathsf{error}(patient_record_id(Y)). \\
\mathsf{eval}(cond(r1), \mathsf{idt}) & \leftarrow \mathbf{not}\ \mathsf{eval}(cond(r1), \mathsf{t}), \mathbf{not}\ \mathsf{eval}(cond(r1), \mathsf{f}).
\end{array}$$

The $\mathsf{error}(patient\_id(X))$ and $\mathsf{error}(patient\_record\_id(X))$ indicate possible errors might occur, e.g., the system could not connect to the database so that the system does not know the ID of the patient.

<Rule> Transformation. *XACML Syntax:* Let $R_{id} = [E, T, C]$ be a <Rule> component where $E \in \{\ \mathsf{p}, \mathsf{d}\ \}$, T is a <Target> and C is a <Condition>. The transformation of <Rule> R_{id} into LP $\mathcal{P}_{R_{id}}$ is as follows (see (6) for <Rule> evaluation)

$$\begin{array}{l}
\mathsf{val}(R_{id}, E) \leftarrow \mathsf{val}(T, \mathsf{m}), \mathsf{val}(C, \mathsf{t}). \\
\mathsf{val}(R_{id}, \mathsf{na}) \leftarrow \mathsf{val}(T, \mathsf{m}), \mathsf{val}(C, \mathsf{f}). \\
\mathsf{val}(R_{id}, \mathsf{na}) \leftarrow \mathsf{val}(T, \mathsf{nm}). \\
\mathsf{val}(R_{id}, i_E) \leftarrow \mathbf{not}\ \mathsf{val}(R_{id}, E), \mathbf{not}\ \mathsf{val}(R_{id}, \mathsf{na}).
\end{array}$$

<Policy> Transformation. *XACML Syntax:* Let $P_{id} = [T, \langle R_1, \ldots, R_n \rangle, \mathsf{CombID}]$ be a <Policy> component where T is a <Target>, $\langle R_1, \ldots, R_n \rangle$ be a sequence of <Rule> elements and CombID be a combining algorithm identifier. In order to indicate that the <Policy> contains <Rule> R_i, thus for every <Rule> R_i contained in $P_{id} = [T, \langle R_1, \ldots, R_n \rangle, \mathsf{CombID}]$, $\mathcal{P}_{P_{id}}$ also contains:

$$\mathsf{decision\_of}(P_{id}, R_i, E) \leftarrow \mathsf{val}(R_i, E).\ (1 \leq i \leq n)$$

Next, we do a transformation for <Policy> P_{id} and add into LP $\mathcal{P}_{P_{id}}$ is as follows (see (7) for <Policy> evaluation)

$$\begin{array}{ll}
\mathsf{val}(P_{id}, \mathsf{i_d}) & \leftarrow \mathsf{val}(T, \mathsf{idt}), \mathsf{algo}(\mathsf{CombID}, P_{id}, \mathsf{d}). \\
\mathsf{val}(P_{id}, \mathsf{i_p}) & \leftarrow \mathsf{val}(T, \mathsf{idt}), \mathsf{algo}(\mathsf{CombID}, P_{id}, \mathsf{p}). \\
\mathsf{val}(P_{id}, \mathsf{na}) & \leftarrow \mathsf{val}(T, \mathsf{nm}). \\
\mathsf{val}(P_{id}, \mathsf{na}) & \leftarrow \mathsf{val}(R_1, \mathsf{na}), \ldots, \mathsf{val}(R_n, \mathsf{na}). \\
\mathsf{val}(P_{id}, E) & \leftarrow \mathsf{val}(T, \mathsf{m}), \mathsf{decision_of}(P_{id}, R, V), V \neq \mathsf{na}, \mathsf{algo}(\mathsf{CombID}, P_{id}, E). \\
\mathsf{val}(P_{id}, E) & \leftarrow \mathsf{val}(T, \mathsf{idt}), \mathsf{decision_of}(P_{id}, R, V), V \neq \mathsf{na}, \mathsf{algo}(\mathsf{CombID}, P_{id}, E), \\
 & \quad E \neq \mathsf{p}. \\
\mathsf{val}(P_{id}, E) & \leftarrow \mathsf{val}(T, \mathsf{idt}), \mathsf{decision_of}(P_{id}, R, V), V \neq \mathsf{na}, \mathsf{algo}(\mathsf{CombID}, P_{id}, E), \\
 & \quad E \neq \mathsf{d}.
\end{array}$$

We write formulae $\mathsf{decision\_of}(P_{id}, R, V), V \neq \mathsf{na}$ to make sure that there is a <Rule> in the <Policy> that is not evaluated to na. We do this to avoid a return value from a combining algorithm that is not naeven tough all of the <Rule> elements are evaluated to na.

<PolicySet> Transformation. The transformation of <PolicySet> is similar to the transformation of <Policy> component.

XACML Syntax: Let $PS_{id} = [T, \langle P_1, \ldots, P_n \rangle, \mathsf{CombID}]$ be a <Policy> component where T is a <Target>, $\langle P_1, \ldots, P_n \rangle$ a sequence of <Policy> (or <PolicySet>) elements and CombID be a combining algorithm identifier.

The transformation of <PolicySet> PS_{id} into logic program $\mathcal{PS}_{P_{id}}$ is as follows: first, for every <Policy> (or <PolicySet>) contained in $PS_{id} = [T, \langle P_1, \ldots, P_n \rangle, \mathsf{CombID}]$, $\mathcal{P}_{PS_{id}}$ also contains:

$$\mathsf{decision\_of}(PS_{id}, P_i, E) \leftarrow \mathsf{val}(P_i, E).\ (1 \leq i \leq n)$$

We also add the following rules into $\mathcal{P}_{PS_{id}}$:

$$\begin{array}{rl}
\mathsf{val}(PS_{id}, \mathsf{i_d}) & \leftarrow \mathsf{val}(T, \mathsf{idt}), \mathsf{algo}(\mathsf{CombID}, PS_{id}, \mathsf{d}). \\
\mathsf{val}(PS_{id}, \mathsf{i_p}) & \leftarrow \mathsf{val}(T, \mathsf{idt}), \mathsf{algo}(\mathsf{CombID}, PS_{id}, \mathsf{p}). \\
\mathsf{val}(PS_{id}, \mathsf{na}) & \leftarrow \mathsf{val}(T, \mathsf{nm}). \\
\mathsf{val}(PS_{id}, \mathsf{na}) & \leftarrow \mathsf{val}(P_1, \mathsf{na}), \ldots, \mathsf{val}(P_n, \mathsf{na}). \\
\mathsf{val}(PS_{id}, E) & \leftarrow \mathsf{val}(T, \mathsf{m}), \mathsf{decision_of}(PS_{id}, P, V), V \neq \mathsf{na}, \\
& \quad \mathsf{algo}(\mathsf{CombID}, PS_{id}, E). \\
\mathsf{val}(PS_{id}, E) & \leftarrow \mathsf{val}(T, \mathsf{idt}), \mathsf{decision_of}(PS_{id}, P, V), V \neq \mathsf{na}, \\
& \quad \mathsf{algo}(\mathsf{CombID}, PS_{id}, E), E \neq \mathsf{p}. \\
\mathsf{val}(PS_{id}, E) & \leftarrow \mathsf{val}(T, \mathsf{idt}), \mathsf{decision_of}(PS_{id}, P, V), V \neq \mathsf{na}, \\
& \quad \mathsf{algo}(\mathsf{CombID}, PS_{id}, E), E \neq \mathsf{d}.
\end{array}$$

3.3 Combining Algorithm Transformation

We use P for an variable of <Policy> identifier and R, R_1 and R_2 for variables of <Rule> identifiers. In case the evaluation of <PolicySet>, the input P is for <PolicySet> identifier, R, R_1 and R_2 are for <Policy> (or <PolicySet>) identifiers.

Permit-Overrides Transformation. Let $\mathcal{P}_{\mathsf{po}}$ be a LP obtained by permit-overrides combining algorithm transformation (see (9) for the permit-overrides combining algorithm semantics). $\mathcal{P}_{\mathsf{po}}$ contains:

$$
\begin{array}{ll}
\mathsf{algo}(\mathsf{po}, P, \mathsf{p}) & \leftarrow \mathsf{decision\_of}(P, R, \mathsf{p}). \\
\mathsf{algo}(\mathsf{po}, P, \mathsf{i_{dp}}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{p}), \mathsf{decision\_of}(P, R, \mathsf{i_{dp}}). \\
\mathsf{algo}(\mathsf{po}, P, \mathsf{i_{dp}}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{p}), \mathsf{decision\_of}(P, R_1, \mathsf{i_p}), \mathsf{decision\_of}(P, R_2, \mathsf{d}). \\
\mathsf{algo}(\mathsf{po}, P, \mathsf{i_{dp}}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{p}), \mathsf{decision\_of}(P, R_1, \mathsf{i_p}), \mathsf{decision\_of}(P, R_2, \mathsf{i_d}). \\
\mathsf{algo}(\mathsf{po}, P, \mathsf{i_p}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{p}), \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{i_{dp}}), \mathsf{decision\_of}(P, R, \mathsf{i_p}). \\
\mathsf{algo}(\mathsf{po}, P, \mathsf{d}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{p}), \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{i_{dp}}), \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{i_p}), \\
 & \quad \mathsf{decision\_of}(P, R, \mathsf{d}). \\
\mathsf{algo}(\mathsf{po}, P, \mathsf{i_d}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{p}), \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{i_{dp}}), \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{i_p}), \\
 & \quad \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{d}), \mathsf{decision\_of}(P, R, \mathsf{i_d}). \\
\mathsf{algo}(\mathsf{po}, P, \mathsf{na}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{p}), \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{i_{dp}}), \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{i_p}), \\
 & \quad \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{d}), \textbf{not}\ \mathsf{algo}(\mathsf{po}, P, \mathsf{i_d}).
\end{array}
$$

First-Applicable Transformation. Let $\mathcal{P}_{\mathsf{fa}}$ be a logic program obtained by first-applicable combining algorithm transformation (see (10) for the first-applicable combining algorithm semantics). For each <Policy> (or <PolicySet>) which uses first-applicable combining algorithm, $P_{id} = [T, \langle R_1, \ldots, R_n \rangle, \mathsf{fa}]$, $\mathcal{P}_{P_{id}}$ contains:

$$
\begin{array}{ll}
\mathsf{algo}(\mathsf{fa}, P, E) & \leftarrow \mathsf{decision\_of}(P, R_1, E), E \neq \mathsf{na}. \\
\mathsf{algo}(\mathsf{fa}, P, E) & \leftarrow \mathsf{decision\_of}(P, R_1, \mathsf{na}), \mathsf{decision\_of}(P, R_2, E), E \neq \mathsf{na}. \\
 & \vdots \\
\mathsf{algo}(\mathsf{fa}, P, E) & \leftarrow \mathsf{decision\_of}(P, R_1, \mathsf{na}), \ldots, \mathsf{decision\_of}(P, R_{n-1}, \mathsf{na}), \\
 & \quad \mathsf{decision\_of}(P, R_n, E).
\end{array}
$$

Only-One-Applicable Transformation. Let $\mathcal{P}_{\mathsf{ooa}}$ be a logic program obtained by only-one-applicable combining algorithm transformation (see (11) for the only-one-applicable combining algorithm semantics). $\mathcal{P}_{\mathsf{ooa}}$ contains:

$$
\begin{array}{ll}
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}) & \leftarrow \mathsf{decision\_of}(P, R, \mathsf{i_{dp}}). \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}) & \leftarrow \mathsf{decision\_of}(P, R_1, \mathsf{i_d}), \mathsf{decision\_of}(P, R_2, \mathsf{i_p}), R_1 \neq R_2. \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}) & \leftarrow \mathsf{decision\_of}(P, R_1, \mathsf{i_d}), \mathsf{decision\_of}(P, R_2, \mathsf{p}), R_1 \neq R_2. \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}) & \leftarrow \mathsf{decision\_of}(P, R_1, \mathsf{d}), \mathsf{decision\_of}(P, R_2, \mathsf{i_p}), R_1 \neq R_2. \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}) & \leftarrow \mathsf{decision\_of}(P, R_1, \mathsf{d}), \mathsf{decision\_of}(P, R_2, \mathsf{p}), R_1 \neq R_2. \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_p}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}), \mathsf{decision\_of}(P, R, \mathsf{i_d}). \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_p}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}), \mathsf{decision\_of}(P, R_1, \mathsf{p}), \\
 & \quad \mathsf{decision\_of}(P, R_2, \mathsf{p}), R_1 \neq R_2. \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_d}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}), \mathsf{decision\_of}(P, R, \mathsf{i_d}). \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_d}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}), \mathsf{decision\_of}(P, R_1, \mathsf{d}), \\
 & \quad \mathsf{decision\_of}(P, R_2, \mathsf{d}), R_1 \neq R_2. \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{p}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}), \textbf{not}\ (\mathsf{ooa}, P, \mathsf{i_d}), \\
 & \quad \textbf{not}\ (\mathsf{ooa}, P, \mathsf{i_p}), \mathsf{decision\_of}(P, R, \mathsf{p}). \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{d}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}), \textbf{not}\ (\mathsf{ooa}, P, \mathsf{i_d}), \textbf{not}\ (\mathsf{ooa}, P, \mathsf{i_p}), \\
 & \quad \mathsf{decision\_of}(P, R, \mathsf{d}). \\
\mathsf{algo}(\mathsf{ooa}, P, \mathsf{na}) & \leftarrow \textbf{not}\ \mathsf{algo}(\mathsf{ooa}, P, \mathsf{i_{dp}}), \textbf{not}\ (\mathsf{ooa}, P, \mathsf{i_d}), \textbf{not}\ (\mathsf{ooa}, P, \mathsf{i_p}), \\
 & \quad \textbf{not}\ \mathsf{decision\_of}(P, R, \mathsf{d}), \textbf{not}\ \mathsf{decision\_of}(P, R, \mathsf{p}).
\end{array}
$$

4 Relation between XACML-ASP and XACML 3.0 Semantics

In this section we discuss the relation between the ASP semantics and XACML 3.0 semantics. First we recall the semantics of logic programs based on their answer sets then we show a program $\mathcal{P}_{XACML}$ – the program obtained from tranforming XACML into LP – has a unique answer set and the answer set shows the semantics of XACML 3.0.

4.1 ASP Semantics

The declarative semantics of a logic program is given by a model-theoretic semantics of formulae in the underlying language. We use answer set semantics for LP $\mathcal{P}_{XACML}$. The formal definition of answer set semantics can be found in many literatures like in [3,5].

The answer set semantics of logic program $\mathcal{P}$ assigns to $\mathcal{P}$ a collection of *answer sets* – interpretations of $ground(\mathcal{P})$. An interpretation I of $ground(\mathcal{P})$ is an answer set for $\mathcal{P}$ if I is minimal (w.r.t. set inclusion) among the interpretations satisfying the rules of

$$\mathcal{P}^I = \{A \leftarrow B_1, \ldots, B_m | \; A \leftarrow B_1, \ldots, B_m, \mathbf{not}\ B_{m+1}, \ldots, \mathbf{not}\ B_n \in \mathcal{P} \text{ and } I(\mathbf{not}\ B_{m+1}, \ldots, \mathbf{not}\ B_n) = true\}$$

A logic program can have a unique, many or none answer set(s). Therefore, we show that programs with a particular characteristic are guaranteed to have unique answer set.

Acyclic Programs. We say that a program is *acyclic* when there is no cycle in the program.The acyclicity in the program is guaranteed by the existence of a certain fixed assignment of natural numbers to atoms that is called a *level mapping*.

A *level mapping* for a program $\mathcal{P}$ is a function

$$l : \mathcal{B}_\mathcal{P} \rightarrow \mathbf{N}$$

where $\mathbf{N}$ is the set of natural numbers and $\mathcal{B}_\mathcal{P}$ is the Herbrand base for $\mathcal{P}$. We extend the definition of level mapping to a mapping from ground literals to natural numbers by setting $l(\mathbf{not}\ A) = l(A)$.

Let $\mathcal{P}$ be a logic program and l be a level mapping for $\mathcal{P}$. $\mathcal{P}$ is *acyclic with respect to* l if for every clause $A \leftarrow B_1, \ldots, B_m, \mathbf{not}\ B_{m+1}, \ldots, \mathbf{not}\ B_n$ in $ground(\mathcal{P})$ we find

$$l(A) > l(B_i) \text{ for all } i \text{ with } 1 \leq i \leq n$$

$\mathcal{P}$ is *acyclic* if it is acyclic with respect to some level mapping.

Acyclic programs are guaranteed to have unique answer sets [3].

4.2 XACML Semantics Based On ASP Semantics

We can see from Sect. 3 that all of the XACML 3.0 transformation programs are acyclic. Thus, it is guaranteed that $\mathcal{P}_{XACML}$ has unique answer set.

Conjecture 1. Let $\mathcal{P}_{XACML}$ be a program obtained from XACML 3.0 element transformations and let $\mathcal{P}_Q$ be a program transformation of `<Request>` $\mathcal{Q}$. Let I be the answer set of $\mathcal{P}_{XACML} \cup \mathcal{P}_Q$. Then the following equation holds

$$[\![X]\!](\mathcal{Q}) = V \text{ iff } \mathsf{val}(X, V) \in I$$

5 Analysis XACML Policies Using Answer Set Programming

In most cases, ASP solver can solve efficiently combinatorial problems. There are several combinatorial problems in analysis access control policies, e.g. gap-free property and conflict-free property [11,4]. In this section we consider gap-free analysis since in XACML 3.0 conflict never occurs[4]. We also present a mechanism of how to verify security properties against a set of access control policies.

5.1 Gap-Free Analysis

A set of policies is *gap-free* if there is no access request for which there is an absence of decision. In order to make sure that a set of policies is gap-free we should generate all possible requests and test whether there is at least one request that is not captured by the set of policies.

We use *cardinality constraint* (see [12,13]) to generate all possible values restored in the database for each attribute.

$$\begin{array}{ll}
\mathcal{P}_{generate}: & \\
1\{subject(X) : subject_db(X)\}1 & \leftarrow \top. \\
1\{action(X) : action_db(X)\}1 & \leftarrow \top. \\
1\{resource(X) : resource_db(X)\}1 & \leftarrow \top. \\
1\{environment(X) : environment_db(X)\}1 & \leftarrow \top.
\end{array}$$

The first line of the encoding means that we only consider only one $subject$ attribute value obtained from the subject database. The rest of the encoding means the same as the $subject$ attribute.

XACML defines that there is one `<PolicySet>` as the root of a set of policies. Hence, we say there is a gap whenever we can find a request that makes value of the PS_{root} is na. We force ASP solver to find the gap.

$$\begin{array}{ll}
\mathcal{P}_{gap}: & \\
gap & \leftarrow \mathsf{val}(PS_{root}, \mathsf{na}). \\
\bot & \leftarrow \mathbf{not}\ gap.
\end{array}$$

The answer sets of program $\mathcal{P} = \mathcal{P}_{XACML} \cup \mathcal{P}_{generate} \cup \mathcal{P}_{gap}$ are the witnesses that the set of policies encoded in $\mathcal{P}_{XACML}$ is incomplete. When there is no model satisfies the program then we are sure that the set of policies captures all of possible cases.

[4] A conflict decision never occurs when we strict to using the standard combining algorithm defined in XACML 3.0 since every combining algorithm always return one value

5.2 Property Analysis

The problem of verifying a security property F on XACML policies is not only to show that the property F holds on $\mathcal{P}_{XACML}$ but also we want to see the witnesses whenever the property F does not hold in order to help the policy developer refine the policies. Thus, we can see this problem as finding models for $\mathcal{P}_{XACML} \cup \mathcal{P}_{generate} \cup \mathcal{P}_{\neg F}$. The finding model is the witness that the XACML policies cannot capture that property F.

Example Suppose we have a security property:

F: An anonymous person **cannot** read any patient records.

Thus, the negation of property F is as follows

$\neg F$: An anonymous person **can** read any patient records.

When we define that anonymous persons are those who are neither patients, nor guardians, nor doctors and nor nurses we have a refined property F':

$\neg F'$: not patients, not guardians, not doctors and not nurses **can** read any patient records.

We encode $\mathcal{P}_{\neg F'}$ as follows

$$
\begin{array}{ll}
(1)\ \bot & \leftarrow subject(patient). \\
(2)\ \bot & \leftarrow subject(guardian). \\
(3)\ \bot & \leftarrow subject(doctor). \\
(4)\ \bot & \leftarrow subject(nurse). \\
(5)\ action(read) & \leftarrow \top. \\
(6)\ resource(patient\_record) & \leftarrow \top. \\
(7)\ \bot & \leftarrow \mathbf{not}\ \mathsf{val}(PS_{root}, \mathsf{p}).
\end{array}
$$

First of all we list all of the requirements (line 1 – 6). Later we force that the returned decision should be permit (line 7). When the program $\mathcal{P}_{XACML} \cup \mathcal{P}_{generate} \cup \mathcal{P}_{\neg F}$ returns models, we conclude that the property F does not hold and the returned models are the flaws in the policies.

6 Conclusion and Future Work

We have modelled XACML Policy Decision Point in a declarative way using ASP technique by transforming XACML 3.0 elements into logic programs. Our transformation of XACML 3.0 elements is directly based on XACML 3.0 semantics [8] and we have shown that the answer set of each program transformation is unique and it agrees with the semantics of XACML 3.0. Moreover, we can help policy developers to analyse their access control policies such as checking policies completeness and verifying policy properties by inspecting the answer set of $\mathcal{P}_{XACML} \cup \mathcal{P}_{configuration}$ – the program obtained by transforming XACML 3.0 elements into logic programs joined with configuration program.

For future work, we can extend our work to handle role-based access control in XACML 3.0 [10] and to handle delegation in XACML 3.0 [9]. We also can extend our work for checking reachability of policies. A policy is reachable if there is a request such that the decision is made based on the applicability of that policy. We are safe to remove unreachable policies.

References

1. Gail-Joon Ahn, Hongxin Hu, Joohyung Lee, and Yunsong Meng. Reasoning about xacml policy descriptions in answer set programming (preliminary report). In *13th International Workshop on Nonmonotonic Reasoning (NMR 2010)*, 2010.
2. Gail-Joon Ahn, Hongxin Hu, Joohyung Lee, and Yunsong Meng. Representing and reasoning about web access control policies. In Sheikh Iqbal Ahamed, Doo-Hwan Bae, Sung Deok Cha, Carl K. Chang, Rajesh Subramanyan, Eric Wong, and Hen-I Yang, editors, *COMPSAC*, pages 137–146. IEEE Computer Society, 2010.
3. Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, February 2003.
4. Glenn Bruns and Michael Huth. Access-control via belnap logic: Effective and efficient composition and analysis. In *21st IEEE Computer Security Foundations Symposium*, June 2008.
5. Michael Gelfond. Handbook of knowledge representation. In B. Porter F. van Harmelen, V. Lifschitz, editor, *Foundations of Artificial Intelligence*, volume 3, chapter Answer Sets, pages 285–316. Elsevier, 2007.
6. Tim Moses. eXtensible Access Control Markup Language (XACML) version 2.0. Technical report, OASIS, http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf, August 2010.
7. Carroline Dewi Puspa Kencana Ramli, Hanne Riis Nielson, and Flemming Nielson. The logic of xacml. In *FACS 2011 - 8th International Symposium on Formal Aspects of Component Software - Oslo, Norway, September 14-16, 2011. Proceedings*, Lecture Notes in Computer Science, 2011.
8. Erik Rissanen. eXtensible Access Control Markup Language (XACML) version 3.0 (committe specification 01). Technical report, OASIS, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-cd-03-en.pdf, August 2010.
9. Erik Rissanen. Xacml v3.0 administration and delegation profile version 1.0 (committe specification 01). Technical report, OASIS, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-administration-v1-spec-cs-01-en.pdf, August 2010.
10. Erik Rissanen. Xacml v3.0 core and hierarchical role based access control (rbac) profile version 1.0 (committe specification 01). Technical report, OASIS, http://docs.oasis-open.org/xacml/3.0/xacml-3.0-rbac-v1-spec-cs-01-en.pdf, August 2010.
11. Pierangela Samarati and Sabrina de Capitani di Vimercati. Access control: Policies, models, and mechanisms. In *Foundations of Security Analysis and Design, Tutorial Lectures*, volume 2171 of *Lecture Notes in Computer Science*, pages 137–196. Springer Verlag, 2001.
12. Patrik Simons, Ilkka Niemelá, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, June 2002.
13. Tommi Syrjänen. *Lparse 1.0 User's Manual*.

Types vs. PDGs in Information Flow Analysis

Heiko Mantel and Henning Sudbrock
Computer Science Department, TU Darmstadt, Germany
{mantel,sudbrock}@mais.informatik.tu-darmstadt.de

Abstract Type-based and PDG-based information flow analysis techniques are currently developed independently in a competing manner, with different strengths regarding coverage of language features and security policies. In this article, we study the relationship between these two approaches. One key insight is that a type-based information flow analysis need not be less precise than a PDG-based analysis. For proving this result we establish a formal connection between the two approaches which can also be used to transfer concepts from one tradition of information flow analysis to the other. The adoption of rely-guarantee-style reasoning from security type systems, for instance, enabled us to develop a PDG-based information flow analysis for multi-threaded programs.
Keywords: Information flow security, Security type system, PDG

1 Introduction

When giving a program access to confidential data one wants to be sure that the program does not leak any secrets to untrusted sinks, like, e.g., to untrusted servers on the Internet. Such confidentiality requirements can be characterized by information flow properties. For verifying that a program satisfies an information flow property, a variety of program analysis techniques can be employed.

The probably most popular approach to information flow analysis is the use of security type systems. Starting with [VSI96], type-based information flow analyses were developed for programs with various language features comprising procedures (e.g., [VS97]), concurrency (e.g., [SV98]), and objects (e.g., [Mye99]). Security type systems were proposed for certifying a variety of information flow properties, including timing-sensitive and timing-insensitive properties (e.g., [SS00] and [BC02]) and properties supporting declassification (e.g., [MS04]).

Besides type systems, one can also employ other program analysis techniques for certifying information flow security. For instance, it was proposed in [HUM92] to use program dependency graphs (PDGs) for information flow analysis. A PDG [FOW87] is a graph-based program representation that captures dependencies caused by the data flow and the control flow of a program. PDG-based information flow analyses recently received new attention, resulting in, e.g., a PDG-based information flow analysis for object-oriented programs and a PDG-based information flow analysis supporting declassification [HS09,Ham09].

Type-based and PDG-based information flow analyses are currently developed independently. The two sub-communities both see potential in their approach, but the pros and cons of the two techniques have not been compared in detail. In this article, we compare type-based and PDG-based information flow analyses with respect to their precision. Outside the realm of information flow

security there already exist results that compare the precision of data-flow oriented and type-based analyses, for instance, for safety properties [PO95,NP08]. Here, we clarify the relation between type-based and PDG-based analyses in the context of information flow security. We investigate whether (a) one approach has superior precision, (b) both have pros and cons, or (c) both are equally precise. To be able to establish a precise relation, we consider two prominent analyses that are fully formalized, namely the type-based analysis from Hunt and Sands [HS06] and the PDG-based analysis from Wasserrab et al [WLS09].

Our main result is that the two analyses have exactly the same precision. This result was surprising for us, because one motivation for using PDGs in an information flow analysis was their precision [HS09]. We derive our main result based on a formal connection between the two kinds of security analyses, which we introduce in this article. It turned out that this connection is also interesting in its own right, because it can be used for transferring ideas from type-based to PDG-based information flow analyses and vice versa. In this article, we illustrate this possibility in one direction, showing how to derive a novel PDG-based information flow analysis that is suitable for multi-threaded programs by exploiting our recently proposed solution for rely-guarantee-style reasoning in a type-based security analysis [MSS11]. The resulting analysis is compositional and, thereby, enables a modular security analysis. This is an improvement over the analysis from [GS12], the only provably sound PDG-based information flow analysis for multi-threaded programs developed so far. Moreover, in contrast to [GS12] our novel analysis supports programs with nondeterministic public output.

In summary, the main contributions of this article are

1. the formal comparison of the precision of a type-based and a PDG-based information flow analysis, showing that they have the same precision;
2. the demonstration that our formal connection between the type-based and the PDG-based analysis can be used to transfer concepts from one approach to the other (by transferring rely-guarantee-style reasoning as mentioned above); and
3. a provably sound compositional PDG-based information flow analysis for multi-threaded programs that is compatible with nondeterministic public output.

We believe that the connection between type- and PDG-based information flow analysis can serve as a basis for further mutual improvements of the analysis techniques. Such a transfer is desirable because there are other relevant aspects than an analysis' precision like, e.g., efficiency and availability of tools. Moreover, we hope that the connection between the two approaches to information flow analysis fosters mutual understanding and interaction between the two communities.

2 Type-based Information Flow Analyses

If one grants a program access to secret information, one wants to be sure that the program does not leak secrets to untrusted sinks like, e.g., untrusted servers in a network. A secure program should not only refrain from directly copying secrets to untrusted sinks (as, e.g., with an assignment "$sink{:=}secret$"), but also should not reveal secrets indirectly (as, e.g., by executing "$\mathsf{if}\ (secret > 0)\ \mathsf{then}\ sink{:=}1$").

It is popular to formalize information flow security by the property that values written to public sinks do not depend on secrets, the probably best known

such property being *Noninterference* [GM82,Man11]. In the following, we define information flow security for programs by such a property, and we present a security type system for certifying programs with respect to this property.

2.1 Execution Model and Security Property

We consider a set of *commands Com* that is defined by the grammar

$$c ::= \mathsf{skip} \mid x{:=}e \mid c; c \mid \mathsf{if}\ (e)\ \mathsf{then}\ c\ \mathsf{else}\ c\ \mathsf{fi} \mid \mathsf{while}\ (e)\ \mathsf{do}\ c\ \mathsf{od},$$

where $x \in \mathit{Var}$ is a variable and $e \in \mathit{Exp}$ is an expression. *Expressions* are terms built from variables and from operators that we do not specify further. The set of *free variables* in expression $e \in \mathit{Exp}$ is denoted with $\mathit{fv}(e)$. A *memory* is a function $\mathit{mem} : \mathit{Var} \to \mathit{Val}$ that models a snapshot of a program's memory, where *Val* is a set of *values* and $\mathit{mem}(x)$ is the *value of* x. Judgments of the form $\langle c, \mathit{mem}\rangle \Downarrow \mathit{mem}'$ model program execution, with the interpretation that command c, if executed with initial memory *mem*, terminates with memory mem'. The rules for deriving the judgments are as usual for big-step semantics.[1]

To define the security property, we consider a *security lattice* $\mathcal{D} = \{l, h\}$ with two *security domains* where $l \sqsubseteq h$ and $h \not\sqsubseteq l$. This models the requirement that no information flows from domain h to domain l. This is the simplest policy capturing information flow security.[2] A *domain assignment* is a function $\mathit{dom} : \mathit{Var} \to \mathcal{D}$ that associates a security domain with each program variable. We say that variables in the set $L = \{x \in \mathit{Var} \mid \mathit{dom}(x) = l\}$ are public or *low*, and that variables in the set $H = \{x \in \mathit{Var} \mid \mathit{dom}(x) = h\}$ are secret or *high*. The resulting security requirement is that the final values of low variables do not depend on the initial values of high variables. This requirement captures security with respect to an attacker who sees the initial and final values of low variables, but cannot access values of high variables (i.e., access control works correctly).

Definition 1. *Two memories mem and* mem' *are* low-equal *(written* $\mathit{mem} =_L \mathit{mem}'$*) if and only if* $\mathit{mem}(x) = \mathit{mem}'(x)$ *for all* $x \in L$.

A command c is noninterferent *if whenever* $\mathit{mem}_1 =_L \mathit{mem}_2$ *and* $\langle c, \mathit{mem}_1\rangle \Downarrow \mathit{mem}_1'$ *and* $\langle c, \mathit{mem}_2\rangle \Downarrow \mathit{mem}_2'$ *are derivable then* $\mathit{mem}_1' =_L \mathit{mem}_2'$.

Example 1. Consider command c to the right and assume that $\mathit{dom}(x) = \mathit{dom}(y) = l$ and $\mathit{dom}(z) = h$. Consider furthermore low-equal memories mem_1 and mem_2 with $\mathit{mem}_1(x) = \mathit{mem}_2(x) = \mathit{mem}_1(y) = \mathit{mem}_2(y) = 1$, $\mathit{mem}_1(z) = -1$, and $\mathit{mem}_2(z) = 1$.

```
1. if (z < 0) then
2.   while (y > 0) do
3.     y:=y + z od else
4.   skip fi;
5. x:=y
```

Then $\langle c, \mathit{mem}_1\rangle \Downarrow \mathit{mem}_1'$ and $\langle c, \mathit{mem}_2\rangle \Downarrow \mathit{mem}_2'$ are derivable with $\mathit{mem}_1'(x) = 0$ and $\mathit{mem}_2'(x) = 1$. Since $\mathit{mem}_1' \neq_L \mathit{mem}_2'$ command c is not noninterferent.

2.2 The Type-based Information Flow Analysis by Hunt and Sands

A type-based analysis uses a collection of typing rules to inductively define a subset of programs. The intention is that every program in the subset satisfies

[1] For the rules and proofs of theorems in this article we refer to the authors' website.
[2] The results in this article can be lifted to other security lattices.

$$[\mathsf{exp}]\frac{}{\Gamma \vdash e : \bigsqcup_{x \in fv(e)} \Gamma(x)} \qquad [\mathsf{if}]\frac{\Gamma \vdash e : t \quad pc \sqcup t \vdash \Gamma\,\{c_1\}\,\Gamma_1' \quad pc \sqcup t \vdash \Gamma\,\{c_2\}\,\Gamma_2'}{pc \vdash \Gamma\,\{\mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\}\,\Gamma_1' \sqcup \Gamma_2'}$$

$$[\mathsf{assign}]\frac{\Gamma \vdash e : t}{pc \vdash \Gamma\,\{x{:=}e\}\,\Gamma[x \mapsto pc \sqcup t]} \qquad [\mathsf{seq}]\frac{pc \vdash \Gamma\,\{c_1\}\,\Gamma' \quad pc \vdash \Gamma'\,\{c_2\}\,\Gamma''}{pc \vdash \Gamma\,\{c_1; c_2\}\,\Gamma''}$$

$$[\mathsf{skip}]\frac{}{pc \vdash \Gamma\,\{\mathsf{skip}\}\,\Gamma} \qquad [\mathsf{while}]\frac{\begin{array}{c}\Gamma_i' \vdash e : t_i \quad pc \sqcup t_i \vdash \Gamma_i'\,\{c\}\,\Gamma_i'' \quad 0 \le i \le k \\ \Gamma_0' = \Gamma \quad \Gamma_{i+1}' = \Gamma_i'' \sqcup \Gamma \quad \Gamma_{k+1}' = \Gamma_k'\end{array}}{pc \vdash \Gamma\,\{\mathsf{while}\ (e)\ \mathsf{do}\ c\ \mathsf{od}\}\,\Gamma_k'}$$

Figure 1. Type system from [HS06]

an information flow property like, e.g., the one from Definition 1. Starting with [VSI96], many type-based information flow analyses were developed (see [SM03] for an overview). Here, we recall the type system from Hunt and Sands [HS06] which is, unlike many other type-based security analyses, flow-sensitive (i.e., it takes the order of program statements into account to improve precision).

In [HS06], typing judgments have the form $pc \vdash \Gamma\,\{c\}\,\Gamma'$, where c is a command, $\Gamma, \Gamma' : \mathit{Var} \to \mathcal{D}$ are *environments*, and pc is a security domain. The interpretation of the judgment is as follows: For each variable $y \in \mathit{Var}$, $\Gamma'(y)$ is a valid upper bound on the security level of the value of y after command c has been run if (a) for each $x \in \mathit{Var}$, $\Gamma(x)$ is an upper bound on the security level of the value of x before c has been run and (b) pc is an upper bound on the security level of all information on which it might depend whether c is run.

The typing rules from [HS06] are displayed in Figure 1, where $\sqcup$ denotes the least upper bound operator on $\mathcal{D}$, which is extended to environments by $(\Gamma \sqcup \Gamma')(x) := \Gamma(x) \sqcup \Gamma'(x)$. The rules ensure that for any given c, Γ, and pc there is an environment Γ' such that $pc \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable. Moreover, this environment is uniquely determined by c, Γ, and pc [HS06, Theorem 4.1].

Definition 2. *Let $c \in \mathit{Com}$, $\Gamma(x) = \mathit{dom}(x)$ for all $x \in \mathit{Var}$, and Γ' be the unique environment such that $l \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable. Command c is* accepted by the type-based analysis *if $\Gamma'(x) \sqsubseteq \mathit{dom}(x)$ for all $x \in \mathit{Var}$.*

Example 2. Consider command c and domain assignment *dom* from Example 1. Let $\Gamma(x) = \mathit{dom}(x)$ for all $x \in \mathit{Var}$. Then the judgment $l \vdash \Gamma\,\{c\}\,\Gamma'$ is derivable if and only if $\Gamma'(x) = \Gamma'(y) = \Gamma'(z) = h$ and $\Gamma'(x') = \Gamma(x')$ for all other $x' \in \mathit{Var}$. Since $\Gamma'(x) \not\sqsubseteq \mathit{dom}(x)$ command c is not accepted by the type-based analysis.

Theorem 1. *Commands accepted by the type-based analysis are noninterferent.*

The theorem follows from Theorem 3.3 in [HS06].

3 PDG-based Information Flow Analyses

PDG-based information flow analyses, firstly proposed in [HUM92], exploit that the absence of certain paths in a *program dependency graph* (PDG) [FOW87] is a sufficient condition for the information flow security of a program. In this

section, we recall the PDG-based analysis from [WLS09] which is sound with respect to the property from Definition 1. In order to be self-contained, we recall the construction of control flow graphs (CFGs) and PDGs in Sections 3.1 and 3.2.

3.1 Control Flow Graphs

Definition 3. *A* directed graph *is a pair* (N, E) *where* N *is a set of nodes and* $E \subseteq N \times N$ *is a set of edges. A* path p from node n_1 to node n_k *is a non-empty sequence of nodes* $\langle n_1, \ldots, n_k \rangle \in N^+$ *where* $(n_i, n_{i+1}) \in E$ *for all* $i \in \{1, \ldots, k-1\}$. *We say that node* n *is* on the path $\langle n_1, \ldots, n_k \rangle$ *if* $n = n_i$ *for some* $i \in \{1, \ldots, k\}$.

Definition 4. *A* control flow graph with def and use sets *is a tuple* (N, E, def, use) *where* (N, E) *is a directed graph,* N *contains two distinguished nodes start and stop, and* $def, use : N \to \mathcal{P}(Var)$ *are functions returning the def and use set, respectively, for a node. (The set* $\mathcal{P}(Var)$ *denotes the powerset of the set* Var*.)*

Nodes *start* and *stop* represent program start and termination, respectively, and the remaining nodes represent program statements and control conditions. An edge $(n, n') \in E$ models that n' might immediately follow n in a program run. Finally, the sets $def(n)$ and $use(n)$ contain all variables that are defined and used, respectively, at a node n. In the remainder of this article we simply write "CFG" instead of "CFG with def and use sets."

We recall the construction of the CFG for a command following [WL08], where statements and control conditions are represented by numbered nodes.

Definition 5. *We denote with* $|c|$ *the number of statements and control conditions of* $c \in Com$*, and define* $|c|$ *recursively by* $|\mathsf{skip}| = 1$, $|x{:=}e| = 1$, $|c_1; c_2| = |c_1| + |c_2|$, $|\mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}| = 1 + |c_1| + |c_2|$, *and* $|\mathsf{while}\ (e)\ \mathsf{do}\ c\ \mathsf{od}| = 1 + |c|$.

Definition 6. *For* $c \in Com$ *and* $1 \leq i \leq |c|$ *we denote with* $c[i]$ *the* i^{th} *statement or control condition in* c*, which we define recursively as follows: If* $c = \mathsf{skip}$ *or* $c = x{:=}e$ *then* $c[1] = c$. *If* $c = c_1; c_2$ *then* $c[i] = c_1[i]$ *for* $1 \leq i \leq |c_1|$ *and* $c[i] = c_2[i - |c_1|]$ *for* $|c_1| < i \leq |c|$. *If* $c = \mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}$ *then* $c[1] = e$, $c[i] = c_1[i{-}1]$ *for* $1 < i \leq 1 + |c_1|$, *and* $c[i] = c_2[i{-}1{-}|c_1|]$ *for* $1 + |c_1| < i \leq |c|$. *If* $c = \mathsf{while}\ (e)\ \mathsf{do}\ c_1\ \mathsf{od}$ *then* $c[1] = e$ *and* $c[i] = c_1[i{-}1]$ *for* $1 < i \leq |c|$.

Note that the ith statement or control condition, i.e., $c[i]$, is either an expression, an assignment, or a skip-statement.

Definition 7. *For* $c \in Com$, $N_c = \{1, \ldots, |c|\} \cup \{start, stop\}$.

We define an operator $\ominus : (\mathbb{N} \cup \{start, stop\}) \times \mathbb{N} \to \mathbb{Z} \cup \{start, stop\}$ by $n \ominus z = n - z$ if $n \in \mathbb{N}$ and $n \ominus z = n$ if $n \in \{start, stop\}$.

Definition 8. *For* $c \in Com$ *the set* $E_c \subseteq N_c \times N_c$ *is defined recursively by:*

- $E_{\mathsf{skip}} = E_{x:=e} = \{(start, 1), (1, stop), (start, stop)\}$,
- $E_{\mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}} = \{(start, 1)\} \cup \{(1, n') \mid (start, n' \ominus 1) \in E_{c_1}\} \cup$
  $\{(1, n') \mid (start, n' \ominus (1 + |c_1|)) \in E_{c_2}\} \cup$
  $\{(n, n') \mid [(n \ominus 1, n' \ominus 1) \in E_{c_1} \vee (n \ominus (1 + |c_1|), n' \ominus (1 + |c_1|)) \in E_{c_2}] \wedge n \neq start\}$,

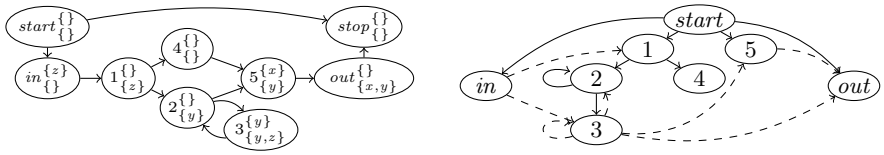


Figure 2. The CFG and the PDG for the command from Example 1

– $E_{c_1;c_2} = \{(n, n') \mid (n, n') \in E_{c_1} \wedge n' \neq stop\} \cup$
$\{(n, n') \mid (n \ominus |c_1|, n' \ominus |c_1|) \in E_{c_2} \wedge n \neq start\} \cup$
$\{(n, n') \mid (n, stop) \in E_{c_1} \wedge (start, n' \ominus |c_1|) \in E_{c_2}\}$, *and*
– $E_{\mathsf{while}\ (e)\ \mathsf{do}\ c\ \mathsf{od}} = \{(start, 1)\} \cup$
$\{(1, n') \mid (start, n' \ominus 1) \in E_c\} \cup \{(n', 1) \mid (n' \ominus 1, stop) \in E_c\} \cup$
$\{(n, n') \mid (n \ominus 1, n' \ominus 1) \in E_c \wedge n \neq start \wedge n' \neq stop\}$.

Definition 9. *For $c \in Com$ we define $def_c : N_c \to \mathcal{P}(Var)$ by $def_c(n) = \{x\}$ if $n \in \{1, \ldots, |c|\}$ and $c[n] = x{:=}e$, and by $def_c(n) = \{\}$ otherwise. Moreover, we define $use_c : N_c \to \mathcal{P}(Var)$ by $use_c(n) = fv(e)$ if $n \in \{1, \ldots, |c|\}$ and $c[n] = x{:=}e$ or $c[n] = e$, and by $use_c(n) = \{\}$ otherwise.*

Definition 10. *The* control flow graph of c *is* $CFG_c = (N_c, E_c, def_c, use_c)$.

Note that, by definition, an edge from *start* to *stop* is contained in E_c. This edge models the possibility that c is not executed.

We now augment CFGs with def and use sets by two nodes *in* and *out* to capture the program's interaction with its environment. Two sets of variables $I, O \subseteq Var$, respectively, specify which variables may be initialized by the environment before program execution and which variables may be read by the environment after program execution. This results in the following variant of CFGs:

Definition 11. *Let $CFG = (N, E, def, use)$ and $I, O \subseteq Var$. Then $CFG^{I,O} = (N', E', def', use')$ where $N' = N \cup \{in, out\}$, $E' = \{(start, in), (out, stop)\} \cup \{(in, n') \mid (start, n') \in E\} \cup \{(n, out) \mid (n, stop) \in E\} \cup \{(n, n') \in E \mid n \notin \{start, stop\} \wedge n' \notin \{start, stop\}\}$, $def'(in) = I$, $use'(in) = def'(out) = \{\}$, $use'(out) = O$, and $def'(n) = def(n)$ and $use'(n) = use(n)$ for $n \in N$.*

Definitions 10 and 11 both augment the usual notion of control flow graphs (see Definition 4). In the remainder of this article, we use the abbreviation CFG for arbitrary control flow graphs (including those that satisfy Definition 10 or 11).

We use a graphical representation for displaying CFGs where we depict nodes with ellipses and edges with solid arrows. For each node n we label the corresponding ellipse with n^X_Y where $X = def(n)$ and $Y = use(n)$.

Example 3. Command c in Example 1 contains three statements and two control conditions (i.e., $|c| = 5$). Hence, $N_c = \{1, \ldots, 5, start, stop\}$. Nodes 1–5 represent the statements and control conditions in Lines 1–5 of the program, respectively. The control flow graph $CFG_c^{\{z\},\{x,y\}}$ is displayed at the left hand side of Figure 2.

3.2 The PDG-based Information Flow Analysis by Wasserrab et al

PDGs are directed graphs that represent dependencies in imperative programs [FOW87]. PDGs were extended to programs with various languages features like procedures (e.g., [HRB90]), concurrency (e.g., [Che93]), and objects (e.g., [HS09]). We recall the construction of PDGs from CFGs for the language from Section 2 based on the following notions of data dependency and control dependency.

Definition 12. *Let (N, E, def, use) be a CFG and $n, n' \in N$. If $x \in def(n)$ we say that* the definition of x at n reaches n' *if there is a path p from n to n' such that $x \notin def(n'')$ for every node n'' on p with $n'' \neq n$ and $n'' \neq n'$.*

Node n' is data dependent *on node n if there exists $x \in Var$ such that $x \in def(n)$, $x \in use(n')$, and the definition of x at n reaches n'.*

Intuitively, a node n' is data dependent on a node n if n' uses a variable that has not been overwritten since being defined at n.

Example 4. Consider the CFG on the left hand side of Figure 2. The definition of y at Node 3 reaches Node 5 because $\langle 3, 2, 5\rangle$ is a path and $y \notin def(2)$. Hence, Node 5 is data dependent on Node 3 because $y \in def(3)$, $y \notin def(2)$, and $y \in use(5)$. Note that Node 2 is also data dependent on Node 3, and that Node 3 is data dependent on itself.

Definition 13. *Let (N, E, def, use) be a CFG. Node n'* postdominates *node n if $n \neq n'$ and every path from n to stop contains n'.*

Node n' is control dependent *on node n if there is a path p from n to n' such that n' postdominates all nodes $n'' \notin \{n, n'\}$ on p and n' does not postdominate n.*

Intuitively, a node n' is control dependent on a node n if n represents the innermost control condition that guards the execution of n'.

Example 5. Consider again the CFG in Figure 2. Node 5 postdominates Node 1 because Node 5 is on all paths from Node 1 to Node *stop*. Hence, Node 5 is not control dependent on Node 1. Nodes 2, 3, and 4 do not postdominate Node 1. Node 3 is not control dependent on Node 1 because Node 3 does not postdominate Node 2 and all paths from Node 1 to Node 3 contain Node 2. However, Node 3 is control dependent on Node 2. Moreover, Nodes 2 and 4 are control dependent on Node 1 because $\langle 1, 2\rangle$ and $\langle 1, 4\rangle$ are paths in the CFG.

Definition 14. *Let $CFG = (N, E, def, use)$ be a control flow graph. The directed graph (N', E') is the* PDG of *CFG (denoted with $PDG(CFG)$) if $N' = N$ and $(n, n') \in E'$ if and only if n' is data dependent or control dependent on n in CFG.*

We use the usual graphical representation for displaying PDGs, depicting Node n with an ellipse labeled with n, edges that reflect control dependency with solid arrows, and edges that reflect data dependency with dashed arrows. Moreover, we do not display nodes that have neither in- nor outgoing edges.

Example 6. The PDG of the CFG at the left of Figure 2 is displayed right of the CFG. Node *stop* is not displayed because it has neither in- nor outgoing edges.

The PDG-based information flow analysis from Wasserrab et al [WLS09] for a command c is based on the PDG of $CFG_c^{H,L}$ (cf. Definition 11).

Definition 15. *The command $c \in Com$ is* accepted by the PDG-based analysis *if and only if there is no path from in to out in $PDG(CFG_c^{H,L})$.*

Example 7. For command c and domain assignment dom from Example 1 the graph $PDG(CFG_c^{H,L})$ is displayed at the right of Figure 2. It contains a path from Node in to Node out, (e.g., the path $\langle in, 3, out\rangle$). In consequence, c is not accepted by the PDG-based analysis.

Theorem 2. *Commands accepted by the PDG-based analysis are noninterferent.*

The theorem follows from [WLS09, Theorem 8].

4 Comparing the Type- and the PDG-based Analysis

While both the type-based analysis from Section 2 and the PDG-based analysis from Section 3 are sound, both analyses are also incomplete. I.e., for both analyses there are programs that are noninterferent (according to Definition 1), but that are not accepted by the analysis. A complete analysis is impossible, because the noninterference property is undecidable (this can be proved in a standard way by showing that the decidability of the property would imply the decidability of the halting problem [SM03]). This raises the question if one of the two analyses is more precise than the other. In this section, we answer this question. As an intermediate step, we establish a relation between the two analyses:

Lemma 1. *Let $c \in Com$, $y \in Var$, and Γ be an environment. Let Γ' be the unique environment such that $l \vdash \Gamma\ \{c\}\ \Gamma'$ is derivable in the type system from Section 2. Moreover, let X be the set of all $x \in Var$ such that there exists a path from in to out in $PDG(CFG_c^{\{x\},\{y\}})$. Then $\Gamma'(y) = \bigsqcup_{x\in X} \Gamma(x)$ holds.*

Proof sketch. The proof (provided on the authors' website) uses a more general lemma for the judgment $pc \vdash \Gamma\{c\}\Gamma'$ that is proven by induction on the structure of c. It states that $\Gamma'(y) = \bigsqcup_{x\in X} \Gamma(x)$ holds if there is no path from *start* to *out* in $PDG(CFG_c^{\{\},\{y\}})$, and that $\Gamma'(y) = pc \sqcup \left(\bigsqcup_{x\in X} \Gamma(x)\right)$ holds otherwise. □

Lemma 1 is the key to establishing the following theorem that relates the precision of the type-based analysis to the precision of the PDG-based analysis, showing that the analyses have exactly the same precision.

Theorem 3. *A command $c \in Com$ is accepted by the type-based analysis if and only if it is accepted by the PDG-based analysis.*

Proof. Our proof is by contraposition. Let $\Gamma(x) = dom(x)$ for all $x \in Var$, and let Γ' be the unique environment such that $l \vdash \Gamma\ \{c\}\ \Gamma'$ is derivable in the type system from Section 2. If c is not accepted by the type-based analysis then $dom(y) = l$ and $\Gamma'(y) = h$ for some $y \in Var$. Hence, by Lemma 1 there exists $x \in Var$ with $dom(x) = h$ and a path $\langle in, \ldots, out\rangle$ in $PDG(CFG_c^{\{x\},\{y\}})$. Hence,

there is a path $\langle in, \ldots, out \rangle$ in $PDG(CFG_c^{H,L})$. Thus, c is not accepted by the PDG-based analysis. If c is not accepted by the PDG-based analysis then there is a path $\langle in, \ldots, out \rangle$ in $PDG(CFG_c^{H,L})$. But then there exist variables x, y with $dom(x) = h$ and $dom(y) = l$ such that there is a path $\langle in, \ldots, out \rangle$ in $PDG(CFG_c^{\{x\},\{y\}})$. Hence, by Lemma 1, $\Gamma'(y) = h$. Since $dom(y) = l$ it follows that $\Gamma'(y) \not\sqsubseteq dom(y)$. Thus, c is not accepted by the type-based analysis. □

Theorem 3 shows that the information flow analyses from [HS06] and [WLS09] have exactly the same precision. More generally this means that, despite their conceptual simplicity, type-based information flow analyses need not be less precise than PDG-based information flow analyses.

Given that both analyses have equal precision, the choice of an information flow analysis should be motivated by other aspects. For instance, if a program's environment is subject to modifications, one might desire a compositional analysis, and, hence, choose a type-based analysis. On the other hand, if a program is not accepted by the analyses one could use the PDG-based analysis to localize the source of potential information leakage by inspecting the path in the PDG that leads to the rejection of the program.

Beyond clarifying the connection between type-based and PDG-based information flow analyses, Theorem 3 also provides a bridge that can be used to transfer concepts from the one tradition of information flow analysis to the other. In the following section, we exploit this bridge to transfer the concept of rely-guarantee-style reasoning for the analysis of multi-threaded programs from type-based to PDG-based information flow analysis.

5 Information Flow Analysis of Multi-threaded Programs

Multi-threaded programs may exhibit subtle information leaks that do not occur in single-threaded programs. Such a leak is illustrated by the following example.

Example 8. Consider two threads with shared memory that execute commands $c_1 = \mathsf{if}\ (x)\ \mathsf{then\ skip; skip\ else\ skip\ fi};\ y{:=}\mathsf{True}$ and $c_2 = \mathsf{skip; skip};\ y{:=}\mathsf{False}$, respectively, and that are run under a Round-Robin scheduler that selects them alternately starting with the first thread and rescheduling after each execution step. If initially $x = \mathsf{True}$ then c_1 assigns True to y after c_2 assigns False to y. Otherwise, c_1 assigns True to y prior to the assignment to y in c_2. I.e., the initial value of x is copied into y. Such leaks are also known as *internal timing leaks*.

Many type-based analyses detect such leaks (e.g., [SV98,SS00,ZM03,MSS11]). Regarding PDG-based analyses, this is only the case for a recently proposed analysis [GS12]. However, this analysis has serious limitations: It forbids publicly observable nondeterminism, and it is not compositional (cf. Section 6 for a more detailed comparison). This motivated us to choose this domain for illustrating how the connection between type-based and PDG-based information flow analysis (from Section 4) can be exploited to transfer ideas from the one analysis style to the other. More concretely, we show how rely-guarantee-style

reasoning can be transferred from a type-based to a PDG-based information flow analysis. The outcome is a sound PDG-based information flow analysis for multi-threaded programs that is superior to the one in [GS12] in the sense that it supports publicly observable nondeterminism.

5.1 A Type-based Analysis for Multi-threaded Programs

We consider multi-threaded programs executing a fixed number of threads that interact via shared memory, i.e., configurations have the form $\langle (c_1, \ldots, c_k), \mathit{mem} \rangle$ where the commands c_i model the threads and mem models the shared memory.

In the following, we recall the type-based analysis from [MSS11] that exploits rely-guarantee-style reasoning, where typing rules for single threads exploit assumptions about when and how variables might be accessed by other threads. Assumptions are modeled by *modes* in the set $\mathit{Mod} = \{\mathit{asm\text{-}noread}, \mathit{asm\text{-}nowrite}\}$, where *asm-noread* and *asm-nowrite* are interpreted as the assumption that no other thread reads and writes a given variable, respectively.[3] The language for commands from Section 2 is extended as follows with a notation for specifying when one starts and stops making assumptions for a thread, respectively:

$$\mathit{ann} ::= \mathsf{acq}(m, x) \mid \mathsf{rel}(m, x) \qquad c ::= \ldots \mid /\!/\mathit{ann}/\!/\ c,$$

where $m \in \mathit{Mod}$ and $x \in \mathit{Var}$. The annotations $/\!/\mathsf{acq}(m, x)/\!/$ and $/\!/\mathsf{rel}(m, x)/\!/$, respectively, indicate that an assumption for x is acquired or released.

Typing judgments for commands have the form $\vdash \Lambda\, \{c\}\, \Lambda'$ where $\Lambda, \Lambda' : \mathit{Var} \rightharpoonup \mathcal{D}$ are *partial environments*. Partial environments provide an upper bound on the security level only for low variables for which a no-read and for high variables for which a no-write assumption is made, respectively. For other variables, the typing rules ensure that $\mathit{dom}(x)$ is an upper bound on the security level of the value of x. We write $\Lambda\langle x \rangle$ for the resulting upper bound (defined by $\Lambda\langle x \rangle = \Lambda(x)$ if Λ is defined for x and by $\Lambda\langle x \rangle = \mathit{dom}(x)$ otherwise). This reflects that (a) low variables that might be read by other threads must not store secrets because the secrets might be leaked in other threads (i.e., the upper bound for low variables without no-read assumption must be l), and that (b) other threads might write secrets into high variables without no-write assumption (and, hence, the upper bound for high variables without no-write assumption cannot be l).

The security type system contains two typing rules for assignments:

$$\frac{\Lambda(x) \text{ is defined} \quad \Lambda' = \Lambda[x \mapsto (\bigsqcup_{x \in fv(e)} \Lambda\langle x \rangle)]}{\vdash \Lambda\, \{x{:=}e\}\, \Lambda'} \qquad \frac{(\bigsqcup_{x \in fv(e)} \Lambda\langle x \rangle) \sqsubseteq \mathit{dom}(x) \quad \Lambda(x) \text{ is not defined} \quad \Lambda' = \Lambda}{\vdash \Lambda\, \{x{:=}e\}\, \Lambda'}$$

The left typing rule is like in the type system from Section 2. The rule applies if Λ is defined for the assigned variable. The right typing rule reflects that if Λ is not defined for the assigned variable x then $\mathit{dom}(x)$ must remain an upper bound on the security level of the value of x, and, hence, only expressions that do not contain secret information may be assigned to x if $\mathit{dom}(x) = l$.

[3] We omit the modes *guar-noread* and *guar-nowrite* representing guarantees from [MSS11], because they are irrelevant for the security type system.

A slightly simplified[4] variant of the typing rule for conditionals is as follows:

$$\frac{\vdash \Lambda\,\{c_1\}\,\Lambda' \quad \vdash \Lambda\,\{c_2\}\,\Lambda' \quad l = \bigsqcup_{x \in fv(e)} \Lambda\langle x\rangle}{\vdash \Lambda\,\{\mathsf{if}\ (e)\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2\ \mathsf{fi}\}\,\Lambda'}$$

In contrast to the corresponding typing rule in Section 2, the guard is required to be low. This ensures that programs with leaks like in Example 8 are not typable.

For the complete set of typing rules we refer to [MSS11].

Remark 1. In contrast to the type system in Section 2 the security level *pc* is not considered here, because the typing rules ensure that control flow does not depend on secrets (permitting some exceptions as indicated in Footnote 4).

Definition 16. *A multi-threaded program consisting of commands $c_1, \ldots, c_k$ is* accepted by the type-based analysis for multi-threaded programs *if the judgment $\vdash \Lambda_0\,\{c_i\}\,\Lambda_i'$ is derivable for each $i \in \{1, \ldots, k\}$ for some Λ_i' (where Λ_0 is undefined for all $x \in Var$) and the assumptions made are valid for the program.*

Validity of assumptions is formalized in [MSS11] by the notion of *sound usage of modes*. Theorem 6 in [MSS11] ensures that the type-based analysis for multi-threaded programs is sound with respect to *SIFUM-security*, an information flow security property for multi-threaded programs. We refer the interested reader to [MSS11] for the definitions of sound usage of modes and of SIFUM-security.

5.2 A Novel PDG-based Analysis for Multi-threaded Programs

We define a PDG-based analysis for multi-threaded programs by transferring rely-guarantee-style reasoning from the type-based analysis (Definition 16) to PDGs. To this end, we augment the set of edges of the program dependency graph $PDG(CFG_c^{H,L})$, resulting in a novel program dependency graph $PDG^{\parallel}(CFG_c^{H,L})$. Using this graph, the resulting analysis for multi-threaded programs is as follows:

Definition 17. *A multi-threaded program consisting of commands $c_1, \ldots, c_k$ is* accepted by the PDG-based analysis for multi-threaded programs *if there is no path from in to out in $PDG^{\parallel}(CFG_{c_i}^{H,L})$ for each $i \in \{1, \ldots, k\}$ and the assumptions made are valid for the program.*

It follows from Definition 17 that the analysis is compositional.

We now define the graph $PDG^{\parallel}(CFG_c^{H,L})$, where the additional edges in $PDG^{\parallel}(CFG_c^{H,L})$ model dependencies for nodes *in* and *out* that result from the concurrent execution of threads that respect the assumptions made for c.

Definition 18. *If command c is not of the form $/\!/ann/\!/\,c'$ we say that c* does not acquire and does not release $m \in Mod$ for $x \in Var$, *respectively. Command*

[4] The original rule from [MSS11] permits that guards depend on secrets (i.e., $h = \bigsqcup_{x \in fv(e)} \Lambda\langle x\rangle$) if the branches are in a certain sense indistinguishable.

$/\!/\mathsf{acq}(x,m)/\!/\,c$ acquires m for x *if and only if* c *does not release* m *for* x. *Command* $/\!/\mathsf{rel}(x,m)/\!/\,c$ releases m for x *if and only if* c *does not acquire* m *for* x.

For $c \in Com$ *we define the function* $modes_c : (N_c \times Mod) \to \mathcal{P}(Var)$ *by* $x \in modes_c(n,m)$ *if and only if for all paths* $p = \langle start, \ldots, n\rangle$ *in* CFG_c *there is a node* n' *on* p *such that* $c[n']$ *acquires* m *for* x, *and if* n'' *follows* n' *on* p *then* $c[n'']$ *does not release* m *for* x.

Definition 19. *Let* $c \in Com$. *Then* $PDG^{\|}(CFG_c^{H,L}) = (N, E \cup E')$ *for* $(N,E) = PDG(CFG_c^{H,L})$ *and* $(n,n') \in E'$ *if and only if one of the following holds:*

1. $n = in$ *and there exist a variable* $x \in H \cap use_c(n')$, *a node* $n'' \in N$ *with* $x \notin modes_c(n'', \textit{asm-nowrite})$, *and a path* p *from* n'' *to* n' *with* $x \notin def_c(n''')$ *for every node* n''' *on* p *with* $n''' \neq n''$ *and* $n''' \neq n'$,
2. $n' = out$ *and there exist a variable* $x \in L \cap def_c(n')$, *a node* $n'' \in N$ *with* $x \notin modes_c(n'', \textit{asm-noread})$, *and a path* p *from* n *to* n'' *such that* $x \notin def_c(n''')$ *for every node* n''' *on* p *with* $n''' \neq n$ *and* $n''' \neq n''$, *or*
3. $n \in \{1, \ldots, |c|\}$, $c[n] \in Exp$, *and* $n' = out$.

The edges defined in Items 1 and 2 are derived from the typing rules for assignments: The edge (n, out) in Item 1, where n defines a low variable whose value might be eventually read by another thread, ensures that the command is rejected if the definition at n might depend on secret input (because then there is a path from in to n). The edge (in, n), where n uses a high variable whose value might have been written by another thread, captures that the high variable might contain secrets when being used at n. The edges defined in Item 3 are derived from the typing rule for conditionals: If the guard represented by Node n depends on secrets (i.e., there is a path from in to n) then the command is rejected because together with the edge (n, out) there is a path from in to out.

Example 9. Consider the following command c where $dom(x) = l$ and $dom(y) = h$:

$$/\!/\mathsf{acq}(\textit{asm-noread}, x)/\!/; x{:=}y; x{:=}0; /\!/\mathsf{rel}(\textit{asm-noread}, x)/\!/$$

Then $PDG^{\|}(CFG_c^{H,L}) = PDG(CFG_c^{H,L})$, and c is accepted by the PDG-based analysis for multi-threaded programs. Let furthermore $c' = x{:=}y; x{:=}0$. Then $PDG^{\|}(CFG_{c'}^{H,L}) \neq PDG(CFG_c^{H,L})$, because the graph $PDG^{\|}(CFG_{c'}^{H,L})$ contains an edge from the node representing $x{:=}y$ to Node out (due to Item 2 in Definition 19). Hence, c' is not accepted, because $PDG^{\|}(CFG_{c'}^{H,L})$ contains a path from Node in to Node out via the node representing the assignment $x{:=}y$.

Not accepting c' is crucial for soundness because another thread executing $x'{:=}x$ could copy the intermediate secret value of x into a public variable x'.

Theorem 4. *If a multi-threaded program is accepted by the PDG-based analysis for multi-threaded programs then the program is accepted by the type-based analysis for multi-threaded programs.*

The proof (provided on the authors' website) is by contradiction; it exploits the connection between PDG-based and type-based analysis stated in Lemma 1. Soundness of the PDG-based analysis follows directly from Theorem 4 and the soundness of the type-based analysis (see [MSS11, Theorem 6]).

6 Related Work

We focus on related work covering flow-sensitive type-based analysis, PDG-based analysis for concurrent programs, and connections between analysis techniques. For an overview on language-based information flow security we refer to [SM03].

Flow-sensitivity of type-based analyses. In contrast to PDG-based information flow analyses, many type-based information flow analyses are not flow-sensitive. The first flow-sensitive type-based information flow analysis is due to Hunt and Sands [HS06]. Based on the idea of flow-sensitive security types from [HS06], Mantel, Sands, and Sudbrock developed the first sound flow-sensitive security type system for concurrent programs [MSS11].

PDG-based analyses for concurrent programs. Hammer [Ham09] presents a PDG-based analysis for concurrent Java programs, where edges between the PDGs of the individual threads are added following the extension of PDGs to concurrent programs from [Kri03]. However, there is no soundness result. In fact, since the construction of PDGs from [Kri03] does not capture dependencies between nodes in the PDG that result from internal timing (cf. Example 8), the resulting PDG-based information flow analysis fails to detect some information leaks.

Giffhorn and Snelting [GS12] present a PDG-based information flow analysis for multi-threaded programs that does not accept programs with internal timing leaks. The analysis enforces an information flow property defined in the tradition of observational determinism [RWW94,ZM03], and, therefore, does not accept any programs that have nondeterministic public output. Hence, the analysis forbids useful nondeterminism, which occurs, for instance, when multiple threads append entries to the same log file. Our novel analysis (from Section 5) permits concurrent writes to public variables and, hence, accepts secure programs that are not accepted by the analysis from [GS12]. Moreover, in contrast to the analysis from [GS12] our novel analysis is compositional.

Connections between different analysis techniques. Hunt and Sands show in [HS06] that the program logic from [AB04] is in fact equivalent to the type-based analysis from [HS06]. Rehof and Fähndrich [RF01] exploit concepts from PDGs (the computation of so-called summary edges for programs with procedures) in a type-based flow analysis. In this article, we go a step further by establishing and exploiting a formal connection between a type-based and a PDG-based analysis.

7 Conclusion

While security type systems are established as analysis technique for information flow security, information flow analyses based on program dependency graphs (PDGs) have only recently received increased attention. In this article, we investigated the relationship between these two alternative approaches.

As a main result, we showed that the precision of a prominent type-based information flow analysis is not only roughly similar to the precision of a prominent PDG-based analysis, but that the precision is in fact exactly the same. Moreover, our result provides a bridge for transferring techniques and ideas from one

tradition of information flow analysis to the other. This is an interesting possibility because there are other relevant attributes than the precision of an analysis (e.g., efficiency and the availability of tools). We showed at the example of rely-guarantee-style information flow analysis for multi-threaded programs that this bridge is suitable to facilitate learning by one sub-community from the other.

We hope that our results clarify the relationship between the two approaches. The established relationship could be used as a basis for communication between the sub-communities to learn from each other and to pursue joint efforts to make semantically justified information flow analysis more practical. For instance, our results give hope that results on controlling declassification with security type systems can be used to develop semantic foundations for PDG-based analyses that permit declassification. Though there are PDG-based analyses that permit declassification (e.g., [HS09]), all of them yet lack a soundness result, and, hence, it is unclear which noninterference-like property they certify.

Acknowledgment. This work was funded by the DFG under the project FM-SecEng in the Computer Science Action Program (MA 3326/1-3).

References

[AB04] T. Amtoft and A. Banerjee. Information Flow Analysis in Logical Form. In *Symposium on Static Analysis*, LNCS 3148, pages 100–115. Springer, 2004.

[BC02] G. Boudol and I. Castellani. Noninterference for Concurrent Programs and Thread Systems. *Theoretical Computer Science*, 281(1–2):109–130, 2002.

[Che93] J. Cheng. Slicing Concurrent Programs - A Graph-Theoretical Approach. In *Workshop on Automated and Algorithmic Debugging*, LNCS 749, pages 223–240. Springer, 1993.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[GM82] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[GS12] D. Giffhorn and G. Snelting. Probabilistic Noninterference Based on Program Dependence Graphs. Technical Report 6, Karlsruher Institut für Technologie (KIT), 2012.

[Ham09] C. Hammer. *Information Flow Control for Java*. PhD thesis, Universität Karlsruhe (TH), 2009.

[HRB90] S. Horwitz, T. W. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[HS06] S. Hunt and D. Sands. On Flow-Sensitive Security Types. In *ACM Symposium on Principles of Programming Languages*, pages 79–90, 2006.

[HS09] C. Hammer and G. Snelting. Flow-sensitive, Context-sensitive, and Object-sensitive Information Flow Control based on Program Dependence Graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[HUM92] C. S. Hsieh, E. A. Unger, and R. A. Mata-Toledo. Using Program Dependence Graphs for Information Flow Control. *Journal of Systems and Software*, 17(3):227–232, 1992.

[Kri03] J. Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, 2003.

[Man11] H. Mantel. Information Flow and Noninterference. In *Encyclopedia of Cryptography and Security (2nd Ed.)*, pages 605–607. Springer, 2011.

[MS04] H. Mantel and D. Sands. Controlled Declassification based on Intransitive Noninterference. In *ASIAN Symposium on Programming Languages and Systems*, LNCS 3302, pages 129–145. Springer, 2004.

[MSS11] H. Mantel, D. Sands, and H. Sudbrock. Assumptions and Guarantees for Compositional Noninterference. In *IEEE Computer Security Foundations Symposium*, pages 218–232, 2011.

[Mye99] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *ACM Symposium on Principles of Programming Languages*, pages 228–241, 1999.

[NP08] M. Naik and J. Palsberg. A Type System Equivalent to a Model Checker. *ACM Transactions on Programming Languages and Systems*, 30(5):1–24, 2008.

[PO95] J. Palsberg and P. O'Keefe. A Type System Equivalent to Flow Analysis. In *ACM Symposium on Principles of Programming Languages*, pages 367–378, 1995.

[RF01] J. Rehof and M. Fähndrich. Type-Based Flow Analysis: From Polymorphic Subtyping to CFL-Reachability. In *ACM Symposium on Principles of Programming Languages*, pages 54–66, 2001.

[RWW94] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf. Non-interference through Determinism. In *European Symposium on Research in Computer Security*, LNCS 875, pages 33–53, 1994.

[SM03] A. Sabelfeld and A. C. Myers. Language-based Information-Flow Security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.

[SS00] A. Sabelfeld and D. Sands. Probabilistic Noninterference for Multi-threaded Programs. In *IEEE Computer Security Foundations Workshop*, pages 200–215, 2000.

[SV98] G. Smith and D. Volpano. Secure Information Flow in a Multi-threaded Imperative Language. In *ACM Symposium on Principles of Programming Languages*, pages 355–364, 1998.

[VS97] D. M. Volpano and G. Smith. A Type-Based Approach to Program Security. In *Conference on Theory and Practice of Software Development*, LNCS 1214, pages 607–621. Springer, 1997.

[VSI96] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(3):1–21, 1996.

[WL08] D. Wasserrab and A. Lochbihler. Formalizing a Framework for Dynamic Slicing of Program Dependence Graphs in Isabelle/HOL. In *Conference on Theorem Proving in Higher Order Logics*, LNCS 5170, pages 294–309. Springer, 2008.

[WLS09] D. Wasserrab, D. Lohner, and G. Snelting. On PDG-based Noninterference and its Modular Proof. In *ACM Workshop on Programming Languages and Analysis for Security*, pages 31–44, 2009.

[ZM03] S. Zdancewic and A. C. Myers. Observational Determinism for Concurrent Program Security. In *IEEE Computer Security Foundations Workshop*, pages 29–43, 2003.

Galliwasp: A Goal-Directed Answer Set Solver

Kyle Marple and Gopal Gupta

University of Texas at Dallas
800 W. Campbell Road
Richardson, TX 75080, USA

Abstract. *Galliwasp* is a goal-directed implementation of answer set programming. Unlike other answer set solvers, *Galliwasp* computes partial answer sets which are provably extensible to full answer sets. *Galliwasp* can execute arbitrary answer set programs in a top-down manner similar to SLD resolution. *Galliwasp* generates candidate answer sets by executing *ordinary rules* in a top-down, goal-directed manner using *coinduction*. *Galliwasp* next checks if the candidate answer sets are consistent with restrictions imposed by *OLON rules*. Those that are consistent are reported as solutions. Execution efficiency is significantly improved by performing the consistency check incrementally, i.e., as soon as an element of the candidate answer set is generated. We discuss the design of the *Galliwasp* system and its implementation. *Galliwasp*'s performance figures, which are comparable to other popular answer set solvers, are also presented.

1 Introduction

As answer set programming (ASP) [8] has gained popularity, answer set solvers have been implemented using various techniques. These techniques range from simple guess-and-check based methods to those based on SAT solvers and complex heuristics. None of these techniques impart any operational semantics to the answer set program in the manner that SLD resolution does to Prolog; they can be thought of as being similar to bottom-up methods for evaluating logic programs. Earlier we described [10, 17] how to find partial answer sets by means of a top-down, goal-directed method based on coinduction. The *Galliwasp* system [16], described in this paper, is the first efficient implementation of this goal-directed method. *Galliwasp* is consistent with the conventional semantics of ASP: no restrictions are placed on the queries and programs that can be executed. That is, any A-Prolog program can be executed on *Galliwasp*.

Galliwasp is an implementation of A-Prolog [7] that uses grounded normal logic programs as input. The underlying algorithm of *Galliwasp* leverages coinduction [23, 10] to find partial answer sets containing a given query. Programs are executed in a top-down manner in the style of Prolog, computing partial answer sets through SLD-style call resolution and backtracking. Each partial answer set is provably extensible to a complete answer set [10, 17].

The algorithm of *Galliwasp* uses call graphs to classify rules according to two attributes: (i) if a rule can be called recursively with an odd number of negations

between the initial call and its recursive invocation, it is said to contain an *odd loop over negation* (OLON) and referred to as an OLON rule for brevity, and (ii) if a rule has at least one path in the call graph that will not result in such a call, it is called an ordinary rule. Our goal-directed method uses ordinary rules to generate candidate answer sets (via co-induction extended with negation) and OLON rules to reject invalid candidate sets. The procedure can be thought of as following the *generate and test* paradigm with ordinary rules being used for generation of candidate answer sets and OLON rules for testing that a candidate answer set is, indeed, a valid answer set.

The main contribution of this paper is an execution mechanism in which consistency checks are incrementally executed as soon as an element of the candidate answer set is generated. By interleaving the generation and testing of candidate answer sets in this manner, *Galliwasp* speeds up execution significantly. As soon as an element `p` is added to a candidate answer set, OLON rules that might reject the answer set due to presence of `p` in it are invoked.

In this paper we describe the design and implementation of *Galliwasp*. This includes an overview of goal-directed ASP, the design and implementation of *Galliwasp*'s components, and techniques developed to improve performance. Finally, the performance figures are presented and compared to other ASP solvers.

There are many advantages of a goal-directed execution strategy: (i) ASP can be extended to general predicates [18], i.e., to answer set programs that do not admit a finite grounding; (ii) extensions of ASP to constraints (in the style of CLP(R)) [12], probabilities (in the style of ProbLog [4]), etc., can be elegantly realized; (iii) Or-parallel implementations of ASP that leverage techniques from parallel Prolog implementations [11] can be developed; (iv) abductive reasoning [14] can be incorporated with greater ease. Work is in progress to extend *Galliwasp* in these directions.

2 Goal-Directed Answer Set Programming

The core of *Galliwasp* is a goal-directed execution method for computing answer sets which is sound and complete with respect to the Gelfond-Lifschitz method [10, 17]. For convenience, two key aspects of our goal-directed method are summarized here: its handling of OLON rules and its use of coinduction.

2.1 OLON Rules and Ordinary Rules

Our goal-directed method requires that we identify and properly handle both ordinary and OLON rules, as defined in the introduction. Rules are classified by constructing and examining a call graph. Unlike in Prolog, literals can be either positive or negative, and the attributes of a rule are determined by the number of negations between any recursive calls in the call graph.

To build the call graph, each positive literal is treated as a node in the graph. For each rule, arcs are drawn from the node corresponding to the head of the rule to every node corresponding to the positive form of a literal in the body

of the rule. If the literal in the body is positive, the arc is blue; if the literal is negative, the arc is red. At most one of each color arc is created between any two nodes, and each arc is labelled to identify all of the rules associated with it.

When the graph has been created, each path is examined in a bottom-up fashion, stopping when any literal in the path is repeated. If there is a path from a node `N` to itself such that the number of red arcs on the path is odd, then we call all of the rules for N's literal associated with the path OLON rules. If there is no path from a node `N` to itself, or if there is a path from `N` to itself such that the number of red arcs on the path is even, we call all of the rules for N's literal associated with the path ordinary rules.

Every rule will have at least one of the two attributes. Additionally, a rule can be both an OLON rule and an ordinary rule, as the example below illustrates:

```
p :- not q. ... (i)
q :- not p. ... (ii)
q :- not r. ... (iii)
r :- not p. ... (iv)
```

Rule (i) is both an OLON rule and an ordinary rule: ordinary because of rule (ii), OLON because of rules (iii) and (iv).

2.2 Coinductive Execution

The goal-directed method [10, 17] generates candidates answer sets by executing ordinary rules via coinduction extended with negation. Given a query, an extended query is constructed to enforce the OLON rules. This extended query is executed in the style of SLD resolution with coinduction (co-SLD resolution) [23, 10].

Under the operational semantics of coinduction, a call `p` succeeds if it unifies with one of its ancestor calls. Each call is remembered, and this set of ancestor calls forms the *coinductive hypothesis set* (CHS). Under our algorithm, the CHS also constitutes a candidate answer set. As such, it would be inconsistent for `p` and `not p` to be in the CHS at the same time: when this is about to happen, the system must backtrack. As a result, only ordinary rules can generate candidate answer sets. Any OLON rule that is not also an ordinary rule will fail during normal execution, as it will attempt to add the negation of its head to the CHS.

Candidate sets produced by ordinary rules must still be checked for consistency with the OLON rules in a program. To accomplish this, the query is extended to perform this check.

Before looking at this extension, let us consider an OLON rule of the form

```
p :- B, not p.
```

where `B` is a conjunction of goals. One of two cases must hold for an answer set to exist: (i) `p` is in the answer set through another rule in the program, (ii) at least one goal in `B` must fail. This is equivalent to saying that the negation of the rule, `not B` $\vee$ `p` must succeed. Note that checks of this form can be created for any OLON rule [10, 17]. This includes both OLON rules of the form

```
p :- B.
```

where the call to `not p` is indirect, and headless rules of the form

```
:- B.
```

where the negation `not B` must succeed.

Our method handles OLON rules by creating a special rule, called the NMR check, which contains a sub-check for each OLON rule in a program. The body of the NMR check is appended to each query prior to evaluation, ensuring that any answer set returned is consistent with all of the OLON rules in the program. Prior to creating the sub-checks, a copy of each OLON rule is written in the form

```
p :- B, not p.
```

as shown above, with the negation of the rule head added to the body in cases where the call is indirect. Each sub-check is then created by negating the body of one of the copied rules. For instance, the sub-check for a rule

```
p :- q, not p.
```

will be

```
chk_p :- not q.
chk_p :- p.
```

The body of the NMR check will then contain the goal `chk_p`, ensuring that the OLON rule is properly applied.

Some modification to co-SLD resolution is necessary to remain faithful to ASP's stable model semantics. This is due to the fact that coinduction computes the greatest fixed point of a program, while the GL method computes a fixed point that is between the least fixed point (*lfp*) and greatest fixed point (*gfp*) of a program, namely, the *lfp* of the residual program.To adapt coinduction for use with ASP, our goal-directed method restricts coinductive success to cases in which there are an even, non-zero number of calls to `not` between a call and an ancestor to which it unifies. This prevents rules such as

```
p :- p.
```

from succeeding, as this would be the case under normal co-SLD resolution. To compute an answer set for a given program and query, our method performs co-SLD resolution using this modified criterion for coinductive success. When both a query and the NMR check have succeeded, the CHS will be a valid answer set [10, 17]. Consider the program below:

```
p :- q.
q :- p.
```

Under normal co-SLD resolution, both `p` and `q` would be allowed to succeed, resulting in the incorrect answer set {`p, q`}. However, using our modified criterion, both rules will fail, yielding the correct answer set {`not p, not q`}.

3 The Galliwasp System

The *Galliwasp* system is an implementation of the above SLD resolution-style, goal-directed method of executing answer set programs. A number of issues arise that are discussed next. Improvements to the execution algorithm that make *Galliwasp* more efficient are also discussed.

3.1 Order of Rules and Goals

As with normal SLD resolution, rules for a given literal are executed in the order given in the program, with the body of each rule executed left to right. With the exception of OLON rules, once the body of a rule whose head matches a given literal has succeeded, the remaining rules do not need to be accessed unless failure and backtracking force them to be selected.

As in top-down implementations of Prolog, this means that the ordering of clauses and goals can directly impact the runtime performance of a program. In the best case scenario, this can allow *Galliwasp* to compute answers much faster than other ASP solvers, but in the worst case scenario *Galliwasp* may end up backtracking significantly more, and as a result take much longer.

One example of this can be found in an instance of the Schur Numbers benchmark used in Sect. 5. Consider the following clauses from the grounded instance for 3 partitions and 13 numbers:

```
_false :- not haspart(1).
_false :- not haspart(2).
_false :- not haspart(3).
_false :- not haspart(4).
_false :- not haspart(5).
_false :- not haspart(6).
_false :- not haspart(7).
_false :- not haspart(8).
_false :- not haspart(9).
_false :- not haspart(10).
_false :- not haspart(11).
_false :- not haspart(12).
_false :- not haspart(13).
```

During benchmarking, *Galliwasp* was able to find an answer set for the program containing the above clauses in 0.2 seconds. However, the same program with the order of only the above clauses reversed could be left running for several minutes without terminating.

Given that *Galliwasp* is currently limited to programs that have a finite grounding, its performance can be impacted by the grounder program that is used. In the example above, all 13 ground clauses are generated by the grounding of a single rule in the original program. Work is in progress to extend *Galliwasp* to allow direct execution of datalog-like ASP programs (i.e., those with only constants and variables) without grounding them first. In such a case, the user would have much more control over the order in which rules are tried.

3.2 Improving Execution Efficiency

The goal-directed method described in Sect. 2 can be viewed as following the *generate and test* paradigm. Given a query `Q, N` where `Q` represents the original user query, and `N` the NMR check, the goal directed procedure *generates* candidate answer sets through the execution of `Q` using ordinary rules. These

candidate answer sets are *tested* by the NMR check `N` which rejects a candidate if the restrictions encoded in the OLON rules are violated.

Naive generate and test can be very inefficient, as the query `Q` may generate a large number of candidate answer sets, most of which may be rejected by the NMR check. This can lead to a significant amount of backtracking, slowing down the execution. Execution can be made significantly more efficient by generating and testing answer sets incrementally. This is done by interleaving the execution of `Q` and `N`. As soon as a literal, say `p`, is added to a candidate answer set, goals in NMR check that correspond to OLON rules that `p` may violate are invoked. We illustrate the incremental generate and test algorithm implemented in *Galliwasp*.

Consider the rules below:

```
p :- r.
p :- s.
q :- big_goal.
r :- not s.
s :- not r.
:- p, r.
```

Let us assume that this fragment is part of a program where `big_goal` will always succeed, but is computationally expensive and contains numerous choice points. For simplicity, let us also assume that the complete program contains no additional OLON rules, and no additional rules for the literals `p`, `q`, `r` or `s`. As the program contains only the OLON rule from the fragment above, the NMR check will have only one sub-check, which will consist of two clauses:

```
nmr_check :- chk_1.
chk_1 :- not p.
chk_1 :- not r.
```

Next, assume that the program is compiled and run with the following query:

```
?- p, q.
```

As with any query, the NMR check will be appended, resulting in the final query:

```
?- p, q, nmr_check.
```

The NMR check ensures that `p` and `r` are never present in the same answer set, so any valid answer set will contain `p`, `q`, `not r`, `s`, `big_goal`, and the literals upon which `big_goal` depends.

If the program is run without NMR check reordering, the first clause for `p` will initially succeed, adding `r` to the CHS. Failure will not occur until the NMR check is reached, at which point backtracking will ensue. Eventually, the initial call to `p` will be reached, the second clause will be selected, and the query will eventually succeed. However, the choicepoints in `big_goal` could result in a massive amount of backtracking, significantly delaying termination. Additionally, `big_goal` will have to succeed twice before a valid answer set is found.

Now let us consider the same program and query using incremental generate and test. The first clause for `p` will still initially succeed. However, as `r` succeeds, the clause of `chk_1` calling `not r` will be removed. Since only one clause will remain for `chk_1`, `chk_1` will be reordered, placing it immediately after the call to `p` and before the call to `q`. The call to `p` will then succeed as before, but

failure and backtracking will occur almost immediately, as the call to `chk_1` will immediately call `not p`. As before, the call to `not p` will result in the second clause for `p` being selected and the query will eventually succeed. However, as failure occurs before `big_goal` is reached, the resulting backtracking is almost eliminated. Additionally, `big_goal` does not need to be called twice before a solution is found. As a result, execution with NMR check reordering will terminate much sooner than execution without reordering.

Adding incremental test and generation to *Galliwasp* leads to a considerable improvement in efficiency, resulting in the performance of the *Galliwasp* system becoming comparable to state-of-the-art solvers.

4 System Architecture of Galliwasp

The *Galliwasp* system consists of two components: a compiler and an interpreter. The compiler reads in a grounded instance of an ASP program and produces a compiled program, which is then executed by the interpreter to compute answer sets. An overview of the system architecture is shown in Fig. 4.

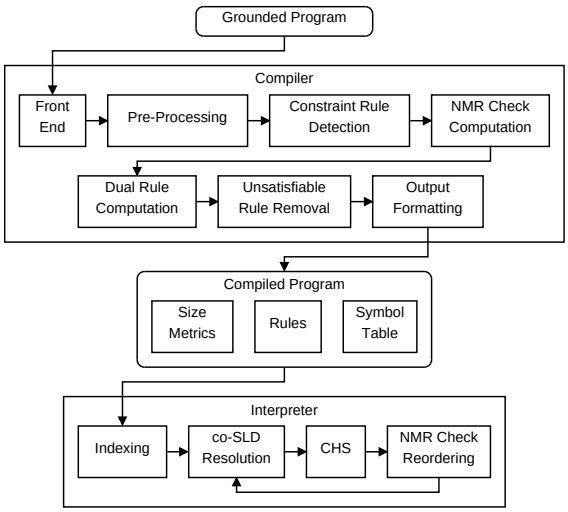


Fig. 1. *Galliwasp*'s system architecture.

4.1 Compiler

The primary goal of *Galliwasp*'s design is to maximize the runtime performance of the interpreter. Because *Galliwasp* computes answer sets for a program based on user-supplied queries, answer sets cannot be computed until a program is

actually run by the interpreter. However, many of the steps in our goal-directed algorithm, as well as parsing, program transformations and static analysis, are independent of the query. The purpose of *Galliwasp*'s compiler is to perform as much of the query-independent work as possible, leaving only the actual computation of answer sets to the interpreter. Because a single program can be run multiple times with different queries, compilation also eliminates the need to repeat the query-independent steps every time a program is executed. The most important aspects of the compiler are the input language accepted, the parsing and pre-processing performed, the construction of the NMR check, the program transformations applied, and the formatting of the compiled output.

Input Language The *Galliwasp* compiler's input language is grounded A-Prolog [7], extended to allow partial compatibility with text-mode output of the *lparse* grounder. This allows grounded program instances to be obtained by invoking *lparse* with the `-t` switch, which formats the output as text.

Only a subset of *lparse*'s text output is supported by *Galliwasp*. The support is made possible by special handling of rules with the literal `_false` as their head, a convention used by *lparse* to add a head to otherwise headless rules, and support for *lparse*'s `compute` statement. Other features of *lparse*, such as constraint and weight literals, choice rules, and optimize statements, are not currently supported in *Galliwasp*. Work is in progress to support them.

When *lparse* encounters a headless rule, it produces a grounded rule with `_false` as the head and a compute statement containing the literal `not _false`. Because the literal `_false` is not present in the body of any rule, special handling is required to properly detect such rules as OLON rules.

The compute statements used by *lparse* are of the form

```
compute N { Q }.
```

where `N` specifies the number of answer sets to compute and `Q` is a set of literals that must be present in a valid answer set. Our system handles these statements by treating them as optional, hard-coded queries. If a compute statement is present, the interpreter may be run without user interaction, computing up to `N` answer sets using `Q` as the query. When the interpreter is run interactively, it ignores compute statements and executes queries entered by the user.

Parsing and Pre-Processing The compiler's front end and pre-processing stages prepare the input program for easy access and manipulation during the rest of the compilation process. The front end encompasses the initial lexical analysis and parsing of the input program, while the pre-processing stage handles additional formatting and simplification, and substitutes integers for literals.

After lexical analysis of the input program is performed, it is parsed into a list of rules and statements by a definite clause grammar (DCG). During parsing, a counter is used to number the rules as they are read, so that the relative order of rules for a given literal can be maintained.

The list of statements produced by the DCG is next converted into a list of rules and a single compute statement. Each rule is checked to remove duplicate

goals. Any rule containing its head as a goal with no intervening negation will also be removed at this stage. The cases covered in this step should not normally occur, but as they are allowed by the language, they are addressed before moving on.

The next stage of pre-processing is integer substitution. Every propositional symbol `p` is mapped to a unique integer $N_p > 0$. A positive literal `p` is represented by N_p, while a negative literal `not p` is represented by $-N_p$. Since the interpreter must use the original names of propositions when it prints the answer, a table that maps each N_p to `p` is included in the compiled output.

Finally, the list of rules is sorted by head, maintaining the relative order of rules with the same head. This eliminates the need for repeated searching in subsequent stages of compilation. After sorting, compilation moves on to the next stage, the detection of OLON rules and construction of the NMR check.

Construction of the NMR Check Construction of the NMR check begins with the detection of the OLON rules in the ASP program. This detection is performed by building and traversing a call graph similar to the one described in Sect. 2.1. These rules are then used to construct the individual checks that form the NMR check, as described in Sect. 2.2.

Clauses for the sub-checks are treated as any other rule in the program, and subject to program transformation in the next stage of compilation. However, the NMR check itself is not modified by the program transformation stage. Instead, if the modified sub-checks allow for immediate failure, this will be detected at runtime by the interpreter.

Program Transformation The program transformation stage consists of computing dual rules, explained below, and removing rules when it can be trivially determined at compile time that they will never succeed. This stage of compilation improves performance without affecting the correctness of our algorithm.

Dual rules, i.e., rules for the negation of each literal, are computed as follows. For proposition `p` defined by the rules,

```
p :- B1.
...
p :- Bn.
```

where each B_i is a conjunction of positive and negative literals, its dual

```
not p :- not B1, ..., not Bn.
```

is added to the program. For any proposition `q` for which there are no rules whose head contains `q`, a fact is added for `not q`.

The addition of dual rules simplifies the design of the interpreter by removing the need to track the scope of negations: all rules can be executed in a uniform fashion, regardless of the number of negations encountered. When a negated calls are encountered, they are expanded using dual rules. While adding these rules may significantly increase the size of the program, this is not a problem: the interpreter performs indexing that allows it to access rules in constant time.

After the dual rules have been computed, the list is checked once more to remove simple cases of rules that can be trivially determined to always fail. This step simply checks to see if a fact exists for the negation of some goal in the rule body and removes the rule if this is the case. If the last rule for a literal is removed, a fact for the negation is added and subsequent rules calling the literal will also be removed.

Output Formatting As with the rest of the compiler, the output produced by the compiler is designed to reduce the amount of work performed by the interpreter. This is done by including space requirements and sorting the rules by their heads before writing them to the output.

As a result of the output formatting, the interpreter is able to read in the input and create the necessary indexes in linear time with respect to the size of the compiled program. After indexing, all that remains is to execute the program, with all other work having been performed during compilation.

4.2 Interpreter

While the compiler was designed to perform a variety of time consuming tasks, the interpreter has been designed to maximize run-time performance, finding an answer set or determining failure as quickly as possible. Two modes of operation, interactive and automatic, are supported. When a program is run interactively, the user can input queries and accept or reject answer sets as can be done with answers to a query in Prolog. In automatic mode it executes a compute statement that is included in the program. In either mode, the key operations can be broken up into three categories: program representation and indexing, co-SLD resolution, and dynamic incremental enforcement of the NMR check.

Program Representation and Indexing One of the keys to *Galliwasp*'s performance is the indexing performed prior to execution of the program. As a result, look-up operations during execution can be performed in constant time, much as in any implementation of Prolog. As mentioned in Sect. 4.1, the format of the compiler's output allows the indexes to be created in time that is linear with respect to the size of the program.

To allow for the NMR check interleaving and simplification discussed in Sect. 4.2, the query and NMR check are stored and indexed separately from the rest of the program. Whereas the rules of the program, including the sub-checks of the NMR check, are stored in ordinary arrays, the query and NMR check are stored in a linked list. This allows their goals to be reordered in constant time.

Co-SLD Resolution Once the program has been indexed, answer sets are found by executing the query using coinduction, as described in Sect. 2.2. Each call encountered is checked against the CHS. If the call is not present, it is added to the CHS and expanded according to ordinary SLD resolution. If the call is already in the CHS, immediate success or failure occurs, as explained below.

As mentioned in Sect. 2.2, our algorithm requires that a call that unifies with an ancestor cannot succeed coinductively unless it is separated from that ancestor by an even, non-zero number of intervening negations. This is facilitated by our use of dual rules, discussed in Sect. 4.1.

When the current call is of the form `not p` and the CHS contains `p` (or vice versa), the current call must fail, lest the CHS become inconsistent. If the current call is identical to one of the elements of the CHS, then the number of intervening negations must have been even. However, it is not clear that it was also non-zero. This additional information is provided by a counter that tracks of the number of negations encountered between the root of the proof tree and the tip of the current branch. When a literal is added to the CHS, it is stored with the current value of the counter. A recursive call to the literal can coinductively succeed only if the counter has increased since the literal was stored.

Dynamic Incremental Enforcement of the NMR Check Because recursive calls are never expanded, eventual termination is guaranteed, just as for any other ASP solver. However, since we use backtracking, the size of the search space can cause the program to execute for an unreasonable amount of time.

To alleviate this problem, we introduced the technique of incrementally enforcing the NMR check (cf. Sect. 3.2 above). Our technique can be viewed as a form of coroutining: steps of the testing and generation phases are interleaved. The technique consists of simplifying NMR sub-checks and of reordering the calls to sub-checks within the query. These modifications are done whenever a literal is added to the CHS, and are "undone" upon backtracking.

When a literal is added to the CHS, all occurrences of that literal are removed from every sub-check, as they would immediately succeed when treated as calls. If such a removal causes the body of a sub-check clause to become empty, the entire sub-check is removed, as it is now known to be satisfied.

The next step is to remove every sub-check clause that contains the negation of the literal: such a clause cannot be satisfied now. If this clause causes the entire sub-check to disappear, the literal obviously cannot be added to the CHS, and backtracking occurs. If the sub-check does not disappear, but is reduced to a single clause (i.e., it becomes deterministic), then a call to the sub-check is moved to the front of the current query: this ensures early testing of whether the addition of the literal is consistent with the relevant OLON rules. If the resultant sub-check contains more than one clause, there is no attempt to execute it earlier than usual, as that might increase the size of the search space.

As mentioned in Sect. 4.2, indexing allows these modifications to be performed in constant time. If the original search space is small, they may result in a small increase in runtime. In most non-trivial cases, however, the effect is a dramatic decrease in the size of the search space. It is this technique that enables *Galliwasp* to be as efficient as illustrated in the next section: early versions of the system used a simple generate-and-test approach, but many of the programs that now terminate in a fraction of a second ran for inordinate amounts of time.

5 Performance Results

Table 1. Performance results; times in seconds

Problem	*Galliwasp*	*clasp*	*cmodels*	*smodels*
queens-12	0.033	0.019	0.055	0.112
queens-13	0.034	0.022	0.071	0.132
queens-14	0.076	0.029	0.098	0.362
queens-15	0.071	0.034	0.119	0.592
queens-16	0.293	0.043	0.138	1.356
queens-17	0.198	0.049	0.176	4.293
queens-18	1.239	0.059	0.224	8.653
queens-19	0.148	0.070	0.272	3.288
queens-20	6.744	0.084	0.316	47.782
queens-21	0.420	0.104	0.398	95.710
queens-22	69.224	0.112	0.472	N/A
queens-23	1.282	0.132	0.582	N/A
queens-24	19.916	0.152	0.602	N/A
mapclr-4x20	0.018	0.006	0.011	0.013
mapclr-4x25	0.021	0.007	0.014	0.016
mapclr-4x29	0.023	0.008	0.016	0.018
mapclr-4x30	0.026	0.008	0.016	0.019
mapclr-3x20	0.022	0.005	0.009	0.008
mapclr-3x25	0.065	0.006	0.011	0.010
mapclr-3x29	0.394	0.006	0.012	0.011
mapclr-3x30	0.342	0.007	0.012	0.011
schur-3x13	0.209	0.006	0.010	0.009
schur-2x13	0.019	0.005	0.007	0.006
schur-4x44	N/A	0.230	1.394	0.349
schur-3x44	7.010	0.026	0.100	0.076
pigeon-10x10	0.020	0.009	0.020	0.025
pigeon-20x20	0.050	0.048	0.163	0.517
pigeon-30x30	0.132	0.178	0.691	4.985
pigeon-8x7	0.123	0.072	0.089	0.535
pigeon-9x8	0.888	0.528	0.569	4.713
pigeon-10x9	8.339	4.590	2.417	46.208
pigeon-11x10	90.082	40.182	102.694	N/A

In this section, we compare the performance of *Galliwasp* to that of *clasp* [5], *cmodels* [9] and *smodels* [20] using instances of several problems. The range of problems is limited somewhat by the availability of compatible programs. While projects such as Asparagus [2] have a wide variety of ASP problem instances available, the majority use *lparse* features unsupported by *Galliwasp*. Work is in progress to support these features.

The times for *Galliwasp* in Table 1 are for the interpreter reading compiled versions of each problem instance. The times for the remaining solvers are for

the solver reading problem instances from files in the *smodels* input language. A timeout of 600 seconds was used, with the processes being automatically killed after that time and the result being recorded as N/A in the table.

Galliwasp is outperformed by *clasp* in all but one case and outperformed by *cmodels*, in most cases, but the results are usually comparable. *Galliwasp*'s performance can vary significantly, even between instances of the same problem, depending on the amount of backtracking required. In the case of programs that timed out or took longer than a few seconds, the size and structure of the programs simply resulted in a massive amount of backtracking.

We believe that manually reordering clauses or changing the grounding method will improve performance in most cases. In particular, experimentation with manual clause reordering has resulted in execution times on par with *clasp*'s. However, the technique makes performance dependent on the query used to determine the optimal ordering. As a result, the technique is not general enough for use in performance comparisons, and no reordering of clauses was used to obtain the results reported here.

6 Related and Future Work

Galliwasp's goal-directed execution method is based on a previously published technique, but has been significantly refined [10, 19]. In particular, the original algorithm was limited to ASP programs which were call-consistent or order-consistent [19]. Additionally, the implementation of the previous algorithm was written as a Prolog meta-interpreter, and incapable of providing results comparable to those of existing ASP solvers. *Galliwasp*, written in C, is the first goal-directed implementation capable of direct comparison with other ASP solvers.

Various other attempts have been made to introduce goal-directed execution to ASP. However, many of these methods rely on modifying the semantics or restricting the programs and queries which can be used, while *Galliwasp*'s algorithm remains consistent with stable model semantics and works with any arbitrary program or query. For example, the revised Stable Model Semantics [21] allows goal-directed execution [22], but does so by modifying the stable model semantics underlying ASP. SLD resolution has also been extended to ASP through credulous resolution [3], but with restrictions on the type of programs and queries allowed. Similar work may also be found in the use of ASP with respect to abduction [1], argumentation [13] and tableau calculi [6].

Our plans for future work focus on improving the performance of *Galliwasp* and extending its functionality. In both cases, we are planning to investigate several possible routes.

With respect to performance, we are looking into exercising better control over the grounding of programs, or eliminating of grounding altogether. The development of a grounder optimized for goal-directed execution is being investigated ([15], forthcoming), as is the extension of answer set programming to predicates (preliminary work can be found in [18]). Both have the potential to

significantly improve *Galliwasp*'s performance by reducing the amount of backtracking that results from the use of grounded instances produced using *lparse*.

In the area of functionality, we plan to expand *Galliwasp*'s input language to support features commonly allowed by other solvers, such as constraint literals, weight literals and choice rules. Extensions of *Galliwasp* to incorporate constraint logic programming, probabilistic reasoning, abduction and or-parallelism are also being investigated. We believe that these extensions can be incorporated more naturally into *Galliwasp* given its goal-directed execution strategy.

The complete source for *Galliwasp* is available from the authors at [16].

7 Conclusion

In this paper we introduced the goal-directed ASP solver *Galliwasp* and presented *Galliwasp*'s underlying algorithm, limitations, design, implementation and performance. *Galliwasp* is the first approach toward solving answer sets that uses goal-directed execution for general answer set programs. Its performance is comparable to other state-of-the-art ASP solvers. *Galliwasp* demonstrates the viability of the top-down technique. Goal-directed execution can offer significant potential benefits. In particular, *Galliwasp*'s underlying algorithm paves the way for ASP over predicates [18] as well as integration with other areas of logic programming. Unlike in other ASP solvers, performance depends on the amount of backtracking required by a program: at least to some extent this can be controlled by the programmer. Future work will focus on improving performance and expanding functionality.

Acknowledgments: The authors are grateful to Feliks Kluźniak for many discussions and ideas.

References

1. Alferes, J.J., Pereira, L.M., Swift, T.: Abduction in Well-Founded Semantics and Generalized Stable Models via Tabled Dual Programs. Theory and Practice of Logic Programming 4, 383–428 (July 2004)
2. Asparagus: `http://asparagus.cs.uni-potsdam.de` (2012)
3. Bonatti, P.A., Pontelli, E., Son, T.C.: Credulous Resolution for Answer Set Programming. In: Proceedings of the 23rd national conference on Artificial Intelligence - Volume 1. pp. 418–423. AAAI'08, AAAI Press (2008)
4. De Raedt, L., Kimmig, A., Toivonen, H.: ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In: Proceedings of the 20th international joint conference on Artifical Intelligence. pp. 2468–2473. IJCAI'07, Morgan Kaufmann (2007)
5. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Clasp: A Conflict-Driven Answer Set Solver. In: Proceedings of the 9th international conference on Logic Programming and Nonmonotonic Reasoning. pp. 260–265. LPNMR'07, Springer-Verlag (2007)

6. Gebser, M., Schaub, T.: Tableau Calculi for Answer Set Programming. In: Proceedings of the 22nd international conference on Logic Programming. pp. 11–25. ICLP'06, Springer-Verlag (2006)
7. Gelfond, M.: Representing Knowledge in A-Prolog. In: Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II. pp. 413–451. Springer-Verlag (2002)
8. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Proceedings of the Fifth international conference on Logic Programming. pp. 1070–1080. MIT Press (1988)
9. Giunchiglia, E., Lierler, Y., Maratea, M.: SAT-Based Answer Set Programming. In: Proceedings of the 19th national conference on Artifical Intelligence. pp. 61–66. AAAI'04, AAAI Press (2004)
10. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive Logic Programming and Its Applications. In: Proceedings of the 23rd international conference on Logic Programming. pp. 27–44. ICLP'07, Springer-Verlag (2007)
11. Gupta, G., Pontelli, E., Ali, K.A., Carlsson, M., Hermenegildo, M.V.: Parallel Execution of Prolog Programs: A Survey. ACM Transactions on Programming Languages and Systems 23(4), 472–602 (July 2001)
12. Jaffar, J., Lassez, J.L.: Constraint Logic Programming. In: POPL. pp. 111–119 (1987)
13. Kakas, A., Toni, F.: Computing Argumentation in Logic Programming. Journal of Logic and Computation 9(4), 515–562 (1999)
14. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive Logic Programming. Journal of Logic and Computation 2(6), 719–770 (1992)
15. Marple, K.: Design and Implementation of a Goal-directed Answer Set Programming System. Ph.D. thesis, University of Texas at Dallas
16. Marple, K.: Galliwasp. `http://www.utdallas.edu/~kbm072000/galliwasp/`
17. Marple, K., Bansal, A., Min, R., Gupta, G.: Goal-Directed Execution of Answer Set Programs. Tech. rep., University of Texas at Dallas (2012), `http://www.utdallas.edu/~kbm072000/galliwasp/publications/goaldir.pdf`
18. Min, R.: Predicate Answer Set Programming with Coinduction. Ph.D. thesis, University of Texas at Dallas (2010)
19. Min, R., Bansal, A., Gupta, G.: Towards Predicate Answer Set Programming via Coinductive Logic Programming. In: AIAI. pp. 499–508. Springer (2009)
20. Niemelä, I., Simons, P.: Smodels - An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs. In: Logic Programming And Nonmonotonic Reasoning, Lecture Notes in Computer Science, vol. 1265, pp. 420–429. Springer-Verlag (1997)
21. Pereira, L., Pinto, A.: Revised Stable Models - A Semantics for Logic Programs. In: Progress in Artificial Intelligence, Lecture Notes in Computer Science, vol. 3808, pp. 29–42. Springer-Verlag (2005)
22. Pereira, L., Pinto, A.: Layered Models Top-Down Querying of Normal Logic Programs. In: Practical Aspects of Declarative Languages, Lecture Notes in Computer Science, vol. 5418, pp. 254–268. Springer-Verlag (2009)
23. Simon, L.: Extending Logic Programming with Coinduction. Ph.D. thesis, University of Texas at Dallas (2006)

Relevancy analysis as a basis for improved tabling in CHRiSM

Colin J. Nicholson[1], Danny De Schreye[2]

K U Leuven[1,2]
Celestijnenlaan 200A
B-3001 Heverlee
Belgium
colin.nicholson@cs.kuleuven.be[1]
danny.deschreye@cs.kuleuven.be[2]

August 20, 2012

Abstract

CHRiSM is a probabilistic logic programming language which combines CHR, a high level programming language that uses multi-headed rewrite rules, and PRISM, a Prolog extension that allows for probabilistic predicates and learning.

Tabling is often important for efficient computation in probabilistic contexts, because it can avoid frequent re-computation of probabilistic distributions. PRISM contains a tabling system which was designed to work with Prolog, but it is unclear how to make it work correctly and efficiently with the constraint store of CHR. As a result, tabling has not been used in CHRiSM.

In this paper, we propose a method to make CHRiSM programs more accommodating to the tabling system of PRISM, by performing an analysis of which CHRiSM constraints can affect the outcome of a given goal and determining when it is safe to use tabled results of subgoals.

Keywords: CHR, tabling, probabilistic logic programming

1 Introduction

Constraint Handling Rules (CHR) is a programming language extension introduced by Frühwirth [2] that is implemented on top of a host language, such as Prolog. CHR enables the use of multi-headed rules and was originally designed for constraint solvers. PRISM (PRogramming In Statistical Modeling) [5] is an extension of Prolog developed by Sato which enables the usage of probabilistic

predicates and probabilistic tasks, such as expectation-maximization learning. CHRiSM (CHance Rules induce Statistical Models) [9] is a programming language that translates to CHR(PRISM).

Like CHR, CHRiSM uses a multiset of constraints to determine whether to consider applying a rule or not. Since CHRiSM is probabilistic, a rule can be considered but still not fire, depending on the outcome of a probability expression. The multiset of constraints is called a constraint store, or just store.

Tabling [10] is a technique of storing calculated solutions that can be retrieved later instead of performing redundant computations. This approach was first used to increase the speed of the evaluation of functions [3]. Tabling has been problematic for CHR in the past, even leading some to propose new semantics for CHR that works well with tabling [4]. The tabling system used in PRISM was designed to avoid unnecessary evaluation of Prolog goals [12]. Tabling is important in a probabilistic computation domain due to the fact that the number of probabilistic choices that can lead to a goal can be exponential. The PRISM tabling system works well with storing predicate calls and their solutions, but problems can arise when attempting to store a CHR constraint store in a similar manner.

For example, consider the predicate c(In,Out) from a CHR(PRISM) program which takes in a constraint store and returns the resulting constraint store after potentially firing one or more rule(s) with "c" in the head.

Consider the cases where the input store is [c], [c,g], and [c,g,g]. If tabling were used in CHRiSM, c([c],Out), c([c,g],Out), and c([c,g,g],Out) would be stored in three separate tables, and we could not simply use the tabled result of c([c],Out) for the latter two calls. If we were able to determine that the "g" constraints could not affect the result of c([c],Out), we could remove the "g" constraints before calling c([c],Out) and add them back to the store after retrieving the tabled results for c([c],Out).

However, we could not remove the "g" constraints from the store without first determining that they can not affect the outcome of c([c],Out), as it is possible that a multi-headed rule may fire when starting with store [c,g] that would not fire if the store was [c]. We need a method for making such a determination.

In this paper, we introduce the concept of relevancy for a given CHRiSM store. The motivation for this is to make the tabling system of PRISM functional in CHRiSM. Tabling can work in CHRiSM if subsets of constraint stores (subgoals) are used as inputs when rules are considered in CHR(PRISM) instead of the entire constraint store. But before using a subgoal as input, we must first be able to determine that we will not get a different result by starting with the complete store instead.

Overview of the paper. This paper will first introduce the CHRiSM language by summarizing previous work of Sneyers et. al (Section 2). Section 3 defines relevancy in CHRiSM and what it means to be part of a *relevancy group*. In Section 4 we present a method for determining whether a multiset of constraints is relevant for a given goal that uses relevancy groups. Section 5 discusses our initial testing and results. We end with conclusions and future work ideas.

2 CHRiSM

2.1 Syntax of CHRiSM

The syntax and semantics of CHRiSM were introduced by Sneyers et. al in [9] and will be restated here. A CHRiSM program $\mathcal{P}$ is a series of chance rules. These chance rules rewrite a multiset, $, of data elements called CHRiSM constraints. A constraint is of the form c(X_1, ..., X_n), with a predicate c of arity n and (Prolog term) arguments X_1, ..., X_n. The multiset $ is the constraint store. The initial store is referred to as the query or goal, while the final store is called the answer or result. The final store is obtained by exhaustive rule application. There are three types of CHRiSM chance rules:

Simplification rules: P ?? h_1,...,h_n <=> g_1,...,g_m | b_1,...,b_o
Propagation rules: P ?? h_1,...,h_n ==> g_1,...,g_m | b_1,...,b_o
Simpagation rules: P ?? h_1,...,h_l \ h_{l+1}, ...,h_n <=> g_1, ...,g_m | b_1, ...,b_o

Here n and m are ≥ 1 and P is a probability expression (defined below). In Simplification rules, h_1,...,h_n are removed head constraints, which are removed from the store as the rules are applied. In Propagation rules, h_1,...,h_n are kept head constraints, which remain in the store. In Simpagation rules, the constraints before the "\" are kept head constraints, while the constraints after the "\" are removed head constraints. In all rules g_1,...,g_m are (Prolog goal) guards which prevent the rule from firing unless guard conditions are met. If no guard is specified, the default guard is "true," which will cause the rule to always fire if it is applied. Also in all rules b_1,...,b_o are a conjunction of Prolog goals, CHRiSM constraints, and probabilistic disjunctions (defined below) of bodies.

If the constraint store $ contains elements that match with the head of a chance rule and the guard is satisfied, rule application is considered. A rule instance is the subset of $ that corresponds to the head of the rule. The rule instance is ignored or it leads to a rule being applied, depending on the probability expression P (every rule instance can only be considered once). With rule application, any removed head constraints are removed from $ and the body is executed, leading to Prolog goals being called and CHRiSM constraints being added to the store.

A probability expression P can be:

1) A number from 0 to 1. This number indicates the probability the rule will fire. A rule with a P of 1 will always be applied and the "1 ??" is not necessary for such a rule. A rule with a P of 0 will never be applied.

2) An expression of the form eval(E). Here E is an arithmetic expression in Prolog which, when evaluated, indicates the probability that the rule will fire. Note: E must be ground when the rule is considered.

3) An experiment name: In other words, a Prolog term which must be ground when the rule is considered. The probability distribution is unknown and will initially be set to a uniform distribution (0.5 for rule probabilities), but can be

changed manually using the builtin set_sw/2 or the EM-learning algorithm, both from PRISM. The arguments of the experiment name can include conditions (cond C) which are evaluated and replaced with either "yes" (if call(C) succeeded) or "no" (if call(C) failed).

4) Omitted, in which case the rule will be treated as if it has a P of 1.

The body of a CHRiSM rule may contain probabilistic disjunctions of two types:

- LPAD-style probabilistic disjunctions [11] of the form "D1:P1 ; ... ; Dn:Pn," where a disjunct Di is chosen with probability Pi. Here the probabilities must sum to 1.
- CHRiSM-style probabilistic disjunctions of the form "P ?? D1 ; ... ; Dn", where P is an experiment name that determines the probability distribution.

These disjunctions are committed choice: once a disjunct is chosen, the choice is not undone later.

2.2 Abstract Operational Semantics of CHRiSM

The abstract operational semantics of CHRiSM mostly follows the abstract operational semantics of CHR [6]. The execution states are similar with the exception that CHRiSM has a unique failed execution state, denoted as "fail," and does not distinguish between different failed states.

Definition 1 (Identified Constraint): An identified constraint, c#i, is a CHRiSM constraint c that has been labeled with a unique integer identifier i to distinguish between different copies of the same constraint. For our purposes, id(c#i) = i and chr(c#i) = c. For sets, id(S) = {i|c#i ∈ S} and chr(S) = {c|c#i ∈ S}. Similarly, id and chr will also be used on multisets and sequences.

Definition 2 (Execution State): An execution state σ is a tuple $\langle \mathbb{G},\mathbb{S},\mathbb{B},\mathbb{T}\rangle_n$, where the goal, $\mathbb{G}$, is a multiset of constraints to be rewritten in solved form. The store, $\mathbb{S}$, is a set of identified constraints that can be matched with rules in the program $\mathcal{P}$ (here chr($\mathbb{S}$) is a multiset, while $\mathbb{S}$ is a set). The built-in store, $\mathbb{B}$, is the conjunction of all Prolog goals that have been called so far. The history, $\mathbb{T}$, is a set of tuples that keep track of the identifiers of CHRiSM constraints that fired a rule and the rule number. This prevents certain constraints from firing the same rules an infinite amount of times. The counter, n, represents the next free identifier.

Σ^{CHR} represents the set of all execution states σ_0, σ_1,... and $\mathcal{D}_{\mathcal{H}}$ denotes the theory defining the Prolog built-in goals and predicates used in the CHRiSM program. For a CHRiSM program $\mathcal{P}$, the transitions (each annotated with a probability) are defined by the binary relation $\underset{\mathcal{P}}{\rightarrowtail} \subset \Sigma^{CHR} \times \Sigma^{CHR}$ in the following ways (with $\uplus$ representing multiset union and $\bar{\exists}_A$B denoting $\exists$x$_1$, ...,

x_n : B, with $\{x_1, \ldots, x_n\}$ = vars(B) \ vars(A), where vars(A) are the (free) variables in A; if A is omitted it is empty):

The transition rules for the abstract operational semantics of CHRiSM are:

1) Fail: $\langle [b] \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{1}{\underset{\mathcal{P}}{\hookrightarrow}}$ fail

where b is a built-in (Prolog) constraint and $\mathcal{D}_{\mathcal{H}} \vDash \neg\, \bar{\exists}(\mathbb{B} \wedge b)$.

2) Solve: $\langle [b] \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{1}{\underset{\mathcal{P}}{\hookrightarrow}} \langle \mathbb{G}, \mathbb{S}, b \wedge \mathbb{B}, \mathbb{T} \rangle_n$

where b is a built-in (Prolog) constraint and $\mathcal{D}_{\mathcal{H}} \vDash \bar{\exists}(\mathbb{B} \wedge b)$.

3) Introduce: $\langle [c] \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{1}{\underset{\mathcal{P}}{\hookrightarrow}} \langle \mathbb{G}, \{c\#n\} \cup \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_{n+1}$

where c is a CHRiSM constraint.

4) Probabilistic-Choice: $\langle [d] \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{p_i}{\underset{\mathcal{P}}{\hookrightarrow}} \langle \{d_i\} \uplus \mathbb{G}, \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n$

where d is a probabilistic disjunction of the form d_1: p_1 ; ... ; d_k:p_k or of the form P ?? d_1 ; ... ; d_k, where the probability distribution given by P assigns the probability p_i to the disjunct d_i.

5) Maybe-Apply:

$\langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{1-p}{\underset{\mathcal{P}}{\hookrightarrow}} \langle\ \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$

$\langle \mathbb{G}, H_1 \uplus H_2 \uplus \mathbb{S}, \mathbb{B}, \mathbb{T} \rangle_n \overset{p}{\underset{\mathcal{P}}{\hookrightarrow}} \langle\ B \uplus \mathbb{G}, H_1 \uplus \mathbb{S}, \theta \wedge \mathbb{B}, \mathbb{T} \cup \{h\} \rangle_n$

where the r-th rule of $\mathcal{P}$ is of the form P ?? H_1'\ H_2' <=> G | B, θ is a matching substitution such that chr(H_1) = θ(H_1') and chr(H_2) = θ(H_2'), h = (r, id(H_1) id(H_2)) $\notin \mathbb{T}$, and $\mathcal{D}_{\mathcal{H}} \vDash \mathbb{B} \rightarrow \bar{\exists}_{\mathbb{B}}\ (\theta \wedge G)$. If P is a number, then p = P. Otherwise p is the probability assigned to the success branch of P.

Execution involves exhaustingly applying the transition rules starting from an initial state of the form $\langle Q,\emptyset,true,\emptyset \rangle_0$ and performing a random walk from this root in the directed acyclic graph, defined by transition relation $\overset{}{\underset{\mathcal{P}}{\hookrightarrow}}$, until a leaf node (final state) is reached. In a path from an initial state to state σ, the probability of σ is the product of the probabilities along the path. A series of k $\geq$ 0 transitions

$\sigma_0 \overset{p1}{\underset{\mathcal{P}}{\hookrightarrow}} \sigma_1 \overset{p2}{\underset{\mathcal{P}}{\hookrightarrow}} \sigma_2 \overset{p3}{\underset{\mathcal{P}}{\hookrightarrow}} \ldots \overset{pk}{\underset{\mathcal{P}}{\hookrightarrow}} \sigma_k$

is denoted $\sigma_0 \overset{p}{\underset{\mathcal{P}}{\hookrightarrow}}{}^{*}\ \sigma_k$, where p = $\Pi_{i=1}^{k} p_i$ if k > 0 and p = 1 otherwise. This type of series of transitions is called a derivation of probability p if σ_0 is an initial state and σ_k is a final state. The probability of a derivation is defined by: prob($\sigma_0 \overset{p}{\underset{\mathcal{P}}{\hookrightarrow}}{}^{*}\ \sigma_k$) = p. Note that, for a CHRiSM program, if all rule probabilities are 1 and the program contains no probabilistic disjunctions, the abstract operational semantics of CHRiSM is equivalent to the abstract operational semantics of CHR [6].

In our examples, we will usually omit probabilities, because the problem that needs to be addressed in order to facilitate tabling is the same for CHR and for CHRiSM. This also implies that the solution that we will propose is a solution for both languages.

2.3 Distribution Semantics

CHRiSM uses the refined operational semantics which follow the same rules as the refined semantics of CHR [1], with the exception that in CHRiSM, the propagation history is not stored.

The main difference between the refined and abstract semantics of CHR involves the conditions under which a rule can fire. In the abstract semantics, a rule can fire on any subset of the current CHR store. In the refined semantics, there is an execution stack and the constraint currently being considered (at the top of the stack) must match the head of a rule for the rule to fire. If the same constraint appears in different rules (or multiple times in the head of the same rule), we follow a top-down matching order for different rules and right-to-left matching order in the head of a rule.

3 Relevancy Analysis of CHRiSM constraints

Because CHRiSM is a nondeterministic language, it is important to note that, for our purposes, the result of the execution of a goal should be represented as a multiset of possible final stores. Consider the following program.

Example 1:

```
a <=> b:  0.5 ; c:  0.5.
b <=> d.
c <=> e.
```

For this program, the query [a] can have two possible results, [d] or [e], so the tabled result for a query must be a multiset of all possible results. So, for this program, the multiset of all possible results for [a] is [[d],[e]].

Definition 8 (Flatten): Let A be a set. We denote $\overset{\sim A}{2}$ as the set of all multisets of elements of A. Then, with $\times$ denoting the Cartesian product of multisets, we define a function Flatten, denoted F, such that
$F(\overset{\sim A}{2} \times \overset{\sim A}{2}) \rightarrow \overset{\sim A}{2}$: (A,B) $\rightarrow$ A $\uplus$ B.

Definition 9 (Relevance): For a CHRiSM program $\mathcal{P}$, with R(Q) denoting the multiset of all possible final stores resulting from the execution of goal Q, goal Q2 is relevant with respect to goal Q1 if R(Q1 $\uplus$ Q2) $\neq$ F(R(Q1) $\times$ R(Q2)).

For the program in Example 1, we have R([a]) = [[d],[e]] and R([c]) = [[e]], and we can say [c] is not relevant to [a] since R([a,c]) = [[d,e],[e,e]], which is the same as F([[d],[e]] $\times$ [[e]]). However, this is not always the case as can be seen in the following program.

Example 2:

```
d, a <=> b.
```

```
a <=> c.
```

For this program, R([d]) = [[d]] and R([a]) = [[c]], while R([d,a]) = [[b]]. F(R([d]) × R([a])) = [[d,c]], and since R([d,a]) ≠ F(R([d]) × R([a])), we conclude that [a] is relevant to [d].

Every constraint in the head or body of a rule in a CHRiSM program $\mathcal{P}$ is assigned an occurrence number, used to distinguish between different occurrences of the same constraint. These number should not be confused with the identifiers of Definition 1. The occurrences are numbered in top-down, left-to-right rule order.

Example 3:

```
a <=> b.
b <=> c.
d,c <=> e.
c <=> f.
```

This program, with occurrence numbers added, is as follows:

a_1 <=> b_1.
b_2 <=> c_1.
d_1,c_2 <=> e_1.
c_3 <=> f_1.

Definition 10 (Input and Output Constraints): For a CHRiSM program $\mathcal{P}$, every constraint occurrence $C_{n,i}$ in the head of a rule R is called an *input constraint*, while every constraint occurrence D_m in the body of a rule R is called an *output constraint*. For the input constraints, the extra subscripts "i" can take the value "k" denoting that the head constraint is kept, or the value "nk" denoting that it is not kept (removed). The set of input constraints for rule R is denoted as In_R, while the set of output constraints for rule R is denoted as Out_R. The set of all input constraints for program $\mathcal{P}$ is denoted as $In_{\mathcal{P}}$ and the set of all output constraints for program $\mathcal{P}$ is $Out_{\mathcal{P}}$. The set of all In_R sets for program $\mathcal{P}$ is denoted as $InS_{\mathcal{P}}$ and the set of all Out_R sets for program $\mathcal{P}$ is $OutS_{\mathcal{P}}$.

For the first rule of Example 3, the set of input constraints, In_1, is $\{a_{1,nk}\}$ and the set of output constraints, Out_1, is $\{b_1\}$.

A directed hypergraph is a pair (V, A) consisting of a set, V, of vertices and a set, A, of directed hyperarcs. A directed hyperarc is an ordered pair (X, Y), where X and Y are sets of vertices, which is considered directed from X to Y.

Definition 11 (Rule Transition): A *rule transition* for rule R in CHRiSM program $\mathcal{P}$ with the set of input constraints In_R and the set of output constraints

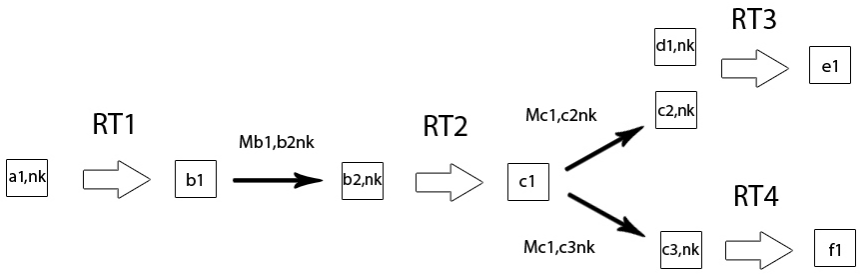


Figure 1: Hypergraph for Example 3

Out_R is a directed arc RT_R, where $\text{RT}_R = (\text{In}_R, \text{Out}_R)$. $\text{RT}_\mathcal{P}$ represents the set of all rule transitions in $\mathcal{P}$.

For the program in Example 3 we have the following:

$\text{In}_1 = \{\text{a}_{1,nk}\}$	Out_1 is $\{\text{b}_1\}$	$\text{RT}_1 = (\{\text{a}_{1,nk}\},\{\text{b}_1\})$
$\text{In}_2 = \{\text{b}_{2,nk}\}$	Out_2 is $\{\text{c}_1\}$	$\text{RT}_2 = (\{\text{b}_{2,nk}\},\{\text{c}_1\})$
$\text{In}_3 = \{\text{d}_{1,nk}, \text{c}_{2,nk}\}$	Out_3 is $\{\text{e}_1\}$	$\text{RT}_3 = (\{\text{d}_{1,nk}, \text{c}_{2,nk}\},\{\text{e}_1\})$
$\text{In}_4 = \{\text{c}_{3,nk}\}$	Out_4 is $\{\text{f}_1\}$	$\text{RT}_4 = (\{\text{c}_{3,nk}\},\{\text{f}_1\})$

Definition 12 (Match Transition): Let $\text{C}_{n,i}$ be an input constraint and C_m be an output constraint in program $\mathcal{P}$, such that there exists a unification θ : $\text{C}_{n,i}\theta = \text{C}_m\theta$. Then $\text{M}_{cm,cni} = (\text{C}_m, \text{C}_{n,i})$ is called the corresponding *match transition*. The set of all match transitions for $\mathcal{P}$ is denoted as $\text{M}_\mathcal{P}$.

Figure 1 shows a directed hypergraph representing the match and rule transitions of Example 3.

Definition 13 (Relevancy Net): A *relevancy net*, $\text{RN}_\mathcal{P}$, of a CHRiSM program $\mathcal{P}$ is a tuple $\langle \text{In}_\mathcal{P}, \text{Out}_\mathcal{P}, \text{InS}_\mathcal{P}, \text{OutS}_\mathcal{P}, \text{RT}_\mathcal{P}, \text{M}_\mathcal{P} \rangle$, where $\text{In}_\mathcal{P}$, $\text{Out}_\mathcal{P}$, $\text{InS}_\mathcal{P}$, $\text{OutS}_\mathcal{P}$, $\text{RT}_\mathcal{P}$, and $\text{M}_\mathcal{P}$ are as previously defined.

Next, we introduce the concept of a trigger set. Given some rule in a program, the trigger set will characterize all sets of input constraints that could possibly trigger this rule, directly or after a number of intermediate transitions.

Definition 14 (Direct Trigger Set): Let I be a set of input constraints and let R be a rule in $\mathcal{P}$. Let S_{out} be a maximal subset of R_{out} and S_{in} a maximal subset of I, such that every $\text{c}_{out} \in \text{S}_{out}$ has a match transition to $\text{c}_{in} \in$

S_{in} and these match transitions define a one-to-one mapping from S_{out} onto S_{in}. Then, the set $(I \setminus S_{in}) \cup In_R$ is a *direct trigger set* of I.

Example 4:
d_1,e_1 <=> b_1,c_1.
f_1 <=> a_1.
b_2,c_2,a_2 <=> g_1.
g_2,h_1 <=> i_1.

In this example, consider the set of input constraints $I = \{b_{2,nk},c_{2,nk},a_{2,nk},h_{1,nk}\}$. Consider rule R1, with $Out_{R1} = \{b_1,c_1\}$. There are maximal subsets $S_{out} = \{b_1,c_1\}$ of Out_{R1} and $S_{in} = \{b_{2,nk},c_{2,nk}\}$ of I, such that the match transitions between these define a one-to-one onto mapping from S_{out} to S_{in}. The corresponding direct trigger set of I is $(I \setminus S_{in}) \cup In_{R1} = \{d_{1,nk},e_{1,nk},a_{2,nk},h_{1,nk}\}$.

A set I can have several direct trigger sets. This is clear if we consider different rules from the program. But even if we fix the rule R, there can be different maximal sets, and therefore different trigger sets.

By $\xrightarrow{T}$, we will denote the direct trigger set relation on $2^{I_P} \times 2^{I_P}$:

$\xrightarrow{T} : 2^{I_P} \times 2^{I_P}$: $I \xrightarrow{T} J$ if J is a direct trigger set of I. A series of k direct trigger relations, $I_1 \xrightarrow{T} I_2 \xrightarrow{T} \ldots \xrightarrow{T} I_k \xrightarrow{T} I_{k+1}$, will be denoted as $I_1 \xrightarrow{T}{}^* I_{k+1}$.

Definition 15 (Trigger Set): Let In_R be the input constraint set of a rule R in $\mathcal{P}$. A *trigger set* for In_R is any set of input constraints J, such that $In_R \xrightarrow{T}{}^* J$.

Returning to Example 4, take the input constraint set of R4 : $In_{R4} = \{g_{2,nk},h_{1,nk}\}$. Using rule R3, there is a direct trigger set $\{b_{2,nk},c_{2,nk},a_{2,nk},h_{1,nk}\}$, using the one-to-one and onto match transition $\{g_1\} \rightarrow \{g_{2,nk}\}$.

It follows from Example 4 that $\{d_{1,nk},e_{1,nk},a_{2,nk},h_{1,nk}\}$ is also a trigger set for In_{R4}, using R1. If we use R2 instead, another trigger set for In_{R4} is $\{b_{2,nk},c_{2,nk},f_{1,nk},h_{1,nk}\}$.

Note that, although one could consider arbitrarily long series of direct trigger relations, in the context of recursive programs, the number of different trigger sets is finite. The reason for this is that a trigger set is a subset of In_P, and In_P itself is finite. Therefore, we can compute the set of all trigger sets in finite time, using, for instance, a fix point computation.

We now turn to identifying constraints that might be relevant in the context of tabling. First note that if a CHRiSM program only has single-headed rules, then the computation for some constraint can not be influenced by the presence of another constraint in the constraint store. Only multi-headed rules are dependent on multiple constraints in the store and only these influence whether certain constraints are relevant for the computation of others.

The following notion of a relevance group gives a global (goal independent) characterization of which constraints could influence each other's computations.

To introduce the concept, we need a second function chr$^\sim$, similar to chr, that removes subscripts of our second type from constraints. Again, for sets S $\subseteq$ In$_P$ $\cup$ Out$_P$, chr$^\sim$(S) = {chr$^\sim$(c) | c $\in$ S}.

Definition 16 (Relevancy Group): Let $\mathcal{P}$ be a CHRiSM program. A set of constraints RG is a relevancy group if either

- RG = chr$^\sim$(In$_R$), where #In$_R$ $\geq$ 2, or
- RG = chr$^\sim$(T), where T is a trigger set for In$_R$ and #In$_R$ $\geq$ 2.

Reconsider Example 3. The only In$_R$ set with a cardinality larger than 1 is In$_3$ = {$d_{1,nk}$, $c_{2,nk}$}. The only trigger sets for In$_3$ are {$d_{1,nk}$, $a_{1,nk}$} and {$d_{1,nk}$, $b_{2,nk}$}. Thus the relevancy groups of Example 3 are {d, c}, {d, a} and {d, b}.

The notion of a relevancy group is an abstraction of the multisets of constraints that could possibly trigger (potentially after several derivation steps) a multi-headed rule to fire.

In the move from a real multiset that could trigger such a rule to fire and the relevancy group itself, there are two aspects that are abstracted:

- the multiplicity with which a constraint occurs in the multiset, and
- which actual instance of the input constraint in the relevancy group occurs in the multiset.

Example 5:

```
a(1),b(2) <=> s(1).
g(X), g(Y), s(Z) <=> m(X,Y,Z).
```

For this program, the relevancy groups are {a(1), b(2)}, {g(X), g(Y), s(Z)}, and {g(X), g(Y), a(1), b(2)}. Actual queries or stores that could trigger a multi-headed rule in this program are, for instance, [a(1), b(2), g(1)], [g(1), g(2), s(X)] and [g(X1), g(X2), g(X3), a(1), b(2)]. In general, the correspondence between the relevancy groups and queries that potentially trigger multi-headed rules is given in the next definition.

Definition 17 (Constraint Multisets Represented by a Relevancy Group): Let RG = {c_1, ..., c_n} be a relevancy group of program $\mathcal{P}$. Then RG represents all multisets of constraints [c_1^1, ..., $c_1^{m_1}$, ..., c_n^1, ..., $c_n^{m_n}$] such that for all i $\in$ [1,n] and j $\in$ [1,m$_i$]: c_i^j matches with c_i and for all i $\in$ [1,n], m$_i$ $\geq$ 1.

The notion of a relevancy group is defined for a program as a whole. If we start off with a specific query, we may want to relate the relevancy groups to that query.

Definition 17 (Relevancy Groups of a Query): Let Q be a query for a program $\mathcal{P}$. A relevancy group for Q is a set S, such that

- the elements of S are members of the multiset Q, and
- there exists a relevancy group RG of $\mathcal{P}$ and an onto mapping m: S $\rightarrow$ RG, such that for each s $\in$ S, s matches with m(s).

We denote the set of all relevancy groups for a query Q as RG_Q.

Consider the program in Example 5, with relevancy groups {a(1), b(2)}, {g(X), g(Y), s(Z)}, and {g(X), g(Y), a(1), b(2)}. For query Q = [g(1), a(1), b(2)], the only relevancy groups for Q is {a(1), b(2)}. For the query Q = [g(1), g(2), a(2), s(3)] there is again one relevancy group: RG_Q = {{g(1), g(2), s(3)}}. By the construction of the notion of relevancy groups for a query, we now have the following proposition.

Proposition 1: For any CHRiSM program $\mathcal{P}$ and query Q, if there is a multisubset M of Q, such that the constraints in M cause the triggering of a multi-headed rule in $\mathcal{P}$, then the set of all distinct elements of M is a member of RG_Q.

4 Using Relevancy Groups for Tabling

We now have the ability to solve the problem posed in the introduction of this paper. Our original goal was to use the tabled results of R([c]) for goals such as [c,g], and [c,g,g], if it was determined that the "g" constraints could not affect the outcome of R([c]). This is achieved by storing the results of our subgoal, [c], and re-using the results when we encounter the same subgoal at a later stage.

When we first consider a goal Q, we must divide Q into two multisubsets, Q1 and Q2, such that Q1 $\uplus$ Q2 = Q. After calculating R(Q1), we combine the constraints in Q2 with the result of R(Q1) to get the new store. For [c,g,g] as described above, Q1 = [c] and Q2 = [g,g]. Now we must illustrate the method of determining how to decide which constraints are in each subset, Q1 and Q2, with an example.

Example 6:

```
v <=> a.
c <=> b.
c,s <=> a,a.
c,a,a <=> g.
x,y <=> z.
c,c,c <=> p.
```

For this program, for query [c,s,x,s,y,v,x], the relevancy groups are RG_Q = {{c}, {c,s}, {c,a}, {x,y}, {c,v}}. The refined semantics dictate that we first

consider the leftmost constraint in the store, c. We want to find the "smallest" subquery of Q that could be tabled. Clearly the constraints s, a, and v are relevant for c. Therefore we consider subqueries Q1 = [c,s,s,v] and Q2 = [x,y,x] of Q. These two subqueries are not relevant to each other and tabling for Q1 can be reused in further computations for Q.

We propose a method for determining when it is safe to use tabled results of subgoals in CHRiSM, without leading to incorrect results, that uses relevancy groups. Note: for the following lemmas and proofs, it is assumed that Q1 and Q2 share no constraints that are matchable with the same literal. This will always be the case for our methods since Q1 will be made of all constraints in the store that match with the constraints in certain relevancy groups.

Lemma 1: For a CHRiSM program $\mathcal{P}$ with queries Q1 and Q2, TQ1 denoting all possible constraints that can be added to the store by executing R(Q1), and TQ2 denoting all possible constraints that can be added to the store by executing R(Q2), if no multiset of constraints in Q2 $\uplus$ TQ2 can combine with any multiset of constraints in Q1 $\uplus$ TQ1 to fire a multi-headed rule in $\mathcal{P}$, then F(R(Q1) $\times$ R(Q2)) = R(Q1 $\uplus$ Q2).

Proof: If the constraints in Q2 have no effect on Q1, we will get the same results if we start with query Q1 and combine the resulting final store of R(Q1) with the final store produced by R(Q2) or if we just execute R(Q1 $\uplus$ Q2).

Proposition 2: For a given CHRiSM program $\mathcal{P}$, with $RG_{\mathcal{P}}$ representing all possible relevancy groups in $\mathcal{P}$, let Q1 and Q2 be queries and Q1Q2 = Q1 $\uplus$ Q2.

If $RG_{Q1Q2} = RG_{Q1} \cup RG_{Q2}$, then R(Q1 $\uplus$ Q2) = F(R(Q1) $\times$ R(Q2)), and tabled results of R(Q1) can safely be combined with results of R(Q2).

Proof: We know from Proposition 1 that RG_{Q1}, RG_{Q2}, and RG_{Q1Q2} contain all possible combinations of multiple constraints in Q1, Q2, and Q1Q2, respectively, that can lead to multi-headed rules in $\mathcal{P}$ being fired. Since $RG_{Q1Q2} = RG_{Q1} \cup RG_{Q2}$, we know that no constraints of Q2 or constraints added to the store by Q2 combine with Q1 or constraints added to the store by Q1 to cause a rule to fire when executing R(Q1Q2) since in that case, RG_{Q1Q2} would contain a member not contained in the union of RG_{Q1} and RG_{Q2}, and it would be the case that $RG_{Q1Q2} \neq RG_{Q1} \cup RG_{Q2}$.

Since no constraints of Q2 (or constraints added to the store by Q2) will combine with a constraint or multiset of constraints in Q1 (or constraints added to the store by Q1) to fire a rule in $\mathcal{P}$, by Lemma 1 we can say F(R(Q1) $\times$ R(Q2)) = R(Q1 $\uplus$ Q2).

Reconsider Example 3, with relevancy groups {{d,c}, {d,a}, {d,b}}. For queries Q1 = [d,c] and Q2 = [f,f], we have Q1Q2 = [d,c,f,f]. RG_{Q1} = {{d,c}}, RG_{Q2} = {{ }}, and RG_{Q1Q2} = {{d,c}}. So, $RG_{Q1Q2} = RG_{Q1} \cup RG_{Q2}$. For this

program, R([d,c]) = [[e]], R([f,f]) = [[f,f]], and R([d,c,f,f]) = [[e,f,f]], and tabling is safe.

If we had instead considered queries Q1 = [d,f] and Q2 = [a,b], we would have RG_{Q1} = {{ }}, RG_{Q2} = {{ }} and RG_{Q1Q2} = {{d,a}, {d,b}}. This means we can not use the tabled result of R([d,f]) and combine it with R([a,b]), as it may give a different result from R([d,f,a,b]). Indeed, R(Q1) = [[d,f]], R(Q2) = [[f,f]] and R(Q1Q2) = [[e,f,f]].

5 Implementation and Evaluation

When a built-in of PRISM such as the Viterbi calculation is used, PRISM conducts explanation search. An explanation in PRISM is a conjunction of ground switch instances which occur in the derivation path of a sampling execution of a probabilistic goal. Explanation search finds all possible explanations for a given goal. Due to the fact that the number of possible explanations can be exponential, PRISM uses a tabling system during explanation search. PRISM's Viterbi computation built-ins give the most probable explanation for a given goal and other related information. To understand how irrelevant constraints can slow down explanation search, consider the following example.

Example 7:

```
values(arg(N), [1,2,3,4,5]).
set_sw(arg(n), [0.1, 0.2, 0.3, 0.1, 0.3]).
a ==> msw(getN(n),N), e(N), g :  0.7 ;
      msw(getN(n),N), e(N) : 0.3.
g,a <=> msw(getN(n),N), a, e(N), g :  0.3 ;
        msw(getN(n),N), a, e(N) : 0.7.
```

Table 1: Timings (in ms) of Viterbi computation of the program in Example 7 with and without the removal of e(N) constraints from the store.

Goal	Es Present	Es Removed
[a,a,a,a,a]	164	5
[a,a,a,a,a,a]	338	8
[a,a,a,a,a,a,a]	1916	4
[a,a,a,a,a,a,a,a]	11824	4
[a,a,a,a,a,a,a,a,a]	TIME OUT	13

For this program, we can see the e(N) constraints are not relevant for the 'a' constraints, while the 'g' constraints are relevant. To test the improved efficiency of tabling with smaller constraint stores, we ran PRISMs Viterbi computation on the goal of a multiset of 'a' constraints. Our experiments included input goals with five to nine 'a' constraints. Two implementations were considered: one which adds e(N) constraints to the store to be tabled and one which does

not include the e(N) constraints in the tabled store. For instance, for the latter and for the goal [a,a,a,a,a], the execution builds tables for [a,a,a,a,a], [a,a,a,a], [a,a,a], [a,a], [a] and for all the super-multisets of these, including an additional g. For the former the execution also builds tables for super-multisets of these, including also a growing number of e(1), e(2), ..., e(5) constraints. The results are shown in Table 1.

As can be seen, the tabling of stores with e(N) constraints in this example greatly slows down explanation search to the point where, after an input goal that is greater than 8 constraints, the Viterbi calculation is not possible in a reasonable amount of time. By tabling the stores without the e(N) constraints, the Viterbi built-in works with much greater efficiency. In fact, the timings are so small that their differences are not significant even if - as in our case - the average time of many experiments is taken.

Of course, in this example, we actually only need tables for [a] and [g,a]. This is because 'a' is not relevant for itself. However, due to our set abstraction in relevancy groups, our relevancy analysis can not detect this.

6 Conclusions and Future Work

In this paper, we developed a method for determining when it is safe to use tabled results of subgoals in CHRiSM. With this method, tabling is now an option in CHRiSM, while before the behavior of a growing constraint store made tabling impractical.

The immediate future work will be a more thorough implementation and experimental evaluation of tabling with relevancy groups. Initial tests involving tabling with smaller constraint stores have been very promising.

Further future work will focus on refining the current method to be able to determine which constraints are not relevant for a given subgoal using more in-depth methods.

A first extension is to refine our abstraction used in the relevancy groups to allow a distinction between one occurrence of a constraint versus two or more occurrences of the same constraint. Such an extension would solve the tabling for Example 7 - and many other programs - completely.

Another area which holds promise is the analysis of guards in CHRiSM rules. Guard reasoning has been done before in the area of removing redundant CHR guards [8]. It is possible that similar reasoning could be done in the area of determining which constraints in a given store could be relevant for a given goal not just in terms of whether they match the head of a multi-headed rule, but also if the argument(s) of the head constraints can satisfy the guard.

References

[1] Duck, G.J., Stuckey, P.J., Garca de la Banda, M., Holzbaur, C.: The Refined Operational Semantics of Constraint Handling Rules. ICLP 2004,

B. Demoen and V. Lifschitz, Eds. LNCS, vol. 3132, Springer, 90 - 104.

[2] Frühwirth, T.: Constraint simplification rules. Tech. Rep. ECRC-92-18, European Computer-Industry Research Centre, Munich, Germany, July 1992.

[3] Michie, D.: "Memo" functions and machine learning. Nature, April 1968, 19 - 22.

[4] Sarna-starosta, B., Ramakrishnan, C.R.: Compiling Constraint Handling Rules for Efficient Tabled Evaluation. 9th International Symposium on Practical Aspects of Declarative Languages (PADL), 2007.

[5] Sato, T.: A glimpse of symbolic-statistical modeling by PRISM. Journal of Intelligent Information Systems, 31(2):161176, 2008.

[6] Sneyers, J., Van Weert, P., Schrijvers, T., De Koninck, L.: As Time Goes By: Constraint Handling Rules A Survey of CHR Research from 1998 to 2007. Theory and Practice of Logic Programming, Vol. 10(1), January 2010, 1 - 47.

[7] Sneyers, J., Frühwirth, T.: Generalized CHR Machines. 5th Workshop on Constraint Handling Rules (CHR'08), Hagenberg, Austria, July 2008.

[8] Sneyers, J., Schrijvers, T., Demoen, B.: Guard Reasoning in the Refined Operational Semantics of CHR. LNAI, Special Issue on Recent Advances in Constraint Handling Rules, vol. 5388, October 2008, 213 - 244.

[9] Sneyers, J., Meert, W., Vennekens, J., Kameya, Y., Sato, T.: CHR(PRISM)-based probabilistic logic learning. 26th International Conference on Logic Programming, Edinburgh, UK, July 2010.

[10] Tamaki, H. Sato, T.: OLD resolution with tabulation. In Proceedings of the Third International Conference on Logic Programming, London, 1986, E. Shapiro, Ed., Lecture Notes in Computer Science, Springer-Verlag, 84 - 98.

[11] Vennekens, J., Verbaeten, S., Bruynooghe, M.: Logic programs with annotated disjunctions. In Demoen, B., Lifschitz, V., eds.: ICLP 04. LNCS, vol. 3132, Saint- Malo, France, Springer, (2004), 431 - 445.

[12] Zhou, N., Sato, T.: Efficient fixpoint computation in Linear Tabling. In Proceedings of the 5th ACM SIGPLAN International Conference on Principles an Practices of Declarative Programming, Uppsala, 2003, 275 - 283.

Computing More Specific Versions of Conditional Rewriting Systems *

Naoki Nishida[1] and Germán Vidal[2]

[1] Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan,
nishida@is.nagoya-u.ac.jp
[2] MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
gvidal@dsic.upv.es

Abstract. Rewrite systems obtained by some automated transformation often have a poor syntactic structure even if they have good properties from a semantic point of view. For instance, a rewrite system might have overlapping left-hand sides even if it can only produce at most one constructor normal form (i.e., value). In this paper, we propose a method for computing "more specific" versions of deterministic conditional rewrite systems (i.e., typical functional programs) by replacing a given rule (e.g., an overlapping rule) with a finite set of instances of this rule. In some cases, the technique is able to produce a non-overlapping system from an overlapping one. We have applied the transformation to improve the systems produced by a previous technique for function inversion with encouraging results (all the overlapping systems were successfully transformed to non-overlapping systems).

1 Introduction

Rewrite systems [4] form the basis of several rule-based programming languages. In this work, we focus on the so called *deterministic* conditional rewrite systems (DCTRSs), which are typical functional programs with local declarations [23]. When the rewrite systems are automatically generated (e.g., by program inversion [2, 14, 15, 17, 22, 27, 26, 28] or partial evaluation [1, 7, 8, 35]), they often have a poor syntactic structure that might hide some properties. For instance, the rewriting systems generated by program inversion sometimes have overlapping left-hand sides despite the fact that they actually have the *unique normal form* property w.r.t. constructor terms—i.e., they can only produce at most one constructor normal form for every expression—or are even confluent.

* This work has been partially supported by the Spanish *Ministerio de Economía y Competitividad (Secretaría de Estado de Investigación, Desarrollo e Innovación)* under grant TIN2008-06622-C03-02, by the *Generalitat Valenciana* under grant PROMETEO/2011/052, and by *MEXT KAKENHI* #21700011.

Consider, e.g., the following TRS (left) from [28] (where we use $[\_|\_]$ and nil as list constructors) and its inversion (right):

$$\begin{array}{ll} \mathsf{inc}(\mathsf{nil}) \to [\mathsf{0}] & \mathsf{inc}^{-1}([\mathsf{0}]) \to \mathsf{nil} \\ \mathsf{inc}([\mathsf{0}|xs]) \to [\mathsf{1}|xs] & \mathsf{inc}^{-1}([\mathsf{1}|xs]) \to [\mathsf{0}|xs] \\ \mathsf{inc}([\mathsf{1}|xs]) \to [\mathsf{0}|\mathsf{inc}(xs)] & \mathsf{inc}^{-1}([\mathsf{0}|ys]) \to [\mathsf{1}|\mathsf{inc}^{-1}(ys)] \end{array}$$

Now, observe that every instance of $\mathsf{inc}^{-1}(x)$ using a constructor term has a unique constructor normal form. However, this system is not confluent—consider, e.g., the reductions starting from $\mathsf{inc}^{-1}([\mathsf{0}])$—and, moreover some of the left-hand sides overlap, thus preventing us from obtaining a typical (deterministic) functional program. In this case, one can observe that the recursive call to inc^{-1} in the third rule can only bind variable ys to a non-empty list, say $[z|zs]$. Therefore, the system above could be transformed as follows:

$$\begin{array}{l} \mathsf{inc}^{-1}([\mathsf{0}]) \to \mathsf{nil} \\ \mathsf{inc}^{-1}([\mathsf{1}|xs]) \to [\mathsf{0}|xs] \\ \mathsf{inc}^{-1}([\mathsf{0}, z|zs]) \to [\mathsf{1}|\mathsf{inc}^{-1}([z|zs])] \end{array}$$

Now, the transformed system is non-overlapping (and confluent) and, under some conditions, it is equivalent to the original system (roughly speaking, the derivations to constructor terms are the same).

A "more specific" transformation was originally introduced by Marriott et al. [24] in the context of logic programming. In this context, a *more specific version* of a logic program is a version of this program where each clause is further instantiated or removed while preserving the successful derivations of the original program. According to [24], the transformation increases the number of finitely failed goals (i.e., some infinite derivations are transformed into finitely failed ones), detects failure more quickly, etc. In general, the information about the allowed variable bindings which is hidden in the original program may be made explicit in a more specific version, thus improving the static analysis of the program's properties.

In this paper, we adapt the notion of a more specific program to the context of deterministic conditional term rewriting systems. In principle, the transformation may achieve similar benefits as in logic programming by making some variable bindings explicit (as illustrated in the transformation of function inc^{-1} above). Adapting this notion to rewriting systems, however, is far from trivial:

- First, there is no clear notion of *successful* reduction, and aiming at preserving all possible reductions would give rise to a meaningless notion. Consider, e.g., the rewrite systems $\mathcal{R} = \{\mathsf{f}(x) \to \mathsf{g}(x),\ \mathsf{g}(\mathsf{a}) \to \mathsf{b}\}$ and $\mathcal{R}' = \{\mathsf{f}(\mathsf{a}) \to \mathsf{g}(\mathsf{a}),\ \mathsf{g}(\mathsf{a}) \to \mathsf{b}\}$. Here, one would expect $\mathcal{R}'$ to be a correct more specific version of $\mathcal{R}$ (in the sense of $\to_{\mathcal{R}} = \to_{\mathcal{R}'}$). However, the reduction $\mathsf{f}(\mathsf{b}) \to_{\mathcal{R}} \mathsf{g}(\mathsf{b})$ is not possible in $\mathcal{R}'$.
- Second, different reduction *strategies* require different conditions in order to guarantee the correctness of the transformation.
- Finally, in contrast to [24], we often require computing a set of instances of a rewrite rule (rather than a single, more general instance) in order to produce

non-overlapping left-hand sides. Consider the rewrite system $\mathcal{R} = \{\mathsf{f}(\mathsf{a}) \to \mathsf{a},\ \mathsf{f}(x) \to \mathsf{g}(x),\ \mathsf{g}(\mathsf{b}) \to \mathsf{b},\ \mathsf{g}(\mathsf{c}) \to \mathsf{c}\}$. If we aim at computing a single more specific version for the second rule, only the same rule is obtained. However, if a set of instances is allowed rather a single common instance, we can obtain the system $\mathcal{R}' = \{\mathsf{f}(\mathsf{a}) \to \mathsf{a},\ \mathsf{f}(\mathsf{b}) \to \mathsf{g}(\mathsf{b}),\ \mathsf{f}(\mathsf{c}) \to \mathsf{g}(\mathsf{c}),\ \mathsf{g}(\mathsf{b}) \to \mathsf{b},\ \mathsf{g}(\mathsf{c}) \to \mathsf{c}\}$, which is non-overlapping.

Apart from introducing the notions of *successful* reduction and *more specific* version (MSV), we provide a constructive algorithm for computing more specific versions, which is based on constructing finite (possibly incomplete) *narrowing* trees for the right-hand sides of the original rewrite rules. Here, narrowing [36, 19], an extension of term rewriting by replacing pattern matching with unification, is used to *guess* the allowed variable bindings for a given rewrite rule. We prove the correctness of the algorithm (i.e., that it actually outputs a more specific version of the input system). We have tested the usefulness of the MSV transformation to improve the inverse systems obtained by the program inversion method of [27, 26, 28], and our preliminary results are very encouraging.

This paper is organized as follows. In Section 2, we briefly review some notions and notations of term rewriting and narrowing. Section 3 introduces the notions of more specific version of a rewriting system. Then, we introduce an algorithm for computing more specific versions in Section 4 and prove its correctness. Finally, Section 5 concludes and points out some directions for future research. A preliminary version of this paper—with a more restricted notion of more specific version and where only unconditional rules are considered—appeared in [30].

2 Preliminaries

We assume familiarity with basic concepts of term rewriting and narrowing. We refer the reader to, e.g., [4], [32], and [16] for further details.

Terms and Substitutions. A *signature* $\mathcal{F}$ is a set of function symbols. Given a set of variables $\mathcal{V}$ with $\mathcal{F} \cap \mathcal{V} = \varnothing$, we denote the domain of *terms* by $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We assume that $\mathcal{F}$ always contains at least one constant $\mathsf{f}/0$. We use $\mathsf{f}, \mathsf{g}, \ldots$ to denote functions and $x, y, \ldots$ to denote variables. Positions are used to address the nodes of a term viewed as a tree. A *position* p in a term t is represented by a finite sequence of natural numbers, where ϵ denotes the root position. We let $t|_p$ denote the *subterm* of t at position p and $t[s]_p$ the result of *replacing the subterm* $t|_p$ by the term s. $\mathcal{V}ar(t)$ denotes the set of variables appearing in t. A term t is *ground* if $\mathcal{V}ar(t) = \varnothing$.

A *substitution* $\sigma : \mathcal{V} \mapsto \mathcal{T}(\mathcal{F}, \mathcal{V})$ is a mapping from variables to terms such that $\mathcal{D}\mathsf{om}(\sigma) = \{x \in \mathcal{V} \mid x \neq \sigma(x)\}$ is its domain. Substitutions are extended to morphisms from $\mathcal{T}(\mathcal{F}, \mathcal{V})$ to $\mathcal{T}(\mathcal{F}, \mathcal{V})$ in the natural way. We denote the application of a substitution σ to a term t by $t\sigma$ rather than $\sigma(t)$. The identity substitution is denoted by *id*. A *variable renaming* is a substitution that is a bijection on $\mathcal{V}$. A substitution σ is *more general* than a substitution θ, denoted by $\sigma \leqslant \theta$, if there is a substitution δ such that $\delta \circ \sigma = \theta$, where "$\circ$" denotes the composition of substitutions (i.e., $\sigma \circ \theta(x) = (x\theta)\sigma = x\theta\sigma$). The *restriction*

$\theta\restriction_V$ of a substitution θ to a set of variables V is defined as follows: $x\theta\restriction_V = x\theta$ if $x \in V$ and $x\theta\restriction_V = x$ otherwise. We say that $\theta = \sigma\ [V]$ if $\theta\restriction_V = \sigma\restriction_V$.

A term t_2 is an *instance* of a term t_1 (or, equivalently, t_1 is *more general* than t_2), in symbols $t_1 \leqslant t_2$, if there is a substitution σ with $t_2 = t_1\sigma$. Two terms t_1 and t_2 are *variants* (or equal up to variable renaming) if $t_1 = t_2\rho$ for some variable renaming ρ. A *unifier* of two terms t_1 and t_2 is a substitution σ with $t_1\sigma = t_2\sigma$; furthermore, σ is the *most general unifier* of t_1 and t_2, denoted by $\mathsf{mgu}(t_1, t_2)$ if, for every other unifier θ of t_1 and t_2, we have that $\sigma \leqslant \theta$.

TRSs and Rewriting. A set of rewrite rules $l \to r$ such that l is a nonvariable term and r is a term whose variables appear in l is called a *term rewriting system* (TRS for short); terms l and r are called the left-hand side and the right-hand side of the rule, respectively. We restrict ourselves to finite signatures and TRSs. Given a TRS $\mathcal{R}$ over a signature $\mathcal{F}$, the *defined* symbols $\mathcal{D}_\mathcal{R}$ are the root symbols of the left-hand sides of the rules and the *constructors* are $\mathcal{C}_\mathcal{R} = \mathcal{F} \setminus \mathcal{D}_\mathcal{R}$. *Constructor terms* of $\mathcal{R}$ are terms over $\mathcal{C}_\mathcal{R}$ and $\mathcal{V}$. We sometimes omit $\mathcal{R}$ from $\mathcal{D}_\mathcal{R}$ and $\mathcal{C}_\mathcal{R}$ if it is clear from the context. A substitution σ is a *constructor substitution* (of $\mathcal{R}$) if $x\sigma \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V})$ for all variables x.

For a TRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as the smallest binary relation satisfying the following: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exist a position p in s, a rewrite rule $l \to r \in \mathcal{R}$ and a substitution σ with $s|_p = l\sigma$ and $t = s[r\sigma]_p$; the rewrite step is often denoted by $s \to_{p,l\to r} t$ to make explicit the position and rule used in this step. The instantiated left-hand side $l\sigma$ is called a *redex*.

A term t is called *irreducible* or in *normal form* w.r.t. a TRS $\mathcal{R}$ if there is no term s with $t \to_\mathcal{R} s$. A substitution is called *normalized* w.r.t. $\mathcal{R}$ if every variable in the domain is replaced by a normal form w.r.t. $\mathcal{R}$. We sometimes omit "w.r.t. $\mathcal{R}$" if it is clear from the context. We denote the set of normal forms by $NF_\mathcal{R}$. A *derivation* is a (possibly empty) sequence of rewrite steps. Given a binary relation $\to$, we denote by $\to^*$ its reflexive and transitive closure. Thus $t \to^*_\mathcal{R} s$ means that t can be reduced to s in $\mathcal{R}$ in zero or more steps; we also use $t \to^n_\mathcal{R} s$ to denote that t can be reduced to s in exactly n rewrite steps.

A conditional TRS (CTRS) is a set of rewrite rules $l \to r \Leftarrow C$, where C is a set of equations. In particular, we consider only *oriented* equations of the form $s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$. For a CTRS $\mathcal{R}$, we define the associated rewrite relation $\to_\mathcal{R}$ as follows: given terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have $s \to_\mathcal{R} t$ iff there exist a position p in s, a rewrite rule $l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n \in \mathcal{R}$ and a substitution σ such that $s|_p = l\sigma$, $s_i\sigma \to^*_\mathcal{R} t_i\sigma$ for all $i = 1, \ldots, n$, and $t = s[r\sigma]_p$.

Narrowing. The *narrowing* principle [36] mainly extends term rewriting by replacing pattern matching with unification, so that terms containing logic (i.e., *free*) variables can also be reduced by non-deterministically instantiating these variables. Conceptually, this is not significantly different from ordinary rewriting when TRSs contain *extra variables* (i.e., variables that appear in the right-hand side of a rule but not in its left-hand side), as noted in [3]. Formally, given a

TRS $\mathcal{R}$ and two terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, we have that $s \leadsto_{\mathcal{R}} t$ is a *narrowing step* iff there exist[3]

- a nonvariable position p of s,
- a variant $l \to r$ of a rule in $\mathcal{R}$,
- a substitution $\sigma = \mathsf{mgu}(s|_p, l)$ which is a most general unifier of $s|_p$ and l,

and $t = (s[r]_p)\sigma$. We often write $s \leadsto_{p,l\to r,\theta} t$ (or simply $s \leadsto_\theta t$) to make explicit the position, rule, and substitution of the narrowing step, where $\theta = \sigma\restriction_{\mathcal{V}\mathsf{ar}(s)}$ (i.e., we label the narrowing step only with the bindings for the narrowed term). A *narrowing derivation* $t_0 \leadsto^*_\sigma t_n$ denotes a sequence of narrowing steps $t_0 \leadsto_{\sigma_1} \ldots \leadsto_{\sigma_n} t_n$ with $\sigma = \sigma_n \circ \cdots \circ \sigma_1$ (if $n = 0$ then $\sigma = id$). Given a narrowing derivation $s \leadsto^*_\sigma t$ with t a constructor term, we say that σ is a *computed answer* for s.

Example 1. Consider the TRS

$$\mathcal{R} = \left\{ \begin{array}{ll} \mathsf{add}(\mathsf{0}, y) \to y & (R_1) \\ \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y)) & (R_2) \end{array} \right\}$$

defining the addition $\mathsf{add}/2$ on natural numbers built from $\mathsf{0}/0$ and $\mathsf{s}/1$. Given the term $\mathsf{add}(x, \mathsf{s}(\mathsf{0}))$, we have infinitely many narrowing derivations starting from $\mathsf{add}(x, \mathsf{s}(\mathsf{0}))$, e.g.,

$$\begin{array}{ll} \mathsf{add}(x, \mathsf{s}(\mathsf{0})) \leadsto_{\epsilon,R_1,\{x\mapsto \mathsf{0}\}} & \mathsf{s}(\mathsf{0}) \\ \mathsf{add}(x, \mathsf{s}(\mathsf{0})) \leadsto_{\epsilon,R_2,\{x\mapsto \mathsf{s}(y_1)\}} & \mathsf{s}(\mathsf{add}(y_1, \mathsf{s}(\mathsf{0}))) \leadsto_{1,R_1,\{y_1\mapsto \mathsf{0}\}} \mathsf{s}(\mathsf{s}(\mathsf{0})) \\ \ldots & \end{array}$$

with computed answers $\{x \mapsto \mathsf{0}\}$, $\{x \mapsto \mathsf{s}(\mathsf{0})\}$, etc.

Narrowing is naturally extended to deal with equations and CTRSs (see Section 4).

3 More Specific Conditional Rewrite Systems

In this section, we introduce the notion of a *more specific* conditional rewrite system. Intuitively speaking, we produce a more specific CTRS $\mathcal{R}'$ from a CTRS $\mathcal{R}$ by replacing a conditional rewrite rule $(l \to r \Leftarrow C) \in \mathcal{R}$ with a *finite* number of *instances* of this rule, i.e., $\mathcal{R}' = (\mathcal{R} \setminus \{l \to r \Leftarrow C\}) \cup \{(l \to r \Leftarrow C)\sigma_1, \ldots, (l \to r \Leftarrow C)\sigma_n\}$, such that $\mathcal{R}'$ is *semantically* equivalent to $\mathcal{R}$ under some conditions.

The key idea is that more specific versions should still allow the same reductions of the original system. However, as mentioned in Section 1, if we aimed at preserving *all* possible rewrite reductions, the resulting notion would be useless since it would never happen in practice. Therefore, to have a practically applicable technique, we only aim at preserving what we call *successful* reductions. In the following, given a CTRS $\mathcal{R}$, we denote by $\xrightarrow{s}_{\mathcal{R}}$ a generic conditional rewrite relation based on some strategy s (e.g., innermost conditional reduction $\xrightarrow{i}_{\mathcal{R}}$).

[3] We consider the so called *most general* narrowing, i.e., the mgu of the selected subterm and the left-hand side of a rule—rather than an arbitrary unifier—is computed at each narrowing step.

Definition 1 (successful reduction w.r.t. $\overset{s}{\to}$). *Let $\mathcal{R}$ be a CTRS and let $\overset{s}{\to}_{\mathcal{R}}$ be a conditional rewrite relation. A rewrite reduction $t \overset{s}{\to}^*_{\mathcal{R}} u$ where t is a term and $u \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ is a constructor term, is called a* successful *reduction w.r.t. $\overset{s}{\to}_{\mathcal{R}}$.*

Let us now introduce our notion of a *more specific version* of a rewrite rule:

Definition 2 (more specific version of a rule). *Let $\mathcal{R}$ be a CTRS and $\overset{s}{\to}_{\mathcal{R}}$ be a conditional rewrite relation. Let $l \to r \Leftarrow C \in \mathcal{R}$ be a rewrite rule. We say that the finite set of rewrite rules $\mathcal{R}_{msv} = \{l_1 \to r_1 \Leftarrow C_1, \ldots, l_n \to r_n \Leftarrow C_n\}$ is a* more specific version of $l \to r \Leftarrow C$ in $\mathcal{R}$ w.r.t. $\overset{s}{\to}_{\mathcal{R}}$ *if*

- *there are substitutions $\sigma_1, \ldots, \sigma_n$ such that $(l \to r \Leftarrow C)\sigma_i = l_i \to r_i \Leftarrow C_i$ for $i = 1, \ldots, n$, and*
- *for all terms t, u, we have that $t \overset{s}{\to}^*_{\mathcal{R}} u$ is successful in $\mathcal{R}$ iff $t \overset{s}{\to}^*_{\mathcal{R}'} u$ is successful in $\mathcal{R}'$, with $\mathcal{R}' = (\mathcal{R} \setminus \{l \to r \Leftarrow C\}) \cup \mathcal{R}_{msv}$; moreover, we require $t \overset{s}{\to}^*_{\mathcal{R}} u$ and $t \overset{s}{\to}^*_{\mathcal{R}'} u$ to apply the same rules to the same positions and in the same order, except for the rule $l \to r \Leftarrow C$ in $\mathcal{R}$ that is replaced with some rule $l_i \to r_i \Leftarrow C_i$, $i \in \{1, \ldots, n\}$, in $\mathcal{R}'$.*[4]

Note that a rewrite rule is always a more specific version of itself, therefore the existence of a more specific version of a given rule is always ensured. In general, however, the more specific version is not unique, and thus, there can be several strictly more specific versions of a rewrite rule.

The notion of a more specific version of a rule is extended to CTRSs in a stepwise manner: given a CTRS $\mathcal{R}$, we first replace a rule of $\mathcal{R}$ by its more specific version thus producing $\mathcal{R}'$, then we replace another rule of $\mathcal{R}'$ by its more specific version, and so forth. We denote each of these steps by $\mathcal{R} \mapsto_{\mathsf{more}} \mathcal{R}'$. We say that $\mathcal{R}'$ is a *more specific version* of $\mathcal{R}$ if there is a sequence of (zero or more) $\mapsto_{\mathsf{more}}$ steps leading from $\mathcal{R}$ to $\mathcal{R}'$. Note that, given a CTRS $\mathcal{R}$ and one of its more specific versions $\mathcal{R}'$, we have that $\to_{\mathcal{R}'} \subseteq \to_{\mathcal{R}}$ (i.e., $NF_{\mathcal{R}} \subseteq NF_{\mathcal{R}'}$) and $\mathcal{D}_{\mathcal{R}} = \mathcal{D}_{\mathcal{R}'}$ (i.e., $\mathcal{C}_{\mathcal{R}} = \mathcal{C}_{\mathcal{R}'}$).

Example 2. Consider the following CTRS $\mathcal{R}$ (a fragment of a system obtained automatically by the function inversion technique of [28]):

$$\begin{aligned}
\mathsf{inv}([\mathsf{left}|x_2]) &\to (t, \mathsf{n}(x, y)) \Leftarrow \mathsf{inv}(x_2) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y) && (R_1)\\
\mathsf{inv}([\mathsf{str}(x)|t]) &\to (t, \mathsf{sym}(x)) && (R_2)\\
\mathsf{inv}([\mathsf{left}, \mathsf{right}|t]) &\to (t, \mathsf{bottom}) && (R_3)
\end{aligned}$$

where the definition of function inv' is not relevant for this example and we omit the tuple symbol (e.g., tp_2) from the tuple of two terms—we write (t_1, t_2) instead of $\mathsf{tp}_2(t_1, t_2)$. Here, we replace the first rule of $\mathcal{R}$ by the following two instances $\mathcal{R}_{msv}$:

$$\begin{aligned}
\mathsf{inv}([\mathsf{left}, \mathsf{left}|x_3]) &\to (t, \mathsf{n}(x, y)) \Leftarrow \mathsf{inv}([\mathsf{left}|x_2]) \twoheadrightarrow (x_1, x), \mathsf{inv}'(x_1) \twoheadrightarrow (t, y)\\
\mathsf{inv}([\mathsf{left}, \mathsf{str}(x_3)|x_4]) &\to (t, \mathsf{n}(x, y)) \Leftarrow \mathsf{inv}([\mathsf{str}(x_3)|x_4]) \twoheadrightarrow (x_1, x), \mathsf{inv}'(x_1) \twoheadrightarrow (t, y)
\end{aligned}$$

[4] This is required to prevent situations like the following one. Consider $\mathcal{R} = \{\mathsf{f}(\mathsf{0}) \to \mathsf{g}(\mathsf{0}),\ \mathsf{f}(x) \to \mathsf{g}(x),\ \mathsf{g}(\mathsf{0}) \to \mathsf{0}\}$. Here, any instance of rule $\mathsf{f}(x) \to \mathsf{g}(x)$ would be a more specific version if the last condition were not required (since the rule $\mathsf{f}(\mathsf{0}) \to \mathsf{g}(\mathsf{0})$ already belongs to $\mathcal{R}$).

using the substitutions $\sigma_1 = \{x_2 \mapsto [\mathsf{left}|x_3]\}$ and $\sigma_2 = \{x_2 \mapsto [\mathsf{str}(x_3)|x_4]\}$.

Observe that function inv in $(\mathcal{R}\backslash\{R_1\}) \cup \mathcal{R}_{msv}$ is now non-overlapping. Note that producing only a single instance would return an overlapping rule again since the only common generalization of σ_1 and σ_2 is $\{x_2 \mapsto [x_3|x_4]\}$ so that the more specific version would be

$$\mathsf{inv}([\mathsf{left}, x_3|x_4]) \to (t, \mathsf{n}(x, y)) \Leftarrow \mathsf{inv}([x_3|x_4]) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y)$$

and function inv would still be overlapping.

Now, we show a basic property of more specific versions of a CTRS. In the following, we say that a CTRS $\mathcal{R}$ *has the unique constructor normal form property w.r.t.* $\xrightarrow{s}_{\mathcal{R}}$ if, for all successful reductions $(C, t \twoheadrightarrow x) \xrightarrow{s}{}^{*} (u \twoheadrightarrow x)$ and $(C, t \twoheadrightarrow x) \xrightarrow{s}{}^{*} (u' \twoheadrightarrow x)$, with $u, u' \in \mathcal{T}(\mathcal{C}, \mathcal{V})$, we have $u = u'$.

Theorem 1. *Let $\mathcal{R}$ be a CTRS and let $\mathcal{R}'$ be a more specific version of $\mathcal{R}$ w.r.t. $\xrightarrow{s}_{\mathcal{R}}$. Then, $\mathcal{R}$ has the unique constructor normal form property w.r.t. $\xrightarrow{s}_{\mathcal{R}}$ iff so does $\mathcal{R}'$.*

Proof. The claim follows straightforwardly since derivations producing a constructor normal form are successful derivations, and they are preserved by the MSV transformation by definition. □

4 Computing More Specific Versions

In this section, we tackle the definition of a constructive method for computing more specific versions of a CTRS. For this purpose, we consider the following assumptions:

- We restrict the class of CTRSs to *deterministic* CTRSs (DCTRSs [5, 13, 23]). Furthermore, we require them to be constructor systems (see below).
- We only consider constructor-based reductions (a particular case of innermost conditional reduction), i.e., reductions where the computed matching substitutions are constructor.
- We use a form of innermost conditional narrowing to approximate the potential successful reductions and, thus, compute its more specific version.

DCTRSs are *3-CTRSs* [5, 13, 23] (see also [32]) (i.e., CTRSs where extra variables are allowed as long as $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l) \cup \mathcal{V}ar(C)$ for all rules $l \to r \Leftarrow C$), where the conditional rules have the form

$$l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$$

such that

- $\mathcal{V}\mathsf{ar}(s_i) \subseteq \mathcal{V}\mathsf{ar}(l) \cup \mathcal{V}\mathsf{ar}(t_1) \cup \cdots \cup \mathcal{V}\mathsf{ar}(t_{i-1})$, for all $i = 1, \ldots, n$;
- $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l) \cup \mathcal{V}\mathsf{ar}(t_1) \cup \ldots \cup \mathcal{V}\mathsf{ar}(t_n)$.

Moreover, we assume that the DCTRSs are constructor systems with $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$.

In DCTRSs, extra variables in the conditions are not problematic since no redex contains extra variables when it is selected. Actually, as noted by [23], these systems are basically equivalent to functional programs since every rule

$$l \to r \Leftarrow s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$$

can be seen in a functional language as

$$\begin{array}{l} l \;=\; \textbf{let } t_1 = s_1 \textbf{ in} \\ \qquad\quad \textbf{let } t_2 = s_2 \textbf{ in} \\ \qquad\qquad \ldots \\ \qquad\qquad\quad \textbf{let } t_n = s_n \textbf{ in } r \end{array}$$

Here, DCTRSs allow us to represent functional *local definitions* using oriented conditions.

Under these conditions, innermost reduction extends quite naturally to the conditional case. In particular, we follow Bockmayr and Werner's *conditional rewriting without evaluation of the premise* [6], adapted to our setting as follows:

Definition 3 (constructor reduction). *Let $\mathcal{R}$ be a DCTRS. Constructor-based conditional reduction is defined as the smallest relation satisfying the following transition rules:*

$$\textit{(reduction)} \quad \frac{p = inn(s_1) \wedge l \to r \Leftarrow C \in \mathcal{R} \wedge s_1|_p = l\sigma \wedge \sigma \text{ is constructor}}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \;\overset{c}{\to}\; (C\sigma, s_1[r\sigma]_p \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n)}$$

$$\textit{(matching)} \quad \frac{n > 1 \wedge s_1 \in \mathcal{T}(\mathcal{C}, \mathcal{V}) \wedge s_1 = t_1\sigma}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \;\overset{c}{\to}\; (s_2 \twoheadrightarrow t_2, \ldots, s_n \twoheadrightarrow t_n)\sigma}$$

where $inn(s)$ selects the position of an innermost subterm matchable with the left-hand side of a rule (i.e., a term $l'\sigma'$ of the form $f(c_1, \ldots, c_n)$ with $f \in \mathcal{D}$ and $c_1, \ldots, c_n \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ for some $l' \to r' \Leftarrow C' \in \mathcal{R}$ and some σ'), e.g., the leftmost one.

Intuitively speaking, in order to reduce a sequence of equations $s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n$, we consider two possibilities:

- If the first oriented equation has some innermost subterm that matches the left-hand side of a rewrite rule, we perform a reduction step. Note that, in contrast to ordinary conditional rewriting, the conditions are not verified but just added to the set of equations (as in Bockmayr and Werner's reduction without evaluation of the premise).
- If the left-hand side of the first oriented equation is a constructor term, then we match both sides (note that the right-hand side is a constructor term by definition) and remove this equation from the sequence. In the original definition of [6], this matching substitution is computed when applying a given rule in order to verify the conditions. Our definition simply makes computing the substitution more operational by postponing it to the point when its computation is required (and, thus, it is straightforward to compute).

Note that this rule requires having more than one equation, since the initial equation should not be removed.

In order to reduce a ground term s, we consider an initial oriented equation $s \twoheadrightarrow x$, where x is a fresh variable not occurring in s, and reduce it as much as possible using the reduction and matching rules. If we reach an equation of the form $t \twoheadrightarrow x$, where t is a constructor term, we say that s reduces to t; actually, [6, Theorem 2] proves the equivalence between ordinary conditional rewriting and conditional rewriting without evaluation of the premise.

Example 3. Consider again the system $\mathcal{R}$ from Example 2 and the initial term $\mathsf{inv}([\mathsf{left}, \mathsf{str}(\mathsf{a})])$. We have, for instance, the following (incomplete) constructor-based reduction:

$$\begin{aligned}(\mathsf{inv}([\mathsf{left}, \mathsf{str}(\mathsf{a})]) &\twoheadrightarrow w)\\ &\overset{c}{\to} (\mathsf{inv}([\mathsf{str}(\mathsf{a})]) \twoheadrightarrow (x_1, x), \mathsf{inv}'(x_1) \twoheadrightarrow (t, y), (t, \mathsf{n}(x, y)) \twoheadrightarrow w)\\ &\overset{c}{\to} ((\mathsf{nil}, \mathsf{sym}(\mathsf{a})) \twoheadrightarrow (x_1, x), \mathsf{inv}'(x_1) \twoheadrightarrow (t, y), (t, \mathsf{n}(x, y)) \twoheadrightarrow w)\\ &\overset{c}{\to} (\mathsf{inv}'(\mathsf{nil}) \twoheadrightarrow (t, y), (t, \mathsf{n}(\mathsf{sym}(\mathsf{a}), y)) \twoheadrightarrow w)\end{aligned}$$

In the following, given a rule $l \to r \Leftarrow C$, we introduce the use of conditional narrowing to automatically compute an approximation of the successful constructor-based reductions.

Our definition of constructor-based conditional narrowing (a special case of innermost conditional narrowing [9, 12, 18]), denoted by $\overset{c}{\rightsquigarrow}$, is defined as follows:

Definition 4 (constructor-based conditional narrowing). *Let $\mathcal{R}$ be a DC-TRS. Constructor-based conditional narrowing is defined as the smallest relation satisfying the following transition rules:*

$$\textit{(narrowing)} \quad \frac{p = \mathit{inn}(s_1) \ \wedge \ l \to r \Leftarrow C \in \mathcal{R} \ \wedge \ \sigma = \mathsf{mgu}(s_1|_p, l)}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \ \overset{c}{\rightsquigarrow}_\sigma \ (C, s_1[r]_p \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n)\sigma}$$

$$\textit{(unification)} \quad \frac{n > 1 \ \wedge \ s_1 \in \mathcal{T}(\mathcal{C}, \mathcal{V}) \ \wedge \ \sigma = \mathsf{mgu}(s_1, t_1)}{(s_1 \twoheadrightarrow t_1, \ldots, s_n \twoheadrightarrow t_n) \ \overset{c}{\rightsquigarrow}_\sigma \ (s_2 \twoheadrightarrow t_2, \ldots, s_n \twoheadrightarrow t_n)\sigma}$$

where $\mathit{inn}(s)$ selects the position of an innermost subterm whose proper subterms are constructor terms, and which is unifiable with the left-hand side of a rule, e.g., the leftmost one.

As can be seen, our definition of constructor-based conditional narrowing for DC-TRSs mimics the definition of constructor-based reduction but replaces matching with unification in both transition rules.

Note, however, that the first rule is often non-deterministic since a given innermost subterm can unify with the left-hand sides of several rewrite rules.

We adapt the notion of successful derivations to rewriting and narrowing derivations of equations.

Definition 5. *Let $\mathcal{R}$ be a CTRS and let $\overset{s}{\to}_\mathcal{R}$ be a conditional rewrite relation. A rewrite reduction $(C, t \twoheadrightarrow x) \overset{s}{\to}^*_\mathcal{R} (u \twoheadrightarrow x)$ where t is a term, x is a fresh variable, C is a (possibly empty) sequence of (oriented) equations, and $u \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ is a constructor term, is called a* successful *reduction w.r.t. $\overset{s}{\to}_\mathcal{R}$.*

Note that $t \xrightarrow{s}{}^*_{\mathcal{R}} u$ is successful w.r.t. $\xrightarrow{s}_{\mathcal{R}}$ iff so is $(t \twoheadrightarrow x) \xrightarrow{s}{}^*_{\mathcal{R}} (u \twoheadrightarrow x)$, where x is a fresh variable.

Definition 6. *Consider a set of equations C, a term s and a fresh variable x. We say that a constructor-based conditional narrowing derivation of the form $(C, s \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\sigma (t \twoheadrightarrow x)$ is* successful *if $t \in \mathcal{T}(\mathcal{C}, \mathcal{V})$ is a constructor term.*

We say that a derivation of the form $(C, s \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\sigma (C', s' \twoheadrightarrow x)$ is a failing *derivation if no more narrowing steps can be applied and at least one of the following following holds:*

- *C' is not the empty sequence, or*
- *s' is not a constructor term.*

Otherwise, we say that it is an incomplete *derivation.*

Because of the non-determinism of rule narrowing, the computation of all narrowing derivations starting from a given term is usually represented by means of a narrowing *tree*:

Definition 7 (constructor-based conditional narrowing tree). *Let $\mathcal{R}$ be a DCTRS and C be a sequence of equations. A (possibly incomplete)* constructor-based conditional narrowing tree *for C in $\mathcal{R}$ is a (possibly infinite) directed rooted node- and edge-labeled graph τ built as follows:*

- *the root node of τ is labeled with C;*
- *every node C_1 is either a leaf (a node with no output edge) or it is unfolded as follows: there is an output edge from node C_1 to node C_2 labeled with σ for all constructor-based conditional narrowing steps $C_1 \overset{c}{\leadsto}_\sigma C_2$ for the selected innermost narrowable term;*
- *the root node is not a leaf—at least the root node should be unfolded.*

By abuse of notation, we will denote a finite constructor-based conditional narrowing tree τ (and its subtrees) for C as a finite set with the constructor-based conditional narrowing derivations starting from C in this tree, i.e., $C \overset{c}{\leadsto}{}^*_\theta C' \in \tau$ if there is a root-to-leaf path from C to C' in τ labeled with substitutions $\theta_1, \theta_2, \ldots, \theta_n$ such that $C \overset{c}{\leadsto}_{\theta_1} \ldots \overset{c}{\leadsto}_{\theta_n} C'$ is a constructor-based conditional narrowing derivation and $\theta = \theta_n \circ \cdots \circ \theta_1$.

Example 4. Consider again the system $\mathcal{R}$ from Example 2 and the initial sequence of equations $C = (\mathsf{inv}(x_2) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w)$, where w is a fresh variable (the reason to consider this initial sequence of equations will be clear in Definition 9 below). The depth-1 (i.e., derivations are stopped after one narrowing step) constructor-based conditional narrowing tree τ for C contains the following derivations:

$$
\begin{array}{ll}
C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{left} | x_2']\}} & (\mathsf{inv}(x_2') \twoheadrightarrow (x_1', x'),\ \mathsf{inv}'(x_1') \twoheadrightarrow (t', y'), \\
 & \quad (t', \mathsf{n}(x', y')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w) \\
C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{str}(x') | t']\}} & ((t', \mathsf{sym}(x')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w) \\
C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{left},\mathsf{right} | t']\}} & ((t', \mathsf{bottom}) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w)
\end{array}
$$

Note that a constructor-based conditional narrowing tree can be *incomplete* in the sense that we do not require all unfoldable nodes to be unfolded (i.e., some finite derivations in the tree may be incomplete). Nevertheless, if a node is selected to be unfolded, it should be unfolded in all possible ways using constructor-based conditional narrowing (i.e., one cannot *partially* unfold a node by ignoring some matching rules, which would give rise to incorrect results). In order to keep the tree finite, one can introduce a heuristics that determines when the construction of the tree should terminate. We consider the definition of a particular strategy for ensuring termination out of the scope of this paper; nevertheless, one could use a simple depth-k strategy (as in the previous example) or some more elaborated strategies based on well-founded or well-quasi orderings [10] (as in narrowing-driven partial evaluation [1]).

Let us now recall the notion of *least (general) generalization* [34] (also called *anti-unification* [33]), which will be required to compute a common generalization of all instances of a term by a set of narrowing derivations.

Definition 8 (least general generalization [34], *lgg*). *Given two terms s and t, we say that w is a generalization of s and t if $w \leqslant s$ and $w \leqslant t$; moreover, it is called the least general generalization of s and t, denoted by $lgg(s,t)$, if $w' \leqslant w$ for all other generalizations w' of s and t. This notion is extended to sets of terms in the natural way: $lgg(\{t_1,\ldots,t_n\}) = lgg(t_1, lgg(t_2,\ldots lgg(t_{n-1},t_n)\ldots))$ (with $lgg(\{t_1\}) = t_1$ when $n = 1$).*

An algorithm for computing the least general generalization can be found, e.g., in [11]. For instance, $lgg(\mathsf{f}(\mathsf{a},\mathsf{g}(\mathsf{a})),\mathsf{f}(\mathsf{b},\mathsf{g}(\mathsf{b}))) = \mathsf{f}(x_3,\mathsf{g}(x_3))$.

We now introduce a constructive algorithm to produce a more specific version of a rule:

Definition 9 (MSV **algorithm for DCTRSs).** *Let $\mathcal{R}$ be a DCTRS and let $l \to r \Leftarrow C \in \mathcal{R}$ be a conditional rewrite rule such that not all terms in r and C are constructor terms. Let τ be a finite (possibly incomplete) constructor-based conditional narrowing tree for $(C, r \twoheadrightarrow x)$ in $\mathcal{R}$, where x is a fresh variable, and $\tau' \subseteq \tau$ be the tree obtained from τ by excluding the failing derivations (if any). We compute a more specific version of $l \to r \Leftarrow C$ in $\mathcal{R}$, denoted by* $\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau)$ *, as follows:*

- *If $\tau' = \varnothing$, then* $\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau) = \bot$*, where $\bot$ is used to denote that the rule is useless (i.e., no successful constructor-based reduction can use it).*
- *If $\tau' \neq \varnothing$, then we let $\tau' = \tau_1 \uplus \ldots \uplus \tau_n$ be a partition of the set τ such that $\tau_i \neq \varnothing$ for all $i = 1,\ldots,n$. Then,*[5]

$$\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau) = \{(l \to r \Leftarrow C)\sigma_1, \ldots, (l \to r \Leftarrow C)\sigma_n\}$$

where $(l \to r \Leftarrow C)\sigma_i = lgg(\{(l \to r \Leftarrow C)\theta \mid (C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\theta (C', r' \twoheadrightarrow x) \in \tau_i\})$ with $\mathcal{D}\mathsf{om}(\sigma_i) \subseteq \mathcal{V}\mathsf{ar}(C) \cup \mathcal{V}\mathsf{ar}(r)$, $i = 1,\ldots,n$.[6]

[5] The *lgg* operator is trivially extended to equations by considering them as terms, e.g., the sequence $s_1 \twoheadrightarrow t_1, s_2 \twoheadrightarrow t_2$ is considered as the term $\wedge(\twoheadrightarrow(s_1,t_1), \twoheadrightarrow(s_1,t_2))$, where $\twoheadrightarrow$ and $\wedge$ are binary function symbols.

[6] By definition of constructor-based conditional narrowing, it is clear that $\sigma_1,\ldots,\sigma_n$ are constructor substitutions.

Regarding the partitioning of the derivations in the constructor-based conditional narrowing tree τ', i.e., computing $\tau_1, \ldots, \tau_n$ such that $\tau' = \tau_1 \uplus \ldots \uplus \tau_n$, we first apply a simple pre-processing to avoid trivial overlaps between the generated rules: we remove from τ' those derivations $(C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\theta (C', r' \twoheadrightarrow x)$ such that there exists another derivation $(C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_{\theta'} (C'', r'' \twoheadrightarrow x)$ with $\theta' \leqslant \theta$. In this way, we avoid the risk of having such derivations in different subtrees, τ_i and τ_j, thus producing overlapping rules. Once these derivations have been removed, we could proceed as follows:

- No partition ($n = 1$). This is what is done in the original transformation for logic programs [24] and gives good results for most examples (i.e., produces a non-overlapping system).
- Consider a maximal partitioning, i.e., each τ_i just contains a single derivation. This strategy might produce poor results when the computed substitutions overlap, since overlapping instances would then be produced (even if the considered function was originally non-overlapping).
- Use a heuristics that tries to produce a non-overlapping system whenever possible. Basically, it would proceed as follows. Assume we want to apply the MSV transformation to a function f. Let k be a natural number greater than the maximum depth of the left-hand sides of the rules defining f. Then, we partition the tree τ' as $\tau_1, \ldots, \tau_n$ so that it satisfies the following condition (while keeping n as small as possible):
  - for each $(C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_\theta C_1$ and $(C, r \twoheadrightarrow x) \overset{c}{\leadsto}{}^*_{\theta'} C_2$ in τ_i, we have that $top_k(l\theta) = top_k(l\theta')$, where $l \to r \Leftarrow C$ is the considered rule.

  Here, given a fresh constant $\top$, top_k is defined as follows: $top_0(t) = \top$, $top_k(x) = x$ for $k > 0$, $top_k(f(t_1, \ldots, t_m)) = f(top_{k-1}(t_1), \ldots, top_{k-1}(t_m))$ for $k > 0$, i.e., $top_k(t)$ returns the topmost symbols of t up to depth k and replaces the remaining subterms by the fresh constant $\top$.

Example 5. Let us apply the MSV algorithm to the first rule of the system $\mathcal{R}$ introduced in Example 2. Here, we consider the depth-1 constructor-based conditional narrowing tree τ with the derivations shown in Example 4. We first remove derivations labeled with less general substitutions, so that from the first and third derivations of Example 4, only the following one remains:

$$C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{left}|x_2']\}} (\mathsf{inv}(x_2') \twoheadrightarrow (x_1', x'),\ \mathsf{inv}'(x_1') \twoheadrightarrow (t', y'),\ (t', \mathsf{n}(x', y')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w)$$
$$C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{str}(x')|t']\}} (t', \mathsf{sym}(x')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w$$

Now, either by considering a maximal partitioning or the one based on function top_k, we compute the following partitions:

$$\tau_1 = \{C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{left}|x_2']\}} (\mathsf{inv}(x_2') \twoheadrightarrow (x_1', x'),\ \mathsf{inv}'(x_1') \twoheadrightarrow (t', y'),\ (t', \mathsf{n}(x', y')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w)\}$$
$$\tau_2 = \{C \overset{c}{\leadsto}_{\{x_2 \mapsto [\mathsf{str}(x')|t']\}} ((t', \mathsf{sym}(x')) \twoheadrightarrow (x_1, x),\ \mathsf{inv}'(x_1) \twoheadrightarrow (t, y),\ (t, \mathsf{n}(x, y)) \twoheadrightarrow w)\}$$

so that the computed more specific version, $\mathcal{R}_{msv}$, contains the two rules already shown in Example 2.

Now, we consider the correctness of the MSV transformation.

Theorem 2. *Let $\mathcal{R}$ be a DCTRS, $l \to r \Leftarrow C \in \mathcal{R}$ be a rewrite rule such that not all terms in r and C are constructor terms. Let τ be a finite (possibly incomplete) constructor-based conditional narrowing tree for $(C, r \twoheadrightarrow x)$ in $\mathcal{R}$. Then,*

- *If* $\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau) = \mathcal{R}_{msv}$, *then* $\mathcal{R}_{msv}$ *is a more specific version of* $l \to r \Leftarrow C$ *in* $\mathcal{R}$ *w.r.t.* $\overset{c}{\to}_{\mathcal{R}}$.
- *If* $\mathrm{MSV}(\mathcal{R}, l \to r \Leftarrow C, \tau) = \bot$, *then* $l \to r \Leftarrow C$ *is not used in any successful reduction in* $\mathcal{R}$ *w.r.t.* $\overset{c}{\to}_{\mathcal{R}}$.

The proof can be found in the full version of this paper [31].

5 Conclusion and Future Work

We have introduced the notion of a *more specific* version of a rewrite rule in the context of conditional rewrite systems with some restrictions (i.e., typical functional programs). The transformation is useful to produce non-overlapping systems from overlapping ones while preserving the so called successful reductions. We have introduced an automatic algorithm for computing more specific versions and have proved its correctness.

We have undertaken the extension of the implemented program inverter of [28] with a post-process based on the MSV transformation. We have tested the resulting transformation with the 15 program inversion benchmarks of [20].[7] In nine of these benchmarks (out of fifteen) an overlapping system was obtained by inversion while the remaining six are non-overlapping. By applying the MSV transformation to all overlapping rules—except for a predefined operator `du`—using a depth-3 constructor-based conditional narrowing tree in all examples, we succeeded in improving the nine overlapping systems, while the other six systems were left unchanged: nine overlapping systems are transformed into non-overlapping ones, and the remaining six are transformed into themselves. These promising results point out the usefulness of the approach to improve the quality of inverted systems.

As for future work, we plan to explore the use of the MSV transformation to improve the accuracy of termination analyses.

References

1. E. Albert and G. Vidal. The narrowing-driven approach to functional logic program specialization. *New Generation Comput.*, 20(1):3–26, 2002.

[7] Unfortunately, the site shown in [20] is not accessible now. Some of the benchmarks can be found in [14, 15, 21]. All the benchmarks are reviewed in [25] and also available from the following URL: `http://www.trs.cm.is.nagoya-u.ac.jp/repius/`. The two benchmarks `pack` and `packbin` define non-tail-recursive functions which include some tail-recursive rules; we transform them into pure tail recursive ones using the method introduced in [29] (to avoid producing non-operationally terminating programs), that are also inverted using the approach of [28].

2. J. M. Almendros-Jiménez and G. Vidal. Automatic partial inversion of inductively sequential functions. In *Proc. of the 18th International Symposium on Implementation and Application of Functional Languages*, vol. 4449 of *LNCS*, pp. 253–270, Springer, 2006.
3. S. Antoy and M. Hanus. Overlapping rules and logic variables in functional logic programs. In *Proc. of the 22nd International Conference on Logic Programming*, vol. 4079 of *LNCS*, pp. 87–101, Springer, 2006.
4. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
5. H. Bertling and H. Ganzinger. Completion-time optimization of rewrite-time goal solving. In *Proc. of the 3rd International Conference of Rewriting Techniques and Applications*, vol. 355 of *LNCS*, pp. 45–58, Springer, 1989.
6. A. Bockmayr and A. Werner. LSE narrowing for decreasing conditional term rewrite systems. In *Proc. of the 4th International Workshop on Conditional and Typed Rewriting Systems*, vol. 968 of *Lecture Notes in Computer Science*, pp. 51–70, Springer, 1995.
7. A. Bondorf. Towards a self-applicable partial evaluator for term rewriting systems. In *Proc. of the International Workshop on Partial Evaluation and Mixed Computation*, pp. 27–50. North-Holland, Amsterdam, 1988.
8. A. Bondorf. A self-applicable partial evaluator for term rewriting systems. In *Proc. of International Conference on Theory and Practice of Software Development, Barcelona, Spain*, vol. 352 of *LNCS*, pp. 81–95, Springer, 1989.
9. P. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution. *Theor. Comput. Sci.*, 59:3–23, 1988.
10. N. Dershowitz. Termination of rewriting. *J. Symb. Comput.*, 3(1&2):69–115, 1987.
11. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science*, vol. B: Formal Models and Semantics, pp. 243–320, Elsevier, Amsterdam, 1990.
12. L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In *Proc. of the Symposium on Logic Programming*, pp. 172–185, IEEE Press, 1985.
13. H. Ganzinger. Order-sorted completion: The many-sorted way. *Theor. Comput. Sci.*, 89(1):3–32, 1991.
14. R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In *Proc. of the first Asian Symposium on Programming Languages and Systems*, vol. 2895 of *LNCS*, pp. 246–264, Springer, 2003.
15. R. Glück and M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundam. Inform.*, 66(4):367–395, 2005.
16. M. Hanus. The integration of functions into logic programming: From theory to practice. *J. Log. Program.*, 19&20:583–628, 1994.
17. P. G. Harrison. Function inversion. In *Proc. of the International Workshop on Partial Evaluation and Mixed Computation*, pp. 153–166, North-Holland, Amsterdam, 1988.
18. S. Hölldobler. *Foundations of Equational Logic Programming*, vol. 353 of *LNAI*, Springer, 1989.
19. J.-M. Hullot. Canonical forms and unification. In *Proc. of the 5th International Conference on Automated Deduction*, vol. 87 of *LNCS*, pp. 318–334, Springer, 1980.
20. M. Kawabe and Y. Futamura. Case studies with an automatic program inversion system. In *Proc. of the 21st Conference of Japan Society for Software Science and Technology*, number 6C-3, 5 pages, 2004.
21. M. Kawabe and R. Glück. The program inverter LRinv and its structure. In *Proc. of the 7th International Symposium on Practical Aspects of Declarative Languages*, vol. 3350 of *LNCS*, pp. 219–234, Springer, 2005.

22. H. Khoshnevisan and K. M. Sephton. InvX: An automatic function inverter. In *Proc. of the 3rd International Conference of Rewriting Techniques and Applications*, vol. 355 of *LNCS*, pp. 564–568, Springer, 1989.
23. M. Marchiori. On deterministic conditional rewriting. Technical Report MIT-LCS-TM-405, MIT Laboratory for Computer Science, 1997. Available from `http://www.w3.org/People/Massimo/papers/MIT-LCS-TM-405.pdf`.
24. K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. *Annals of Mathematics and Artificial Intelligence*, 1:303–338, 1990. Available from `http://www.springerlink.com/content/k3p11kl316m73764/`.
25. N. Nishida and M. Sakai. Completion after program inversion of injective functions. *Electr. Notes Theor. Comput. Sci.*, 237:39–56, 2009.
26. N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse computation programs of constructor term rewriting systems. *IEICE Trans. Inf. & Syst.*, J88-D-I(8):1171–1183, 2005 (in Japanese).
27. N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In *Proc. of the 16th International Conference on Rewriting Techniques and Applications*, vol. 3467 of *LNCS*, pp. 264–278, Springer, 2005.
28. N. Nishida and G. Vidal. Program inversion for tail recursive functions. In *Proc. of the 22nd International Conference on Rewriting Techniques and Applications*, vol. 10 of *LIPIcs*, pp. 283–298, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.
29. N. Nishida and G. Vidal. Conversion to first-order tail recursion for improving program inversion, 2012. *Submitted for publication.*
30. N. Nishida and G. Vidal. More specific term rewriting systems. In *the 21st International Workshop on Functional and (Constraint) Logic Programming*, Nagoya, Japan, 2012. Informal proceedings, available from the URL: `http://www.dsic.upv.es/~gvidal/german/wflp12/paper.pdf`.
31. N. Nishida and G. Vidal. Computing more specific versions of conditional rewriting systems. Technical report, available from the following URL: `http://www.dsic.upv.es/~gvidal/german/lopstr12/paperTR.pdf`.
32. E. Ohlebusch. *Advanced topics in term rewriting.* Springer-Verlag, UK, 2002.
33. F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Proc. of the Sixth Annual Symposium on Logic in Computer Science*, pp. 74–85, IEEE Computer Society, 1991.
34. G. Plotkin. Building-in equational theories. *Machine Intelligence*, 7:73–90, 1972.
35. J. G. Ramos, J. Silva, and G. Vidal. Fast Narrowing-Driven Partial Evaluation for Inductively Sequential Systems. In *Proc. of the 10th ACM SIGPLAN International Conference on Functional Programming*, pp. 228–239, ACM Press, 2005.
36. J. R. Slagle. Automated theorem-proving for theories with simplifiers, commutativity and associativity. *J. ACM*, 21(4):622–642, 1974.

Extending Matching Operation in Grammar Programs for Program Inversion *
(Extended Abstract)

Minami Niwa, Naoki Nishida, and Masahiko Sakai

Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan
{mniwa@sakabe.i., nishida@, sakai@}is.nagoya-u.ac.jp

Abstract. The inversion method proposed by Glück and Kawabe uses grammar programs as intermediate results, which comprise sequences of operations (data generation, matching, etc.). The semantics of the grammar programs is defined as a stack machine where data terms are stored in the stack. The matching operation for a constructor symbol pops up the topmost term of the stack and pushes its direct subterms to the stack if the topmost term is rooted by the constructor. This paper strengthens the matching operation by augmenting a parameter that specifies what position the matching operation can look into from the top of the stack. This extension is sometimes effective to avoid the failure execution of the determinization method based on LR parsing described in the latter half of the inversion method.

1 Introduction

Inverse computation for an n-ary functions f is, given an output v of f, the calculation of (all) the possible inputs $v_1, \ldots, v_n$ of f such that $f(v_1, \ldots, v_n) = v$. As an approach to inverse computation, *program inversion* has been studied [12, 11, 8, 7, 2–4, 1, 9, 5], which takes the definition of f as input and computes the definition of the inverse function f^{-1}. Surprisingly, the essential ideas of the existing methods for inversion are almost the same, except for [9, 5]. In general, function definitions for inverses, that are generated by the inversion methods, are non-deterministic with respect to the application of function definitions even if the target functions are injective. For this reason, determinization of such inverse programs is one of the most interesting topics in developing inversion methods and has been investigated in several ways [2–4, 6].

The inversion method LRinv, proposed by Glück and Kawabe [2–4], adopts an interesting approach to determinization. A functional program is first translated into a context-free grammar, called a *grammar program*, whose language is the sequences of atomic operations (data generation, matching, etc.) corresponding to the computation of the original functions—the language is semantically equivalent to the given functional program. The semantics of the operations is defined as a stack machine where data terms are stored in the stack (input arguments, intermediate results, and outputs), in which the meanings of operations are defined as *pop* and *push* operations on the stack. The grammar program is inverted by reversing context-free grammars and replacing each operation by its opposite operation, then it is determinized by a method based on *LR(0)*

* This work has been partially supported by *MEXT KAKENHI* #21700011.

parsing into a grammar program, which is easily convertible to a functional program without overlapping between function definitions. The mechanism of the determinization is complicated but quite interesting, e.g., some grammar program corresponding to non-terminating and overlapping function definition is transformed into the grammar program corresponding to a terminating and non-overlapping function definitions. Moreover, the determinization method is independent from the inversion principle, and thus, it is useful as a post-processing of any other inversion methods. However, the determinization is not successful for all grammar programs obtained by the inversion phase; there exist some examples for which this method fails but another inversion method succeeds (cf. [9]). The failure of the determinization is mainly caused by *shift/shift conflicts* that are branches in the collection of *item sets* with two or more different atomic operations, except for matching ones. In this determinization, grammar programs with such conflicts do not proceed to the code generation phase.

In this paper, we try to avoid shift/shift conflicts as much as possible, applying the code generation method to grammar programs. To this end, we extend the syntax and semantics of matching operations in grammar programs. In the abstract semantics of programs, any value used in the execution is freely accessed by means of variable names. The original matching operation for a constructor pops the topmost element from the stack and pushes direct subterms of the element to the stack if the topmost term is rooted by the constructor. This means that the operation refers to the topmost element of the stack only, and thus, the permutation of the elements in the stack is often necessary to access values addressed at non-topmost of the stack. To avoid the use of such permutations for matching operations, we extend matching operations by specifying a natural number k so as to allow them to take the kth topmost element of the stack. This extension sometimes avoids the failure execution caused by the shift/shift conflict, and thus, allows us to proceed to the code generation phase. We show an example for which the original method is not successful but the extended one succeeds in producing non-overlapping programs. We also show an unsuccessful example in the sense that the extended method produces an overlapping system, while the original method fails.

The contribution of this paper is to make LRinv strictly more powerful.

This paper is organized as follows. In Section 2, we recall LRinv by means of an example. In Section 3, we show an example for which LRinv is unsuccessful and describe an idea for avoiding shift/shift conflicts. In Section 4, we extend the syntax and semantics of matching operations in grammar programs, and show some examples. In Section 5, we briefly describe future work of this research.

2 Overview of LRinv

In this section, we briefly recall LRinv [2–4] by using an example, while we describe the complete syntax and semantics of grammar programs. We also recall the concrete definitions of item sets and conflicts related to item sets, called *shift/shift conflicts*, that influence the success or failure of the determinization method (i.e., LRinv).

LRinv assumes that the functions to be inverted are injective over inductive data structures. Throughout this paper, we use $\mathcal{C}$ and $\mathcal{F}$ for finite sets of *constructor* and *function* symbols that appears in programs, where each symbol

f has a fixed arity n, represented by f/n. The set of ground terms over $\mathcal{C}$ is denoted by $T(\mathcal{C})$. For the sake of readability, functional programs are written as *constructor TRSs* (cf., [10]). The syntax of grammar programs is defined by the following BNF:

$$\begin{array}{rll} q ::= & d_1 \dots d_m & \text{(grammar program)} \\ d ::= & f \to t_1 \dots t_m & \text{(definition)} \\ t ::= & a & \text{(atomic operation)} \\ & \mid f & \text{(function call)} \\ a ::= & c! & \text{(constructor application)} \\ & \mid c? & \text{(pattern matching)} \\ & \mid \pi & \text{(permutation)} \\ \pi ::= & (i_1 \dots i_n) & \end{array}$$

where $f \in \mathcal{F}$, $c \in \mathcal{C}$, $m \geq 0$, $n > 0$, $\{i_1, \dots, i_n\} = \{1, \dots, n\}$. Note that function symbols are *non-terminals* and others are *terminals*. For the sake of readability, this paper does not deal with the *duplication/equality* operator $\lfloor \_ \rfloor$. The set of grammar programs is denoted by $\mathcal{G}$. For a grammar program $G \in \mathcal{G}$ defining a function $f \in \mathcal{F}$, a notation $L(G, f)$ represents the set of operation sequences generated from the non-terminal f, where operations are function calls and atomic operations. The set of operations is denoted by $\mathcal{O}p$ and the set of atomic operations is denoted by $\mathcal{A}$. To represent sequences of objects, we may use the usual list constructor : and the empty list ϵ, e.g., the sequence 1 2 3 is denoted by $1 : 2 : 3 : \epsilon$. We may write @ as the binary operator for list concatenation.

The semantics $\to_G$ of a grammar program $G \in \mathcal{G}$ is defined over pairs of sequences operations and terms, a binary relation over $\mathcal{O}p^* \times T(\mathcal{C})^*$, as follows:

$$\begin{array}{rll} (f : ts, vs) & \to_G (us@ts, vs) & \text{where } f \to us \in G \\ (c! : ts, v_1 : \dots : v_n : vs) & \to_G (ts, c(v_1, \dots, v_n) : vs) & \text{where } c/n \in \mathcal{C} \\ (c? : ts, c(v_1, \dots, v_n) : vs) & \to_G (ts, v_1 : \dots : v_n : vs) & \text{where } c/n \in \mathcal{C} \\ ((i_1 \ \dots \ i_n) : ts, v_1 : \dots : v_n : vs) & \to_G (ts, v_{i_1} : \dots : v_{i_n} : vs) & \end{array}$$

Let f be an n-ary function that returns a tuple of m values. When computing $f(v_1, \dots, v_n)$, we start with $(f : \epsilon, v_1 : \dots : v_n : \epsilon)$ and obtain the result $(u_1, \dots, u_m)$ if $(f : \epsilon, v_1 : \dots : v_n : \epsilon) \to_G^* (\epsilon, u_1 : \dots : u_m : \epsilon)$. The second component of the pairs plays a role of a *stack*. Let c be an n-ary constructor symbol. The constructor application $c!$ pops the n topmost values $v_1, \dots, v_n$ from the stack and pushes the value $c(v_1, \dots, v_n)$ onto the stack. The pattern matching $c?$ pops the topmost value $c(v_1, \cdots, v_n)$ from the stack and then pushes values $v_1, \dots, v_n$ onto the stack. A permutation $(i_1 \ \dots \ i_n)$ reorders the n topmost values on the stack by moving in parallel the i_jth element to the jth position. We merge sequences of permutations in G into single optimized ones, such as (2 1) (3 1 2) into (3 2 1), and remove identity permutations, such as (1 2 3).

To invert a grammar program G, each definition $f \to t_1 \ \dots \ t_n$ is inverted to the definition $f^{-1} \to (t_n)^{-1} \ \dots \ (t_1)^{-1}$, where $(g)^{-1} = g^{-1}$ for $g \in \mathcal{F}$, $(c?)^{-1} = c!$ and $(c!)^{-1} = c?$ for $c \in \mathcal{C}$, and $((i_1 \ \dots \ i_m))^{-1} = (j_1 \ \dots \ j_m)$ with $k = i_{j_k}$ for a permutation $(i_1 \ \dots \ i_m)$ [3].

Example 1. Let us consider the function snoc defined by the following TRS:

$$\left\{ \begin{array}{c} \mathsf{snoc}(\mathsf{nil}, y) \to \mathsf{cons}(y, \mathsf{nil}) \\ \mathsf{snoc}(\mathsf{cons}(x, xs), y) \to \mathsf{cons}(x, \mathsf{snoc}(xs, y)) \end{array} \right\}$$

where $\mathcal{C} = \{\mathsf{nil}/0, \mathsf{cons}/2, \ldots\}$ and $\mathcal{F} = \{\mathsf{snoc}/2\}$. The function snoc appends the second argument to the end of the first argument, e.g., $\mathsf{snoc}(\mathsf{cons}(1, \mathsf{cons}(2, \mathsf{nil})), 3) = \mathsf{cons}(1, \mathsf{cons}(2, \mathsf{cons}(3, \mathsf{nil})))$. This program is translated into the following grammar program:

$$G_{\mathsf{snoc}} = \left\{ \mathsf{snoc} \to \mathsf{nil?}\ \mathsf{nil!}\ (2\ 1)\ \mathsf{cons!} \qquad \mathsf{snoc} \to \mathsf{cons?}\ (2\ 3\ 1)\ \mathsf{snoc}\ (2\ 1)\ \mathsf{cons!} \right\}$$

The grammar program G_{snoc} is inverted as follows:

$$G_{\mathsf{snoc}^{-1}} = \left\{ \begin{array}{l} \mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1)\ \mathsf{nil?}\ \mathsf{nil!} \\ \mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1)\ \mathsf{snoc}^{-1}\ (3\ 1\ 2)\ \mathsf{cons!} \end{array} \right\}$$

$G_{\mathsf{snoc}^{-1}}$ is translated back into the following *conditional* TRS [10]:

$$R_{\mathsf{snoc}^{-1}} = \left\{ \begin{array}{l} \mathsf{snoc}^{-1}(\mathsf{cons}(x, \mathsf{nil})) \to (\mathsf{nil}, x) \\ \mathsf{snoc}^{-1}(\mathsf{cons}(x, y)) \to (\mathsf{cons}(x, z), w) \Leftarrow \mathsf{snoc}^{-1}(y) \twoheadrightarrow (z, w) \end{array} \right\}$$

Note that this TRS is also obtained by another inversion method, e.g., [9]. If the inverse TRS resulting from this simple inversion above is non-overlapping, program inversion is completed. However, $G_{\mathsf{snoc}^{-1}}$ does not correspond to a non-overlapping TRS (see $R_{\mathsf{snoc}^{-1}}$). Thus, we proceed to the determinization method based LR(0) parsing as follows.

Collecting item sets. Given a grammar program, the collection of LR(0) item sets is computed by a closure operation (see below).

Code generation. Given a *conflict-free* collection of LR(0) item sets, a grammar program whose the corresponding TRS is non-overlapping is generated. For lack of space, we do not describe the detail.

In the following, we recall how to construct from a grammar program the canonical LR(0) collection. An *LR(0) parse item* (an *item*, for short) is a function definition with a dot "$\cdot$" at some position on the right side. An item indicates how much of a sequence of operations has been performed at a certain point during the evaluation of a grammar program. For example, the item

$$\mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1) \cdot \mathsf{nil?}\ \mathsf{nil!}$$

indicates that we have successfully performed a pattern matching cons? and a permutation (2 1), and that we hope to find a value nil on the top the stack. We group items together into *LR(0) item sets* (*item sets*, for short) which represent the set of all possible operations that a computation can take at a certain point during evaluation. For a grammar program q, to calculate the collection $\mathcal{I}$ of all reachable item sets, we introduce two relations, *shift* and *reduce*, which correspond to determining the parse action in LR(0) parsing:

$$\begin{array}{lll} (\mathsf{shift}) & I_1 \rightsquigarrow^t I_2 & \text{iff} \quad I_2 = \{f \to ts_1\ t \cdot ts_2 \mid f \to ts_1 \cdot t\ ts_2 \in \mathit{closure}(I_1)\} \\ (\mathsf{reduce}) & I \hookrightarrow f & \text{iff} \quad f \to t_1 \ldots t_n \cdot \in \mathit{closure}(I) \end{array}$$

where the function $\mathit{closure} : \mathcal{I} \to \mathcal{I}$ and its auxiliary function $\mathit{cls} : \mathcal{I} \times \mathcal{F} \to \mathcal{I}$ are defined as $\mathit{closure}(I) = \mathit{cls}(I, \emptyset)$, $\mathit{cls}(\emptyset, F) = \emptyset$, and $\mathit{cls}(I, F) = I \cup \mathit{cls}(\{f \to \cdot\ ts \mid f \to ts \in q, f \in F'\}, F \cup F')$ where $F' = \{f \mid f' \to ts_1 \cdot f\ ts_2 \in I, f \notin F\}$. Shift with an operation t transforms item set I_1 to item set I_2 under t, reduce is to return from item set I after n operations of function f were shifted, and $\mathit{closure}(I)$ calculates a new item set I of grammar program q. Intuitively, item $f' \to ts_1 \cdot t\ ts_2 \in \mathit{closure}(I)$ indicates that, at some point during evaluation

of program q, we may perform operation t. The closure calculation terminates since there is only a finite number of different items for every program.

The set of all item sets of a program q, the *canonical collection* $\mathcal{I}_q$, is defined as follows: $\mathcal{I}_q = \{I \mid I_0 \rightsquigarrow^* I\}$ where $main(q)$ denotes the main function of q, s is a new function symbol, $I_0 = \{\mathsf{s} \to \cdot\, h\}$ and $h = main(q)$; $I_1 \rightsquigarrow^* I_2$ iff $I_1 = I_2 \ \vee\ (\exists t.\ \exists I'.\ I_1 \rightsquigarrow^t I' \ \wedge\ I' \rightsquigarrow^* I_2)$. The set is finite since there is only a finite number of different item sets.

At the end of Step 1 (collecting item sets), we examine whether there is a conflict in the collection or not: if there is no conflict, we proceed to Step 2 (code generation). In this paper, we focus on *shift/shift conflicts* only since such conflicts are much more important for the determinization method. If an item set has two or more shift actions labeled with atomic operations, all of them must be matching operations. Shift/shift conflict is the case that an item set has two or more shift actions labeled with atomic operations where one or more is not a matching operations. For instance, $I \rightsquigarrow^{\mathsf{nil!}} I'$ and $I \rightsquigarrow^{\mathsf{cons?}} I''$ are in shift/shift conflicts, while $I \rightsquigarrow^{\mathsf{nil?}} I'$ and $I \rightsquigarrow^{\mathsf{cons?}} I''$ are not.

Example 2. Consider the grammar program $G_{\mathsf{snoc}^{-1}}$ in Example 1 again. The collection of item sets and relations are as follows:

– Collection of item sets:

$$I_0 = \left\{ \mathsf{s} \to \cdot \mathsf{snoc}^{-1} \right\} \quad I_1 = \left\{ \begin{array}{l} \mathsf{snoc}^{-1} \to \mathsf{cons?} \cdot (2\ 1)\ \mathsf{nil?}\ \mathsf{nil!} \\ \mathsf{snoc}^{-1} \to \mathsf{cons?} \cdot (2\ 1)\ \mathsf{snoc}^{-1}\ (3\ 1\ 2)\ \mathsf{cons!} \end{array} \right\}$$

$$I_2 = \left\{ \begin{array}{l} \mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1) \cdot \mathsf{nil?}\ \mathsf{nil!} \\ \mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1) \cdot \mathsf{snoc}^{-1}\ (3\ 1\ 2)\ \mathsf{cons!} \end{array} \right\}$$

$$I_3 = \left\{ \mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1)\ \mathsf{nil?} \cdot \mathsf{nil!} \right\} \quad \cdots \quad I_8 = \left\{ \mathsf{s} \to \mathsf{snoc}^{-1} \cdot \right\}$$

– Relations:

$$\begin{array}{llllll} I_0 \rightsquigarrow^{\mathsf{snoc}^{-1}} I_8 & I_0 \rightsquigarrow^{\mathsf{cons?}} I_1 & I_1 \rightsquigarrow^{(2\ 1)} I_2 & I_2 \rightsquigarrow^{\mathsf{snoc}^{-1}} I_5 & I_2 \rightsquigarrow^{\mathsf{nil?}} I_3 \\ I_2 \rightsquigarrow^{\mathsf{cons?}} I_1 & I_3 \rightsquigarrow^{\mathsf{nil!}} I_4 & \cdots & I_8 \hookrightarrow^1 \mathsf{s} & \end{array}$$

Since this collection is conflict-free, we proceeds to Step 2 (the code generation), transforming the grammar program $G_{\mathsf{snoc}^{-1}}$ into the following grammar program:

$$\left\{ \mathsf{snoc}^{-1} \to \mathsf{cons?}\ (2\ 1)\ \mathsf{f} \qquad \mathsf{f} \to \mathsf{nil?}\ \mathsf{nil!} \qquad \mathsf{f} \to \mathsf{cons?}\ (2\ 1)\ \mathsf{f}\ (3\ 1\ 2)\ \mathsf{cons!} \right\}$$

where f is a fresh non-terminal symbol introduced by means of the determinization. Considering f as an auxiliary function of snoc, this resulting grammar program is translated back into the following non-overlapping conditional TRS:

$$\left\{ \begin{array}{r} \mathsf{snoc}^{-1}(\mathsf{cons}(x, xs)) \to \mathsf{f}(xs, x) \\ \mathsf{f}(\mathsf{nil}, y) \to (\mathsf{nil}, y) \\ \mathsf{f}(\mathsf{cons}(x, xs), y) \to (\mathsf{cons}(y, z), w) \Leftarrow \mathsf{f}(xs, x) \twoheadrightarrow (z, w) \end{array} \right\}$$

3 Observing Shift/Shift Conflicts

In this section, we observe an example where a shift/shift conflict appears in the collection of the item sets.

Example 3. Consider the following constructor TRS, a variant of unbin shown in [3]:

$$\left\{\begin{array}{rr} \mathsf{unbin2}(x) \to \mathsf{ub}(x, \mathsf{nil}) & \mathsf{ub}(\mathsf{suc}(\mathsf{zero}), y) \to y \\ \mathsf{ub}(\mathsf{suc}(\mathsf{suc}(x)), y) \to \mathsf{ub}(\mathsf{suc}(x), \mathsf{inc}(y)) & \mathsf{inc}(\mathsf{nil}) \to \mathsf{cons}(0, \mathsf{nil}) \\ \mathsf{inc}(\mathsf{cons}(0,xs)) \to \mathsf{cons}(1, xs) & \mathsf{inc}(\mathsf{cons}(1,xs)) \to \mathsf{cons}(0, \mathsf{inc}(xs)) \end{array}\right\}$$

where $\mathcal{C} = \{\mathsf{nil}/0, \mathsf{cons}/2, 0/0, 1/0, \mathsf{zero}/0, \mathsf{suc}/1, \ldots\}$ and $\mathcal{F} = \{\mathsf{unbin2}/1, \mathsf{ub}/2, \mathsf{inc}/1\}$. The function $\mathsf{unbin2}$ converts positive natural numbers represented as $\mathsf{suc}(\mathsf{zero})$, $\mathsf{suc}(\mathsf{suc}(\mathsf{zero}))$, ... to binary-numeral expressions $\mathsf{nil}, \mathsf{cons}(0, \mathsf{nil}), \ldots$.[1] This program is translated and inverted to the following grammar program:

$$G_{\mathsf{unbin2}^{-1}} = \left\{\begin{array}{l} \mathsf{unbin2}^{-1} \to \mathsf{ub}^{-1}\ (2\ 1)\ \mathsf{nil}? \\ \mathsf{ub}^{-1} \to \mathsf{zero}!\ \mathsf{suc}! \\ \mathsf{ub}^{-1} \to \mathsf{ub}^{-1}\ \mathsf{suc}?\ (2\ 1)\ \mathsf{inc}^{-1}\ (2\ 1)\ \mathsf{suc}!\ \mathsf{suc}! \\ \mathsf{inc}^{-1} \to \mathsf{cons}?\ 0?\ \mathsf{nil}?\ \mathsf{nil}! \\ \mathsf{inc}^{-1} \to \mathsf{cons}?\ 1?\ 0!\ \mathsf{cons}! \\ \mathsf{inc}^{-1} \to \mathsf{cons}?\ 0?\ \mathsf{inc}^{-1}\ 1!\ \mathsf{cons}! \end{array}\right\}$$

This grammar program corresponds to the following conditional TRS:

$$\left\{\begin{array}{l} \mathsf{unbin2}^{-1}(x) \to y \ \Leftarrow \mathsf{ub}^{-1}(x) \twoheadrightarrow (y, \mathsf{nil}) \\ \mathsf{ub}^{-1}(x) \to (\mathsf{suc}(\mathsf{zero}), x) \\ \mathsf{ub}^{-1}(x) \to (\mathsf{suc}(\mathsf{suc}(y)), \mathsf{inc}^{-1}(z)) \ \Leftarrow\ \mathsf{ub}^{-1}(x) \twoheadrightarrow (\mathsf{suc}(y), z) \\ \qquad\qquad \vdots \end{array}\right\}$$

This conditional TRS is not *operationally terminating* since $\mathsf{ub}^{-1}(x)$ calls $\mathsf{ub}^{-1}(x)$ itself. The item sets and their relations for $G_{\mathsf{unbin2}^{-1}}$ are as follows:

- Collection of item sets:

$$I_0 = \left\{ \mathsf{s} \to \cdot \mathsf{unbin2}^{-1} \right\}$$

$$I_1 = \left\{\begin{array}{r} \mathsf{unbin2}^{-1} \to \mathsf{ub}^{-1} \cdot (2\ 1)\ \mathsf{nil}? \\ \mathsf{ub}^{-1} \to \mathsf{ub}^{-1} \cdot \mathsf{suc}?\ (2\ 1)\ \mathsf{inc}^{-1}\ (2\ 1)\ \mathsf{suc}!\ \mathsf{suc}! \end{array}\right\}$$

$$I_2 = \left\{ \mathsf{ub}^{-1} \to \mathsf{zero}! \cdot \mathsf{suc}! \right\} \quad I_3 = \left\{ \mathsf{unbin2}^{-1} \to \mathsf{ub}^{-1}\ (2\ 1) \cdot \mathsf{nil}? \right\}$$

$$I_4 = \left\{ \mathsf{ub}^{-1} \to \mathsf{ub}^{-1}\ \mathsf{suc}? \cdot (2\ 1)\ \mathsf{inc}^{-1}\ (2\ 1)\ \mathsf{suc}!\ \mathsf{suc}! \right\} \quad \cdots$$

- Relations:

$$\begin{array}{llllll} I_0 \rightsquigarrow^{\mathsf{unbin2}^{-1}} I_{22} & I_0 \rightsquigarrow^{\mathsf{ub}^{-1}} I_1 & I_0 \rightsquigarrow^{\mathsf{zero}!} I_2 & I_1 \rightsquigarrow^{(2\ 1)} I_3 & & \\ I_1 \rightsquigarrow^{\mathsf{suc}?} I_4 & I_2 \rightsquigarrow^{\mathsf{suc}?} I_5 & I_3 \rightsquigarrow^{\mathsf{nil}?} I_6 & I_4 \rightsquigarrow^{(2\ 1)} I_7 & \cdots & \end{array}$$

Item set I_1 has two shift relations $I_1 \rightsquigarrow^{(2\ 1)} I_3$ and $I_1 \rightsquigarrow^{\mathsf{suc}?} I_4$. These relations labeled with a matching operation and a non-matching operation cause a shift/shift conflict. Therefore, we cannot proceed to Step 2.

The shift relation $I_1 \rightsquigarrow^{(2\ 1)} I_3 \rightsquigarrow^{\mathsf{nil}?} I_6$ in Example 3 indicates that the structure of the topmost element is not examined and the second topmost element is examined whether to be nil. In other words, the sequence $(2\ 1)\ \mathsf{nil}?$ of the relation corresponds to the application of the matching operation to the second

[1] To make $\mathsf{unbin2}$ simpler as an injective function, we use this coding instead of the usual Peano-style coding nil (as 0), $\mathsf{cons}(1, \mathsf{nil})$ (as 1), $\mathsf{cons}(1, \mathsf{cons}(0, \mathsf{nil}))$ (as 2), ...

topmost element of the stack, and thus, the sequence can be considered a single action in the execution of the corresponding functional program. From this observation, combining (2 1) and nil? as a single matching operations seems useful to avoid the shift/shift conflict. In the next section, to represent such operation sequences as single matching operations, we extend the syntax and semantics of the matching operation.

4 Extension of Matching Operation

In this section, we extend the syntax and semantics of the matching operation so as to refer to non-topmost elements in the stack.

As described at the end of the previous section, we represent operation sequences $\pi : c?$ such as (2 1) nil? by single operations. To achieve this, we introduce new matching operations $c_{(k)}?$ for $c \in \mathcal{C}$ where $k > 0$, e.g., $\mathsf{nil}_{(2)}?$ for (2 1) nil?. $\mathcal{G}_{\text{ext}}$ denotes the set of grammar programs obtained by adding the following atomic operations to the syntax for $\mathcal{G}$:

$$c_{(k)}? \qquad \text{where } c \in \mathcal{C} \text{ and } k > 0$$

The new operation $c_{(k)}?$ plays a role of the matching operation not only for the topmost element but for the kth topmost element. Thus, the semantics of a program G' in $\mathcal{G}_{\text{ext}}$ is obtained by adding the following case to $\to_G$:

$$\begin{aligned}(c_{(k)}? : ts, v_1 : \cdots : v_{k-1} : c(u_1, \ldots, u_n) : vs) \\ \to_{G'} (ts, v_1 : \cdots : v_{k-1} : u_1 : \cdots : u_n : vs)\end{aligned}$$

where $c/n \in \mathcal{C}$. We denote $\mathcal{A} \cup \{c_{(k)}? \mid c \in \mathcal{C},\ k > 0\}$ by $\mathcal{A}_{\text{ext}}$.

In the rest of this section, we show that the replacement of $(k\ i_1\ \ldots\ i_{k-1}) : c?$ by $c_{(k)}? : (k\ k+1\ \ldots\ k+(n-1)\ i_1\ \ldots\ i_{k-1})$ preserves the equivalence of the semantics.

We first formalize the replacement $\Leftrightarrow$ as the symmetric closure satisfying all of the following:

- $ts : (k\ i_i\ \ldots\ i_{k-1}) : c? : us \Leftrightarrow ts : c_{(k)}? : (k\ k{+}1\ \ldots\ k{+}(n{-}1)\ i_1\ \ldots\ i_{k-1}) : us$ where $c/n \in \mathcal{C}$, $n > 0$, $\{i_1, \ldots, i_{k-1}\} = \{1, \ldots, k-1\}$ and
- $ts : (k\ i_1\ \ldots\ i_{k-1}) : c? : us \Leftrightarrow ts : c_{(k)}? : (i_1\ \ldots\ i_{k-1}) : us$, where $c/0 \in \mathcal{C}$, and $\{i_1, \ldots, i_{k-1}\} = \{1, \ldots, k-1\}$.

For a sequence ts of operations over the extended grammar language, the set $\text{rep}_{\Leftrightarrow}(ts)$ denotes the set of sequences obtained from ts by applying the reflexive and transitive closure of $\Leftrightarrow$: $\text{rep}_{\Leftrightarrow}(ts) = \{ts' \mid ts \Leftrightarrow^* ts'\}$ where continuous sequences of permutations are optimized as much as possible.

Theorem 4. *Let $G = \{g_1 \to ts_1, \ldots, g_n \to ts_n\}$ be a grammar program in $\mathcal{G}$ that defines an n-ary function f returning m values, and G' be a grammar program in $\mathcal{G}_{\text{ext}}$ such that $G' = \{g_i \to ts_i' \mid 1 \le i \le n,\ ts_i' \in \text{rep}_{\Leftrightarrow}(ts)\}$. Then, for all terms $v_1, \ldots, v_n, u_1, \ldots, u_m \in T(\mathcal{C})$, $(f : \epsilon, v_1 : \cdots : v_n : \epsilon) \to_G^* (\epsilon, u_1 : \cdots : u_m : \epsilon)$ iff $(f : \epsilon, v_1 : \cdots : v_n : \epsilon) \to_{G'}^* (\epsilon, u_1 : \cdots : u_m : \epsilon)$.*

Proof. Let ts be a sequence ts in $\mathcal{A}_{\text{ext}}^*$, $ts' \in \text{rep}_{\Leftrightarrow}(ts)$, $G = \{f \to ts\}$, and $G' = \{f \to ts'\}$. Then, for all sequences $vs, us \in T(\mathcal{C})^*$, $(ts, vs) \to_G^* (\epsilon, us)$ iff $(ts, vs) \to_{G'}^* (\epsilon, us)$. This can be straightforwardly proved by induction on the length of ts. This theorem follows from this claim. □

Example 5. Consider the grammar program $G_{\mathsf{unbin2}^{-1}}$ in Example 3 again. By replacing (2 1) nil? in the first rule by $\mathsf{nil}_{(2)}?$, we obtain the following grammar program:

$$G'_{\mathsf{unbin2}^{-1}} = \left\{ \mathsf{unbin2}^{-1} \to \mathsf{ub}^{-1}\ \mathsf{nil}_{(2)}? \quad \cdots \right\}$$

The collection of $G'_{\mathsf{unbin2}^{-1}}$ is as follows:

– Collection of item sets:

$$I_0 = \left\{ \mathsf{s} \to \cdot\mathsf{unbin2}^{-1} \right\}$$

$$I_1 = \left\{ \begin{array}{l} \mathsf{unbin2}^{-1} \to \mathsf{ub}^{-1} \cdot \mathsf{nil}_{(2)}? \\ \quad \mathsf{ub}^{-1} \to \mathsf{ub}^{-1} \cdot \mathsf{suc?}\ (2\ 1)\ \mathsf{inc}^{-1}\ (2\ 1)\ \mathsf{suc!}\ \mathsf{suc!} \end{array} \right\} \quad \cdots$$

– Relations:

$$I_0 \rightsquigarrow^{\mathsf{unbin2}^{-1}} I_{21} \quad I_0 \rightsquigarrow^{\mathsf{ub}^{-1}} I_1 \quad I_0 \rightsquigarrow^{\mathsf{zero!}} I_2 \quad I_1 \rightsquigarrow^{\mathsf{nil}_{(2)}?} I_3 \quad I_1 \rightsquigarrow^{\mathsf{suc?}} I_4 \quad \cdots$$

Although I_1 has a two shift relation by atomic operations, both atomic operations are matching operations, i.e., there is no shift/shift conflict unlike the collection in Example 3. Therefore, we can proceed to the code generation phase. Ignoring the extension of the syntax and semantics (i.e., considering $\mathsf{nil}_{(2)}$ a constructor symbol), we apply the code generation method to $G'_{\mathsf{unbin2}^{-1}}$, thus obtaining the following grammar program:

$$\left\{ \begin{array}{lll} \mathsf{unbin2}^{-1} \to \mathsf{zero!}\ \mathsf{suc!}\ \mathsf{f} & & \\ \quad \mathsf{f} \to \mathsf{nil}_{(2)}? & \mathsf{f} \to \mathsf{suc?}\ (2\ 1)\ \mathsf{cons?}\ \mathsf{g}\ (2\ 1)\ \mathsf{suc!}\ \mathsf{suc!}\ \mathsf{f} \\ \quad \mathsf{g} \to \mathsf{0?}\ \mathsf{h} & \mathsf{g} \to \mathsf{1?}\ \mathsf{0!}\ \mathsf{cons!} \\ \quad \mathsf{h} \to \mathsf{nil?}\ \mathsf{nil!} & \mathsf{h} \to \mathsf{cons?}\ \mathsf{g}\ \mathsf{1!}\ \mathsf{cons!} \end{array} \right\}$$

By replacing $\mathsf{nil}_{(2)}?$ by (2 1) nil back and by translating the grammar program back into a constructor TRS, we obtain the following non-overlapping system:

$$\left\{ \begin{array}{lll} \mathsf{unbin2}^{-1}(x) \to \mathsf{f}(\mathsf{suc}(\mathsf{zero}), x) & & \\ \quad \mathsf{f}(x, \mathsf{nil}) \to x & \mathsf{f}(\mathsf{suc}(x), \mathsf{cons}(y, ys)) \to \mathsf{f}(\mathsf{suc}(\mathsf{suc}(x)), \mathsf{g}(y, ys)) \\ \quad \mathsf{g}(0, y) \to \mathsf{h}(y) & \mathsf{g}(1, y) \to \mathsf{cons}(0, y) \\ \quad \mathsf{h}(\mathsf{nil}) \to \mathsf{nil} & \mathsf{h}(\mathsf{cons}(y, ys)) \to \mathsf{cons}(1, \mathsf{g}(y, ys)) \end{array} \right\}$$

This TRS is desirable as an inverse of unbin2 although the correctness for the application of the determinization method has not been guaranteed yet.

As another solution, it may seem useful to exchange the first and second arguments of the function symbol ub—replace all $\mathsf{ub}(t_1, t_2)$ in the initial TRS by $\mathsf{ub}(t_2, t_1)$. Unfortunately, this is not a solution since a shift/shift conflict arises in applying the inversion and determinization method to the TRS obtained by the replacement.

Here, we apply the transformation $\Leftrightarrow$ of π c? to $\mathsf{c}_{(k)}?$ π' by hand in order to avoid the shift/shift conflicts we faced. Since the number of candidates for G' in Theorem 4 is finite, to automate this application, the depth-first search is sufficient to detect sequences to be replaced—such sequences are related to shift/shift conflict branches of item sets. This strategy does not allow us to apply the replacement to grammar programs that have no shift/shift conflict in the collections. This indicates that the extended method in this paper is successful for all the examples for which the original method is successful. Therefore, the extended method is strictly better than the original one because of unbin2.

While we mentioned successful examples only, this extension does not always succeed in the determinization. In the following, we show an example where we succeed in avoiding a shift/shift conflict in the collection but fail to produce a non-overlapping TRS while an overlapping and terminating one is produced.

Example 6. Consider the following TRS, a variant of unbin2:

$$\left\{\begin{array}{ll} \mathsf{unbin3}(x) \to \mathsf{ub}(x, \mathsf{cons}(0, \mathsf{nil})) & \\ \mathsf{ub}(\mathsf{suc}(\mathsf{zero}), y) \to y & \mathsf{ub}(\mathsf{suc}(\mathsf{suc}(x)), y) \to \mathsf{ub}(\mathsf{suc}(x), \mathsf{inc}(y)) \\ \vdots & \end{array}\right\}$$

where $\mathcal{C} = \{\mathsf{nil}/0, \mathsf{cons}/2, 0/0, 1/0, \mathsf{zero}/0, \mathsf{suc}/1, \ldots\}$ and $\mathcal{F} = \{\mathsf{unbin3}/1, \mathsf{ub}/2, \mathsf{inc}/1\}$. This program is inverted into the following conditional TRS:

$$\left\{\begin{array}{lll} \mathsf{unbin3}^{-1}(x) \to z & \Leftarrow & \mathsf{ub}^{-1}(x) \to (z, \mathsf{cons}(0, \mathsf{nil})) \\ \mathsf{ub}^{-1}(x) \to (\mathsf{suc}(\mathsf{zero}), x) & & \\ \mathsf{ub}^{-1}(x) \to (\mathsf{suc}(\mathsf{suc}(y), \mathsf{inc}^{-1}(z)) & \Leftarrow & \mathsf{ub}^{-1}(x) \twoheadrightarrow (\mathsf{suc}(y), z) \\ \vdots & & \end{array}\right\}$$

This conditional TRS is not operationally terminating, either. By replacing (2 1) cons? in the first rule by $\mathsf{cons}_{(2)}$? (2 3 1), we obtain the following grammar program:

$$G'_{\mathsf{unbin3}^{-1}} = \left\{ \mathsf{unbin3}^{-1} \to \mathsf{ub}^{-1}\ \mathsf{cons}_{(2)}?\ (2\ 3\ 1)\ 0\ \mathsf{nil}? \quad \cdots \right\}$$

By applying the determinization method to $G_{\mathsf{unbin3}^{-1}}$ again, we obtain the following grammar program:

$$\left\{\begin{array}{ll} \mathsf{unbin3}^{-1} \to \mathsf{zero!}\ \mathsf{suc!}\ \mathsf{f} & \\ \mathsf{f} \to \mathsf{cons}_{(2)}?\ (2\ 3\ 1)\ 0\ \mathsf{nil}? & \mathsf{f} \to \mathsf{suc}?\ (2\ 1)\ \mathsf{cons}?\ \mathsf{g}\ (2\ 1)\ \mathsf{suc!}\ \mathsf{suc!}\ \mathsf{f} \\ \mathsf{g} \to 0?\ \mathsf{h} & \mathsf{g} \to 1?\ 0!\ \mathsf{cons!} \\ \mathsf{h} \to \mathsf{nil}?\ \mathsf{nil!} & \mathsf{h} \to \mathsf{cons}?\ \mathsf{g}\ 1!\ \mathsf{cons!} \end{array}\right\}$$

This grammar program corresponds to the following TRS:

$$\left\{\begin{array}{ll} \mathsf{unbin3}^{-1}(x) \to \mathsf{f}(\mathsf{suc}(\mathsf{zero}), x) & \mathsf{f}(x, \mathsf{cons}(0, \mathsf{nil})) \to x \\ \mathsf{f}(\mathsf{suc}(x), \mathsf{cons}(y, ys))) \to \mathsf{f}(\mathsf{suc}(\mathsf{suc}(x)), \mathsf{g}(y, ys)) & \mathsf{g}(0, y) \to \mathsf{h}(y) \\ \mathsf{g}(1, y) \to \mathsf{cons}(0, y) & \mathsf{h}(\mathsf{nil}) \to \mathsf{nil} \\ \mathsf{h}(\mathsf{cons}(y, ys)) \to \mathsf{cons}(1, \mathsf{g}(y, ys)) & \end{array}\right\}$$

This TRS is still overlapping but terminating. The extended method fails to produce a non-overlapping TRS but transforms a non-operationally-terminating conditional TRS to a terminating one. Therefore, the extended determinization is meaningful for $G'_{\mathsf{unbin3}^{-1}}$ although it fails to produce a non-overlapping program.

Our approach is a breakthrough for avoiding shift/shift conflicts but not for producing non-overlapping systems since, e.g., $\mathsf{nil}_{(2)}$? in Example 5 is a pattern matching for the second topmost element of the stack but suc? is one for the topmost element, i.e., the branches labeled with $\mathsf{nil}_{(2)}$? and suc? do not correspond to the disjoint pattern matching. Though, the code generation sometimes produces non-overlapping programs, and thus, our approach is sometimes meaningful.

5 Future Work

As matters stand, proving the correctness of the determinization for extended grammar programs is left unanswered. We believe that the application of the determinization method to extended grammar programs is correct if no fresh atomic operators is introduced by means of the application. This is because the correctness of the determinization method for original grammar programs guarantees the equivalence of words generated by the input grammar and its resulting grammar. We will prove the correctness by using this intuitive observation.

We showed a successful example and an unsuccessful example of the extended LRinv, but we have never discussed the reason why the extended method succeeds in producing the non-overlapping TRSs in the two example and why produces the overlapping (and terminating) TRS. As our future work, we will characterize the difference between the successful and unsuccessful examples, clarifying a sufficient condition for producing non-overlapping TRSs. Moreover, we should compare the extended method with the existing inversion methods (e.g., [9]). In general, the resulting programs of the inversion methods are different. For this reason, it is very difficult to compare the inversion methods from a theoretical point of view. Thus, we will compare the extended method with the other inversion methods by means of the benchmarks shown in several papers on program inversion.

Acknowledgements
We thank the anonymous reviewers for their useful comments to improve this paper.

References

1. J. M. Almendros-Jiménez and G. Vidal. Automatic partial inversion of inductively sequential functions. In *Proc. of the 18th Int'l Symp. on Implementation and Application of Functional Languages*, vol. 4449 of *LNCS*, pp. 253–270, Springer, 2006.
2. R. Glück and M. Kawabe. A program inverter for a functional language with equality and constructors. In *Proc. of the first Asian Symposium on Programming Languages and Systems*, vol. 2895 of *LNCS*, pp. 246–264, Springer, 2003.
3. R. Glück and M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundam. Inform.*, 66(4):367–395, 2005.
4. M. Kawabe and R. Glück. The program inverter LRinv and its structure. In *Proc. of the 7th Int'l Symposium on Practical Aspects of Declarative Languages*, vol. 3350 of *LNCS*, pp. 219–234, Springer, 2005.
5. K. Matsuda, S.-C. Mu, Z. Hu, and M. Takeichi. A grammar-based approach to invertible programs. In *Proc. of the 19th European Symposium on Programming*, vol. 6012 of *LNCS*, pp. 448–467, Springer, 2010.
6. N. Nishida and M. Sakai. Completion after program inversion of injective functions. *Electr. Notes Theore. Comput. Sci.*, 237:39–56, 2009.
7. N. Nishida, M. Sakai, and T. Sakabe. Generation of inverse computation programs of constructor term rewriting systems. *IEICE Trans. Inf. & Syst.*, J88-D-I(8):1171–1183, 2005 (in Japanese).
8. N. Nishida, M. Sakai, and T. Sakabe. Partial inversion of constructor term rewriting systems. In *Proc. of the 16th Int'l Conf. on Rewriting Techniques and Applications*, vol. 3467 of *LNCS*, pp. 264–278, Springer, 2005.
9. N. Nishida and G. Vidal. Program inversion for tail recursive functions. In *Proc. of the 22nd Int'l Conf. on Rewriting Techniques and Applications*, vol. 10 of *LIPIcs*, pp. 283–298, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011.
10. E. Ohlebusch. *Advanced topics in term rewriting.* Springer-Verlag, UK, 2002.
11. A. Romanenko. Inversion and metacomputation. In *Proc. of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Sigplan Notices, 26(9), pp. 12–22, ACM, New York, 1991.
12. A. Romanenko. The Generation of Inverse Functions in Refal. In *Proc. of the Int'l Workshop on Partial Evaluation and Mixed Computation*, pp. 427–444, North-Holland, 1988.

Towards a Generic Framework for Guided Test Case Generation in CLP

José Miguel Rojas[1] and Miguel Gómez-Zamalloa[2]

[1] Technical University of Madrid, Spain
[2] DSIC, Complutense University of Madrid, Spain

Abstract. It is well known that performing test case generation by symbolic execution on large programs becomes quickly impracticable due to the path explosion phenomenon. This issue is considered a major challenge in the software testing arena. Another common limitation in the field is that test case generation by symbolic execution tends to produce an unnecessarily large number of test cases even for medium size programs. In this paper we propose a constraint logic programming approach to devise a generic framework to guide symbolic execution and thus test case generation. We show how the framework can help alleviate these scalability drawbacks that most symbolic execution-based test generation approaches endure.

1 Introduction

Testing remains a mostly manual stage within the software development process. Test Case Generation (TCG) is devoted to the automation of a crucial part of the testing process, the generation of input data. In previous work [1,9], we have developed a glass-box Constraint Logic Programming (CLP)-based approach to TCG for imperative object-oriented programs, which consists of mainly two phases: First, the imperative program is translated into an equivalent CLP program by means of partial evaluation [8]. Second, symbolic execution is performed on the CLP-translated program, controlled by a termination criterion (in this context also known as coverage criterion), relying on CLP's constraint solving facilities and its standard evaluation mechanism, with extensions for handling dynamic memory allocation.

Symbolic execution consists in executing a program with the contents of variables being symbolic formulas over the input arguments rather than concrete values [10]. The outcome is a set of equivalence classes of inputs, each of them consisting of the constraints that characterize a set of feasible concrete executions of a program that takes the same path. We refer as *test cases* to the set of such path constraints obtained by symbolically executing a program using a particular coverage criterion. Concrete instantiations of the test cases can be generated to obtain actual test inputs for the program, amenable for further validation by testing tools.

In this paper we propose a generic framework and methodology to *guide* symbolic execution in CLP-based TCG. It is well known that symbolic execution of large programs can become quickly impracticable due to the large number and the size of paths that need to be explored. This issue is considered a major challenge in the fields of symbolic execution and TCG [13]. Furthermore, a common limitation of TCG by symbolic execution is that it tends to produce an unnecessarily large number of test cases even for medium size programs.

Guided TCG is a heuristics that aims at steering symbolic execution, and thus TCG, towards specific program paths to generate more relevant test cases and filter

out less interesting ones with respect to a given coverage criterion. The main goal is to improve on scalability and efficiency by achieving a high degree of control over the coverage criterion and hence avoiding the exploration of unfeasible paths. This work extends and generalizes the methodology of [2], where TCG is guided with resource consumption information and resource policies.

The structure of the paper is as follows. Section 2 conveys the essentials of our CLP-based approach to TCG. Section 3 introduces the generic framework for guided TCG. Section 4 presents an instantiation of the generic framework based on trace-abstractions and targeting structural coverage criteria. Section 5 discusses a concrete and complementary strategy to avoid the exploration of unfeasible paths. Finally, Section 6 situates our work in the existing research space, sketches ongoing and future work and concludes.

2 CLP-based Test Case Generation

Our CLP-based approach to TCG for imperative object-oriented programs essentially consists of two phases: (1) the imperative program is translated into an equivalent CLP counterpart through partial evaluation; and (2) symbolic execution is performed on the CLP-translated program by relying on the CLP standard evaluation mechanisms. Details on the methodology can be found elsewhere [1, 8, 9].

2.1 CLP-Translated Programs

All features of the imperative object-oriented program under test are covered by its equivalent *executable* CLP-translated counterpart. Essentially, there exists a one-to-one correspondence between blocks in the control flow of the original program and rules in the CLP counterpart. Formally:

Definition 1 (CLP-Translated Program). *A CLP-Translated Program consists of a set of predicates, each of them defined by a set of mutually exclusive rules of the form* $m(In, Out, H_{in}, H_{out}, E, T) : -[\bar{G},]b_1, \ldots, b_n.$*, such that:*

(i) *In and Out are, resp., the (possibly empty) list of input and output arguments.*
(ii) H_{in} *and* H_{out} *are, resp., the input and output heaps.*
(iii) E *is an exception flag indicating whether the execution of* m *ends normally or with an uncaught exception.*
(iv) *If predicate* m *is defined by multiple rules, the guards in each one contain mutually exclusive conditions. We denote by* m^k *the* $k-th$ *rule defining* m*.*
(v) $\bar{G}$ *is a sequence of constraints that act as execution guards on the rule.*
(vi) $b_1, \ldots, b_n$ *is a sequence of instructions, including arithmetic operations, calls to other predicates and built-ins to operate on the heap.*
(vii) T *is the trace term for* m *of the form* $m(k, \langle T_{c_i}, \ldots, T_{c_m} \rangle)$*, where* k *is the index of the rule and* $T_{c_i}, \ldots, T_{c_m}$ *are free logic variables representing the trace terms associated to the subsequence* $c_i, \ldots, c_m$ *of calls to other predicates in* $b_1, \ldots, b_n$*.*

Notice that the trace term T is a not a cardinal element in the translated program, but rather a supplementary argument with a central role in this paper.

2.2 Symbolic Execution

CLP-translated programs are symbolically executed using the standard CLP execution mechanism with special support for the use of dynamic memory.

Definition 2 (Symbolic Execution). *Let M be a method, m be its corresponding predicate from its associated CLP-translated program P, and P' be the union of P and a set of built-in predicates to handle dynamic memory. The symbolic execution of m is the CLP derivation tree, denoted as $\mathcal{T}_m$, with root $m(In, Out, H_{in}, H_{out}, E, T)$ and initial constraint store $\theta = \{\}$ obtained using P'.*

2.3 Test Case Generation

As soon as a program contains loops or recursion, its symbolic execution tree is in general infinite. It is therefore essential to impose a *termination criterion*:

Definition 3 (Finite symbolic execution tree, test case, and TCG). *Let m be the corresponding predicate for a method M in a CLP-translated program P, and let $\mathcal{C}$ be a termination criterion.*

- *$\mathcal{T}_m^{\mathcal{C}}$ is the finite and possibly incomplete symbolic execution tree of m with root $m(In, Out, H_{in}, H_{out}, E, T)$ w.r.t. $\mathcal{C}$. Let B be the set of successful (terminating) branches of $\mathcal{T}_m^{\mathcal{C}}$.*
- *A test case for m w.r.t. $\mathcal{C}$ is a tuple $\langle \theta, T \rangle$, where θ and T are, resp., the constraint store and the trace term associated to one branch $b \in B$.*
- *TCG is the process of generating the set of test cases associated to all branches in B.*

Each *test case* produced by TCG represents a class of inputs that will follow the same execution path, and its trace is the sequence of rules applied along such path. In a subsequent stage, it is possible to produce actual values from the obtained constraint stores (e.g., by using labeling mechanisms in standard *clpfd* domains) therefore obtaining concrete and executable test cases. However, this is not an issue of this paper and we will comply with the above abstract definition of *test case*.

Example 1. Fig. 1 shows a Java program made up of three methods: `lcm` calculates the least common multiple of two integers, `gcd` calculates the greatest common divisor of two integers, and `abs` returns the absolute value of an integer. The right side of the figure shows the equivalent CLP-translated program. Observe that each Java method corresponds to a set of CLP rules, e.g., method `lcm` is translated into predicates `lcm`, `cont`, `check` and `div`. The translation preserves the control flow of the program and transforms iteration into recursion (e.g. method `gcd`). Note that the example has been chosen deliberately small and simple to ease comprehension. For readability, the actual CLP code has been simplified, e.g., input and output heap arguments are not shown, since they do not affect the computation. Our current implementation [3] supports full sequential Java.

Coverage Criteria. As shown in Def. 3, so far we have been interested in covering *all* feasible paths of the program under test w.r.t. a termination criterion. Now, our goal is to improve on efficiency by taking into account a *selection criterion* as well. First, let us define a *coverage criterion* as a pair of two components $\langle TC, SC \rangle$. TC

```
int lcm(int a,int b) {
    if (a < b) {
        int aux = a;
        a = b;
        b = aux;
    }
    int d = gcd(a,b);
    try {
        return abs(a*b)/d;
    } catch (Exception e) {
Ⓜ       return -1;}
}
int gcd(int a,int b) {
    int res;
    while (b != 0) {
        res = a%b;
        a = b; b = res;
    }
    return abs(a);
}
int abs(int a) {
    if (a >= 0)
Ⓚ       return a;
    else
        return -a;
}
```

```
lcm([A,B],[R],_,_,E,lcm(1,[T])) :-
        A #>= B, cont([A,B],[R],_,_,E,T).
lcm([A,B],[R],_,_,E,lcm(2,[T])) :-
        A #<= B, cont([B,A],[R],_,_,E,T).
cont([A,B],[R],_,_,E,cont(1,[T,V])) :-
        gcd([A,B],[G],_,_,E,T),
        check([A,B,G],[R],_,_,E,V).
check([A,B,G],[R],_,_,E,check(1,[T,V])) :-
        M #= A*B, abs([M],[S],_,_,E,T),
        div([S,G],[R],_,_,E,V).
check([A,B,G],[R],_,_,exc,check(2,[])).
div([A,B],[R],_,_,ok,div(1,[])) :-
        B #\= 0, R #= A/B.
div([A,0],[-1],_,_,exc_caught,div(2,[])).   Ⓜ
gcd([A,B],[D],_,_,E,gcd(1,[T])) :-
        loop([A,B],[D],_,_,E,T).
loop([A,0],[F],_,_,E,loop(1,[T])) :-
        abs([A],[F],_,_,E,T).
loop([A,B],[E],_,_,G,loop(2,[T])) :-
        B #\= 0, body([A,B],[E],_,_,G,T).
body([A,B],[R],_,_,E,body(1,[T])) :-
        B #\= 0, M #= A mod B,
        loop([B,M],[R],_,_,E,T).
body([A,0],[R],_,_,exc,body(2,[])).
abs([A],[A],_,_,ok,abs(1,[])) :- A #>= 0.   Ⓚ
abs([A],[-A],_,_,ok,abs(2,[])) :- A #< 0.
```

Fig. 1: Motivating Example: Java (left) and CLP-translated (right) programs.

is a *termination criterion* that ensures finiteness of symbolic execution. This can be done either based on execution steps or on loop iterations. In this paper, we adhere to loop-k, which limits to a threshold k the number of allowed loop iterations and/or recursive calls (of each concrete loop or recursive method). *SC* is a *selection criterion* that steers TCG to determine which paths of the symbolic execution tree will be explored. In other words, *SC* decides which test cases the TCG must produce. In the rest of the paper we focus on the following two coverage criteria:

- all-local-paths: It requires that all *local* execution paths within the method under test are exercised up to a loop-k limit. This has a potential interest in the context of unit testing, where each method must be tested in isolation.
- program-points(P): Given a set of program points P, it requires that all of them are exercised by at least one test case up to a loop-k limit. Intuitively, this criterion is the most appropriate choice for bug-detection and reachability verification purposes. A particular case of it is *statement coverage* (up to a limit), where all statements in a program or method must be exercised.

3 A Generic Framework for Guided TCG

The TCG framework as defined so far has been used in the context of coverage criteria only consisting of a termination criterion. In order to incorporate a selection

criterion, one can employ a post-processing phase where only the test cases that are sufficient to satisfy the selection criterion are selected by looking at their traces. This is however not an appropriate solution in general due to the exponential explosion of the paths that have to be explored in symbolic execution.

The goal in this paper is to design a methodology where the TCG process is driven towards satisfying the selection criterion, stressing to avoid as much as possible the generation of irrelevant and redundant paths. The key idea that allows us to guide the TCG process is to use the trace terms of our CLP-translated program as input arguments. Let us observe also that we could either supply fully or partially instantiated traces, the latter ones represented by including free logic variables within the trace terms. This allows guiding, completely or partially, the symbolic execution towards specific paths.

Definition 4 (trace-guided TCG). *Given a method M, a termination criterion TC, and a (possibly partial) trace π, trace-guided TCG generates the set of test cases with traces, denoted* $\mathsf{tgTCG}(M, TC, \pi)$*, obtained for all successful branches in $\mathcal{T}_m^{TC}$ with root m(Args$_{in}$,Args$_{out}$,H$_{in}$,H$_{out}$,E,π). We also define the* $\mathsf{firstOf\text{-}tgTCG}(M, TC, \pi)$ *to be the set corresponding to the leftmost successful branch in $\mathcal{T}_m^{TC}$.*

Observe that the TCG guided by one trace either generates: (a) exactly one test case if the trace is complete and corresponds to a feasible path, (b) none if it is unfeasible, or, (c) possibly several test cases if it is partial. In this case the traces of all test cases are instantiations of the partial trace.

Now, relying on trace-guided TCG and on the existence of a *trace generator* we define a generic scheme of *guided TCG.*

Definition 5 (guided TCG). *Given a method M; a coverage criterion $CC = \langle TC, SC \rangle$; and a trace generator TraceGen, that generates, on demand and one by one, (possibly partial) traces according to CC. Guided TCG is defined as the following algorithm:*

```
Input: M, and ⟨TC, SC⟩
TestCases = {}
while TraceGen has more traces and TestCases does not satisfy SC
   Ask TraceGen to generate a new trace in Trace
   TestCases = TestCases ∪ firstOf-tgTCG(M, TC, Trace)
Output: TestCases
```

The intuition is as follows: The trace generator generates a trace, possibly using for that SC, TC and the current $TestCases$. If the generated trace is feasible, then the first solution of its trace-guided TCG is added to the set of test cases. The process finishes either when SC is satisfied, or when the trace generator has already generated all traces up to TC. If the trace generator is complete (see below), this means that SC cannot be satisfied within the limit imposed by TC.

Example 2. Let us consider the TCG for method `lcm` with program-points for points μ and κ as selection criterion. Observe the correspondence of these program points in both the Java and CLP code of Fig. 1. Let us assume that the trace generator starts generating the following two traces:

```
lcm(1,[cont(1,[G,check(1,[A,div(2,[])])])])
lcm(2,[cont(1,[G,check(1,[A,div(2,[])])])])
```

The first iteration does not add any test case since the first trace is unfeasible. The second trace is feasible and with the generated test case the selection criterion is satisfied and therefore the process finishes. The obtained test case is shown in Example 7.

On Soundness, Completeness and Effectiveness: Intuitively, a concrete instantiation of the guided TCG scheme is *sound* if all test cases it generates satisfy the coverage criterion, and *complete* if it never reports that the coverage criterion is not satisfied when it is indeed satisfiable. *Effectiveness* is related to the number of iterations the algorithm performs. Those three features depend completely on the trace generator. We will refer then to trace generators as being sound, complete or effective. The intuition is that a trace generator is sound if every trace it generates satisfies the coverage criterion, and complete if it produces an over-approximation of the set of traces satisfying it. Effectiveness is related to the number of unfeasible traces it generates, the larger the number, the less effective the trace generator.

4 Trace Generators for Structural Coverage Criteria

This section presents a general approach for building sound, complete and effective trace generators for structural coverage criteria by means of program transformations. It then develops concrete instantiations for the all-local-paths and program-points coverage criteria and proposes concrete implementations in Prolog of the guided TCG scheme for both of them. Let us first define the notion of *trace-abstraction* of a program which will be the basis for defining our trace generators.

Definition 6 (trace-abstraction of a program). *Given a CLP-translated program with traces P, its trace-abstraction is obtained as follows: for every rule of P, (1) remove all atoms in the body of the rule except those corresponding to rule calls, and (2) remove all arguments from the head and from the surviving atoms of (1) except the last one (i.e., the trace term).*

```
lcm(lcm(1,[T])) :- cont(T).
lcm(lcm(2,[T])) :- cont(T).
cont(cont(1,[T,V])) :- gcd(T), check(V).
check(check(1,[T,V])) :- abs(T), div(V).
check(check(2,[])).
div(div(1,[])).
div(div(2,[])).
```

```
gcd(gcd(1,[T])) :- loop(T).
loop(loop(1,[T])) :- abs(T).
loop(loop(2,[T])) :- body(T).
body(body(1,[T])) :- loop(T).
body(body(2,[])).
abs(abs(1,[])).
abs(abs(2,[])).
```

Fig. 2: Trace-abstraction.

Example 3. Fig. 2 shows the trace-abstraction of our CLP-translated program of Fig. 1. Let us observe that it basically corresponds to its control-flow graph.

The trace-abstraction can be directly used as a trace-generator as follows: (1) Apply the termination criterion in order to ensure finiteness of the process. (2) Select, in a post-processing, those traces that satisfy the selection criterion. Such a trace generator produces on backtracking a superset of the set of traces of the program

satisfying the coverage criterion. Note that, this can be done as long as the criteria are structural. The obtained trace generator is by definition sound and complete. However, it can be very ineffective and inefficient due to the large number of unfeasible and/or unnecessary traces that it can generate. In the following, we propose two concrete, and more effective, instantiations for the all-local-paths and program-points coverage criteria. In both cases, this is done by taking advantage of the notion of partial traces and the implicit information on the concrete coverage criteria.

4.1 An Instantiation for the all-local-paths Coverage Criterion

Let us start from the trace-abstraction program and apply a syntactic program slicing which removes from it the rules that do not belong to the considered method.

Definition 7 (slicing for all-local-paths coverage criterion). *Given a trace-abstraction program P and an entry method M:*

1. *Remove from P all rules that do not belong to method M.*
2. *In the bodies of remaining rules, remove all calls to rules which are not in P.*

The obtained sliced trace-abstraction, together with the termination criterion, can be used as a trace generator for the all-local-paths criterion for a method. The generated traces will have free variables in those trace arguments that correspond to the execution of other methods, if any.

```
lcm(lcm(1,[T])) :- cont(T).
lcm(lcm(2,[T])) :- cont(T).
cont(cont(1,[G,T])) :- check(T).
check(check(1,[A,T])) :- div(T).
check(check(2,[])).
div(div(1,[])).
div(div(2,[])).
```

```
lcm(1,[cont(1,[G,check(1,[A,div(1,[])])])])
lcm(1,[cont(1,[G,check(1,[A,div(2,[])])])])
lcm(1,[cont(1,[G,check(2,[])])])
lcm(2,[cont(1,[G,check(1,[A,div(1,[])])])])
lcm(2,[cont(1,[G,check(1,[A,div(2,[])])])])
lcm(2,[cont(1,[G,check(2,[])])])
```

Fig. 3: Slicing of method `lcm` for all-local-paths criterion.

Example 4. Fig. 3 shows on the left the sliced trace-abstraction for method `lcm`. On the right is the finite set of traces that is obtained from such trace-abstraction for any loop-K termination criterion. Observe that the free variables `G`, resp. `A`, correspond to the sliced away calls to methods `gcd` and `abs`.

Let us define the predicates: `computeSlicedProgram(M)`, that computes the sliced trace-abstraction for method M as in Def. 7; `generateTrace(M,TC,Trace)`, that returns in its third argument, on backtracking, all partial traces computed using such sliced trace-abstraction, limited by the termination criterion `TC`; and `traceGuidedTCG(M,TC,Trace,TestCase)`, which computes on backtracking the set tgTCG(M,Trace,TC) in Def. 4, failing if the set is empty, and instantiating on success `TestCase` and `Trace` (in case it was partial). The guided TCG scheme in Def. 5,

instantiated for the all-local-paths criterion, can be implemented in Prolog as follows:

```
(1) guidedTCG(M,TC) :-
(2)     computeSlicedProgram(M),
(3)     generateTrace(M,TC,Trace),
(4)     once(traceGuidedTCG(M,Trace,TC,TestCase)),
(5)     assert(testCase(M,TestCase,Trace)),
(6)     fail.
(7) guidedTCG(_,_).
```

Intuitively, given a (possibly partial) trace generated in line `(3)`, if the call in line `(4)` fails, then the next trace is tried. Otherwise, the generated test case is asserted with its corresponding trace which is now fully instantiated (in case it was partial). The process finishes when `generateTrace/3` has computed all traces, in which case it fails, making the program exiting through line `(7)`.

Example 5. The following test cases are obtained for the all-local-paths criterion for method `lcm`:

Constraint store	*Trace*
{A>=B}	lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), check(1,[abs(1,[]),div(1,[])])])])
{A=B=0,Out=-1}	lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), check(1,[abs(1,[]),div(2,[])])])])
{B>A}	lcm(2,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), check(1,[abs(1,[]),div(1,[])])])])

This set of 3 test cases achieves full code and path coverage on method `lcm` and is thus a perfect choice in the context of unit-testing. In contrast, the original, non-guided, TCG scheme with loop-2 as termination criterion produces 9 test cases.

4.2 An Instantiation for the program-points Coverage Criterion

Let us start by considering a simplified version of the program-points criterion so that only one program point is allowed, denoted as program-point. Starting again from the trace-abstraction program, we propose a syntactic program slicing that filters away the part of the program which is not relevant for the paths that do not visit the program point of interest.

Definition 8 (slicing for program-point coverage criterion). *Given a trace-abstraction program P, a program point of interest pp, and an entry method M, the sliced program P' is computed as follows:*

1. *Initialize P' to be the empty program, and a set of clauses C with the clause corresponding to pp.*
2. *For each c in C which is not the clause for M, add to C all clauses in P whose body has a call to the predicate of clause c, and iterate until the set C stabilizes.*
3. *Add to P' all clauses in C.*
4. *Remove all calls to rules which are not in P' from the bodies of the rules in P'.*

The obtained sliced program, together with the termination criterion, can be used as a trace generator for the program-point criterion. The generated traces can have free variables representing parts of the execution which are not related (syntactically) to the paths visiting the program point of interest.

Example 6. Fig. 4 shows on the left the sliced trace-abstraction program (using Def. 8) for method `lcm` and program point ⓤ from Fig. 1, i.e. the `return` statement within the `catch` block. On the right of the same figure, the traces obtained from such slicing using loop-2 as termination criterion.

```
lcm(lcm(1,[T])) :- cont(T).
lcm(lcm(2,[T])) :- cont(T).
cont(cont(1,[G,T])) :- check(T).
check(check(1,[A,T])) :- div(T).
div(div(2,[])).
```

```
lcm(1,[cont(1,[G,check(1,[A,div(2,[])])])])
lcm(2,[cont(1,[G,check(1,[A,div(2,[])])])])
```

Fig. 4: Slicing for program-point coverage criterion with *pp*=ⓤ from Fig. 1.

Consider again predicates `computeSlicedProgram/2`, `generateTrace/4` and `traceGuidedTCG/4` with the same meaning as in Section 4.1, but being the first two now based on Def. 8 and extended with the program-point argument `PP`. The guided TCG scheme in Def. 5, instantiated for the program-points criterion, can be implemented in Prolog as follows:

```
(1) guidedTCG(M,[],TC) :- !.
(2) guidedTCG(M,[PP|PPs],TC) :-
(3)     computeSlicedProgram(M,PP),
(4)     generateTrace(M,PP,TC,Trace),
(5)     once(traceGuidedTCG(M,Trace,TC,TestCase)), !,
(6)     assert(testCase(M,TestCase,Trace)),
(7)     removeCoveredPoints(PPs,Trace,PPs'),
(8)     guidedTCG(M,PPs',TC).
(9) guidedTCG(M,[PP|_],TC) :- .
```

Intuitively, given the first remaining program point of interest `PP` (line `(2)`), a trace generator for it is computed and used to obtain a (possibly partial) trace that exercises it (lines `(3)` and `(4)`). Then, if the call in line `(5)` fails, then another trace for `PP` is requested. If there are not more traces (i.e. line `(4)` fails) the process finishes through line `(9)` reporting that `PP` is not reachable within the imposed `TC`. If the call in line `(5)` succeeds, the generated test case is asserted with its corresponding trace (now fully instantiated in case it was partial), the remaining program points which are now covered by `Trace` are removed obtaining `PPs'` (line `(7)`), and the process continues with `PPs'`. Note that, a new sliced program will be computed for the first program point in `PPs'`. When all program points have been covered the process finishes through line `(1)`.

The above implementation is valid for the general case of program-points criteria with any finite set size. The trace generator, instead, has been deliberately defined for just one program point since this way the program slicing can be more aggressive, hence saving the generation of unfeasible traces.

Example 7. The following test case is obtained for the program-points criterion for method `lcm` and program points ⓤ and ⓚ. This particular case illustrates specially well how guided TCG can reduce the number of produced test cases through

adequate control of the selection criterion.

Constraint store	*Trace*
{A=B=0,Out=-1}	lcm(1,[cont(1,[gcd(1,[loop(1,[abs(1,[])])]), check(1,[abs(1,[]),div(2,[])])])])

5 Trace-Abstraction Refinement

Unfortunately the trace-abstraction defined in Def. 6 may still often compromise effectiveness, since the number of unfeasible paths to be explored can still be too large. To tackle this problem, we propose a heuristics that aims to refine the trace-abstraction with information taken from the original program that will help reduce the number of unfeasible paths at symbolic execution time. The goal is to reach a balanced level of refinement in between the original program (full refinement) and the trace-abstraction (empty refinement). Intuitively, the more information we include, the less unfeasible paths symbolic execution explores, but the more costly it becomes.

This section presents a quite effective refinement algorithm that consists of two steps. First, in a fixpoint analysis we approximate the instantiation mode of the variables in each predicate of the CLP-translated program. In other words, we infer which variables will be constrained or assigned a concrete value at symbolic execution time. In a second step, by program transformation means, the trace-abstraction is enriched with clause arguments corresponding to such inferred variables, and with those goals in which they are involved.

5.1 Approximating instantiation modes

We develop a static analysis, similar to [6, 7], to soundly approximate the instantiation mode of the input argument variables in the program at symbolic execution time. The analysis is implemented as a fixpoint computation over the simple abstract domain $\{static, dynamic\}$. Namely, *dynamic* means that nothing was inferred about a variable and it will therefore remain a free unconstrained variable during symbolic execution; and *static* means that the variable will unify with a concrete value or will be constrained during symbolic execution. The analysis's result is a set of assertions in the form $\langle P, \mathcal{V}\rangle$ where P is a predicate name and $\mathcal{V}$ is the set of variables in P, each associated with an abstract value from the domain.

This analysis receives as input a CLP-translated program and a set of initial entries (predicate names). An event queue $\mathcal{Q}$ is initialized with this set of initial entries. The algorithm starts to process the events of $\mathcal{Q}$ until no more events are scheduled. In each iteration, an event p is removed from $\mathcal{Q}$ and processed as follows: Retrieve previously stored information $\psi \equiv \langle p, \mathcal{V}\rangle$ if any exists; else set $\psi \equiv \langle p, \emptyset\rangle$. For each rule r defining p, a new $\mathcal{V}_r$ is obtained by evaluating the body of r. The joint operation on the underlying abstract domain is performed to obtain $\mathcal{V}' \Leftarrow joint(\mathcal{V}, \mathcal{V}_r)$. If $\mathcal{V} \not\equiv \mathcal{V}'$ then set $\mathcal{V} \Leftarrow \mathcal{V}'$ and reschedule every predicate that calls p; else, if $\psi' \equiv \psi$ there is no need to recompute the calling predicates and the algorithm continues. That will ensure backward propagation of approximated instantiation modes. To propagate forward, the evaluation of r will schedule one event per call within its body. The process continues until a fixpoint is reached.

5.2 Constructing the trace-abstraction refinement

This is a syntactic program transformation stage of the refinement. It takes as input the original CLP-program and the instantiation information inferred in the first stage. For each rule r of a predicate p in the program, the algorithm retrieves $\langle p, \mathcal{V} \rangle$. We denote $\mathcal{V}_s$ the projection of all variables in $\mathcal{V}$ whose inferred abstract value is *static*. First, we add to the trace-abstraction all input arguments In_α such that $In_\alpha \subset \mathcal{V}_s$. Then, the body $b_1, \ldots, b_n$ of the clause is traversed, to include in the trace-abstraction: 1) all guards and arithmetic operations b_i such that $varset(b_i) \subset \mathcal{V}_s$, and 2) all calls to other predicates, with the corresponding projection of *static* abstract-valued arguments.

Example 8. Consider the Java example of Fig. 5 (left side). First, let us focus on function `power`. It implements a exponentiation algorithm for positive integer exponents. Its CLP counterpart is shown at the right of the figure. The instantiation modes inferred by the first stage of our algorithm is presented at the right-bottom part of the figure. One can observe that variable `B` (the base of the exponentiation) remains *dynamic* all along the program, because it is never assigned any concrete value nor constrained by any guard. On the other hand, variable `E`'s final abstract value is *static*, since the analysis of rules `if` and `loop`, where it is constrained by `0` and the also *static* variable `I`. The following is the refined trace-abstraction that our algorithm constructs:

```
power([E],power(1,[T])) :- if([E],T).
if(E],if(1,[])) :- E #< 0.
if([E],if(2,[T])) :- E #>= 0), loop([E,1,1],T).
loop([E,I],loop(1,[])) :- I #> E.
loop([E,I],loop(2,[T])) :- I #=< E, Ip #= I+1, loop([E,Ip],T).
```

To illustrate how the trace-abstraction refinement can improve on effectiveness of the guided TCG, let us observe method `arraypower`. It iterates over all the elements of an input array `a` and calls function `power` to update all even positions of the array by raising their values to the power of the integer input argument `e`. We report on the following TCG performance results for this example and a coverage criterion $\langle$loop-2, $\{\}\rangle$:

- Standard non-guided TCG of this example generates 11 test cases.
- Trace-abstraction guided TCG with the empty refinement generates 497 possibly (un)feasible traces to be tested.
- Trace-abstraction guided TCG with our trace-abstraction refinement reduces the number of possibly (un)feasible traces to be tested to 161.

These preliminary, yet promising, results unveil the potential integration of the trace-abstraction refinement algorithm presented in this section with the general guided TCG framework developed along this paper. In particular, the refinement is in principle complementary to the slicings presented in Section 4 without any modification. Unfortunately, the slicings could produce a loss of important information added by the refinement. This could be however improved by means of simple syntactic analyses on the sliced parts of the program. A deeper study of these issues remains as future work.

```
void arraypower(int a[],int e) {
    int i=0;
    int n=a.length;
    for (i=0; i<n; i++)
        if (i%2==0)
            a[i]=power(a[i],e);
}
int power(int b, int e) {
    if (e >= 0) {
        int pow = 1;
        while (i <= e) {
            pow *= b;
            i++;
        }
        return pow;
    } else return -1;
}
```

```
power([B,E],[R],_,_,F,power(1,[T])) :-
     if([B,E],[R],_,_,F,T).
if([B,E],[-1],_,_,F,if(1,[])) :-
     E #< 0.
if([B,E],[R],_,_,F,if(2,[T])) :-
     E #>= 0),
     loop([B,E,1,1],[R],_,_,F,T).
loop([B,E,I,P],[P],_,_,ok,loop(1,[])) :-
     I #> E.
loop([B,E,I,P],[R],_,_,F,loop(2,[T])) :-
     I #=< E, Pp #= P*B, Ip #= I+1,
     loop([B,E,Ip,Pp],[R],_,_,F,T).
```

Inferred instantiation modes:
⟨power, {B= *dynamic*,E= *static*}⟩
⟨if, {B= *dynamic*,E= *static*}⟩
⟨loop, {B= *dynamic*,E= *static*,
I= *static*,P= *dynamic*}⟩

Fig. 5: Trace-abstraction Refinement.

6 Conclusions

Previous work also use abstractions to guide symbolic execution and TCG by several means and for different purposes. One of the most relevant to ours is [5], where predicate abstraction, model checking and SAT-solving are combined to produce abstractions and generate test cases for C programs, with good code coverage, but depending highly on an initial set of predicates to avoid infeasible program paths. Rugta *et al.* [14] also proposes to use an abstraction of the program in order to guide symbolic execution and prune the execution tree as a way to scale up. Their abstraction is an under-approximation which tries to reduce the number of test cases that are generated in the context of concurrent programming, where the state explosion is in general more problematic.

The work presented in this paper introduces our framework for *guided TCG,* whose purpose is to guide the process of test generation towards achieving interesting structural coverage. The implicit aim is to improve scalability of TCG by guiding symbolic execution with abstractions, which is one of the most used techniques for this matter. Fundamentally, abstraction aims to reduce large data domains of a program to smaller domains (see [12]). Another scaling technique which is closely related to abstraction is path merging [4,11], which consists in defining points where the merging of symbolic execution should occur. This work complements with compositional reasoning in symbolic execution, which has also been assessed in our TCG framework.

Guided TCG can serve different purposes. It can be used to discover bugs in a program, to analyze reachability of certain parts of a program, to lead symbolic execution to stress more interesting parts of the program, etc. This paper targets unit testing, a widely used software engineering methodology, where units of code (e.g. methods) are tested in isolation to validate their correctness. This represents one step towards fully supporting both unit and integration testing, a different methodology,

where all the pieces of a systems must be tested as a single unit. In general, the later a defect is detected, the higher the cost of remediation. Our approach contributes to allow the testing of a program begin at the inception of the development process and extend through to deployment.

We are currently working on realizing the framework presented in this paper as an usable tool, to be integrated and made available shortly as part of the PET system [3].

Acknowledgments. This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2008-05624 and PRI-AIBDE-2011-0900 projects, by UCM-BSCH-GR35/10-A-910502 grant and by the Madrid Regional Government under the S2009TIC-1465 *PROMETIDOS-CM* project.

References

1. E. Albert, M. Gómez-Zamalloa, and G. Puebla. Test Data Generation of Bytecode by CLP Partial Evaluation. In *Proc. of LOPSTR'08*, number 5438 in LNCS. Springer-Verlag, March 2009.
2. E. Albert, M. Gómez-Zamalloa, and J.M. Rojas. Resource-driven CLP-based test case generation. In *Proc. of LOPSTR'11*, LNCS. Springer-Verlag, 2012. To appear.
3. E. Albert, I. Cabañas, A. Flores-Montoya, M. Gómez-Zamalloa, and S. Gutiérrez. jPET: an Automatic Test-Case Generator for Java. In *18th Working Conference on Reverse Engineering (WCRE 2011)*, pages 441–442. IEEE Computer Society, October 2011.
4. T. Arons, E. Elster, S. Ozer, J. Shalev, and E. Singerman. Efficient symbolic simulation of low level software. In *DATE*, pages 825–830. IEEE, 2008.
5. T. Ball. Abstraction-guided test generation: A case study. Technical Report MSR-TR-2003-86, Microsoft Research, 2003.
6. S.-J. Craig, J. P. Gallagher, M. Leuschel, and K. S. Henriksen. Fully Automatic Binding-Time Analysis for Prolog. In *LOPSTR*, volume 3573 of *LNCS*. Springer, 2004.
7. S. K. Debray. Static inference of modes and data dependencies in logic programs. *ACM Trans. Program. Lang. Syst.*, 11(3):418–450, July 1989.
8. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Decompilation of Java Bytecode to Prolog by Partial Evaluation. *Information and Software Technology*, 51(10):1409–1427, October 2009.
9. M. Gómez-Zamalloa, E. Albert, and G. Puebla. Test Case Generation for Object-Oriented Imperative Languages in CLP. *Theory and Practice of Logic Programming, ICLP'10 Special Issue*, 10 (4–6), 2010.
10. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
11. A. Kölbl and C. Pixley. Constructing efficient formal models from high-level descriptions using symbolic simulation. *International Journal of Parallel Programming*, 33(6):645–666, 2005.
12. Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proc. 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 98–112, Genova, Italy, April 2001. Springer-Verlag.
13. C. S. Păsăreanu and W. Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, 2009.
14. N. Rungta, E.G. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *SPIN*, volume 5578 of *LNCS*. Springer, 2009.

An Extension of π-Calculus with Real-Time and its Realization in Logic Programming

Neda Saeedloei[1,2] and Gopal Gupta[2]

[1] INRIA Paris-Rocquencourt, France,
neda.saeedloei@inria.fr,
[2] University of Texas at Dallas, TX, USA,
gupta@utdallas.edu

Abstract. We extend π-calculus with real-time by adding clocks and assigning time-stamps to actions. The resulting formalism, timed π-calculus, provides a simple and novel way to annotate transition rules of π-calculus with timing constraints. We develop an operational semantics for the proposed language and a logic based implementation of this operational semantics. Our implementation is based on *Horn logical semantics* of programming languages and partial evaluation. The partial evaluated code obtained, is a coroutined coinductive constraint logic program which can be used for verifying real-time systems expressed as time π-calculus processes.

1 Introduction

The π-calculus was introduced by Milner et al. [14] with the aim of modeling concurrent/mobile processes. The π-calculus provides a conceptual framework for describing systems whose components interact with each other. It contains an algebraic language for describing processes in terms of the communication actions they can perform. Theoretically, the π-calculus can model mobility, concurrency and message exchange between processes as well as infinite computation (through the infinite replication operator '!').

In many cases, processes run on controllers that control physical devices; therefore, they have to deal with physical quantities such as time, distance, pressure, acceleration, etc. Examples include communicating controller systems in cars (Anti-lock Brake System, Cruise Controllers, Collision Avoidance, etc.), automated manufacturing, smart homes, etc. Properties of such systems, which are termed cyber-physical systems (CPS) [11, 6], cannot be fully expressed within π-calculus. In a real-time/cyber-physical system the correctness of the system's behavior depends not only on the tasks that the system is designed to perform, but also on the time instants at which these tasks are performed. This class of systems includes many safety critical systems such as those found in robotics and flight control. While π-calculus can handle mobility and concurrency, it is not equipped to model real-time systems or CPS and support reasoning about their behavior related to time and other physical quantities. We extend π-calculus with real time so that these systems can be modeled and reasoned about.

Several extensions of π-calculus with time have been proposed [2, 12, 3, 4]; all these approaches discretize time rather than represent it faithfully as a continuous quantity. Discretizing means that time is represented through finite time intervals. As a result, *infinitesimally small time intervals cannot be represented or reasoned about in these approaches*. In practical real-time systems, e.g., a nuclear reactor, two or more events *can* occur within an infinitesimally small interval. In our approach for extending π-calculus with time, time is faithfully modeled as a continuous quantity.

We consider the extension of π-calculus with continuous time by adding finitely many real-valued clocks and assigning time-stamps to actions. The resulting formalism can be used for describing concurrent, mobile, real-time systems and CPS and reasoning about their behaviors. For simplicity, the behavior of a real-time system is understood as a sequence (finite or infinite) of timed events, not states. We develop an executable operational semantics of timed π-calculus in logic programming (LP) in which concurrency is modeled by *coroutining*, and (rational) infinite computation in presence of constraints by *coinductive constraint logic programming over reals* (Co-CLP) [18]. The executable operational semantics directly leads to an implementation of the timed π-calculus.

The work of Gupta et al. [21, 22] showed how Horn logical semantics and partial evaluation can be used to generate provably correct code. From the Horn logical semantic description of the language $\mathcal{L}$, one immediately obtains an interpreter of language $\mathcal{L}$. The semantics can be viewed dually as operational or denotational. Given a program $\mathcal{P}$ written in language $\mathcal{L}$, the interpreter obtained for $\mathcal{L}$ can be used to execute the program. Moreover, given a partial evaluator for pure Prolog, the interpreter can be partially evaluated w.r.t. program $\mathcal{P}$ to obtain provably correct compiled code for $\mathcal{P}$ [21, 22].

In this paper, we apply the approach of [21, 22] to our timed π-calculus to obtain an implementation of this language. First, we express the syntax of timed π-calculus in the *Definite Clause Grammar* (DCG) notation. This syntax specification trivially and naturally yields an *executable parser* for timed π-calculus. This parser can be used to parse timed π-calculus expressions and obtain their parse trees. Next, we express the semantic algebra and valuation functions of timed π-calculus in logic programming. The semantics and syntax specifications loaded in a logic programming system can be executed which leads to an interpreter for timed π-calculus. Finally, given a timed π-calculus process P, we partially evaluate the interpreter w.r.t. P to obtain a coroutined coinductive constraint logic program Q. The program Q which is the implementation of P, when loaded to a coinductive constraint logic program system can be executed and used for verifying properties of P. We illustrate our approach by applying it to the *Generalized Railroad Crossing* (GRC) problem of Lynch and Heitmeyer [7], modeled in timed π-calculus.

Thus, our contributions are twofold: (i) we extend the π-calculus with real-time clocks: in contrast to other extensions, in our work the notion of time and clocks is adopted directly from the well-understood formalism of timed automata; (ii) we show how an executable operational semantics of timed π-calculus can be

elegantly realized through coinductive constraint logic programming extended with coroutining. The executable semantics faithfully captures real-time behaviors and allows us to prove behavioral and timing properties of a system modeled in timed π-calculus.

2 Timed π-Calculus

2.1 Design Decisions

We define our timed π-calculus as an extension of the original π-calculus [14] with (local) clocks, clock operations and time-stamps. As in π-calculus, timed π-calculus processes use names (including clock names) to interact, and pass names to one another. These processes are identical to processes in π-calculus except that they have access to clocks which they can manipulate.

We assume an infinite set $\mathcal{N}$ of names (channel names and names passing through channels), an infinite set Γ of clock names (disjoint from $\mathcal{N}$) and an infinite set Θ of variables representing time-stamps (disjoint from $\mathcal{N}$ and Γ). When a process outputs a name through a channel, it also sends the time-stamp of the name and the clock[3] that is used to generate the time-stamp. Inspired by the notion of name transmission in π-calculus, we can treat time-stamps and clocks just as other names and transmit them through/with channels. Just as channel transmission results in dynamic configuration of processes, clock and time-stamp transmission can result in dynamic temporal behavior of processes. Thus, messages are represented by triples of the form $\langle m, t_m, c\rangle$, where m is a name in $\mathcal{N}$, t_m is the time-stamp on m, and c is the clock that is used to generate t_m. It is important for the process to send its clock that is used to generate the time-stamp of the name, because the time-stamp of the incoming name in conjunction with the clock received is used by the receiving process to reason about timing requirements of the system and delays.

In our timed π-calculus all the clocks are local clocks; however, their scope is changed as they are sent among processes. This will become clear when we explain how clock passing is performed in Section 2.4. Note that all the clocks advance at the same rate. A clock can be set to zero simultaneously with any transition (transitions are defined formally in Section 2.3). At any instant, the reading of a clock is equal to the time that has elapsed since the last time the clock was reset. We only consider non-Zeno behaviors, that is, only a finite number of transitions can happen within a finite amount of time. We consider two types of clock operations: resetting a clock and checking satisfiability of a clock constraint. Resetting a clock, denoted by γ, is used to remember the time at which a particular action in the system has taken place. A clock constraint, denoted by δ, indicates a timing constraint between some actions that appear in the system. δ and γ are defined by the following syntactic rules, in which c and $c_i, 1 \leq i \leq n$, are clock names, r is a constant in $\mathbf{R}^+$, t is a time-stamp variable

[3] Later when writing the operational semantics of timed π-calculus, we assume that all processes have access to the same wall clock. However, in reality it might not be the case; therefore, extra information has to be sent in the form of clocks.

and $\sim \in \{<, >, \leq, \geq, =\}$. ϵ represents an empty clock constraint or clock reset.

$$\delta ::= (c \sim r)\delta \mid (c - t \sim r)\delta \mid (t - c \sim r)\delta \mid \epsilon$$
$$\gamma ::= (c_1 := 0) \ldots (c_n := 0) \mid \epsilon$$

There are two ways to measure the passing of time while checking for a clock constraint. It can be measured and reasoned about against (i) the last time a clock was reset: e.g., a constraint $(c < 2)$ on sending m indicates that m must be sent out within two units of time since the clock c was reset, or (ii) the last time a clock c was reset in conjunction with a time-stamp t of an arriving or a sent name. Note that in this case, the time-stamp t must be generated by clock c. For instance, suppose that a process P sends two consecutive names that are two units of time apart; if the time-stamp of the first name, generated by clock c is t_1, then the expression $c - t_1 = 2$ can be used to express this constraint.

For a process P, we define $c(P)$ to be the set of clock names in P. For every two processes P and Q we assume $c(P) \cap c(Q) = \emptyset$. A *clock interpretation* $\mathcal{I}$ for a set Γ of clocks is a mapping from Γ to $\mathbf{R}^+$. It assigns a real value to each clock in Γ. A clock interpretation $\mathcal{I}$ for Γ satisfies a clock constraint δ over Γ iff the expression obtained by applying $\mathcal{I}$ to δ evaluates to true. For $t \in \mathbf{R}^+$, $I + t$ denotes the clock interpretation which maps every clock c to the value $I(c) + t$. For $\gamma \subseteq \Gamma$, $[\gamma \mapsto t]I$ denotes the clock interpretation for Γ which assigns t to each $c \in \gamma$, and agrees with I over the rest of the clocks.

2.2 Syntax

The set of timed π-calculus processes is defined by the following syntactic rules in which, P, P', M and M' range over processes, x, y and z range over names in $\mathcal{N}$, c and d range over clock names in Γ, and t_y represents a time-stamp.

$$M ::= \delta\gamma\bar{x}\langle y, t_y, c\rangle.P \mid \delta\gamma x(\langle y, t_y, c\rangle).P \mid \delta\gamma\tau.P \mid 0 \mid M + M'$$
$$P ::= M \mid (P \mid P') \mid !P \mid (z)\ P \mid (c)\ P \mid [x = y]\, P \mid [c = d]\, P$$

The expression $\delta\gamma\bar{x}\langle y, t_y, c\rangle.P$ represents a process that sends the name y, time-stamp of y, t_y, and its clock c via the channel x, and evolves to P, provided that the clock constraint δ is satisfied by the current value of clocks at the time of transition γ specifies the clocks to be reset with this transition. For example the expression $(c < 2)(c := 0)\bar{x}\langle y, t_y, c\rangle.P$ indicates that $\langle y, t_y, c\rangle$ must be sent out on channel x within two units of time since the clock c was last reset, and moreover, the clock c is reset when it is sent.

The expression $\delta\gamma x(\langle y, t_y, c\rangle).P$ stands for a process which is waiting for a message on channel x. When a message arrives, the process will behave like $P\{z/y, t_z/t_y, d/c\}$ (substitution is formally defined in Definition 2) where z is the name received; t_z is the time-stamp of z; and d is the clock of the sending process that is used to generate t_z. The time-stamp t_z must satisfy the clock constraint expressed by δ; γ specifies the clocks to be reset with the transition.

The expression $\delta\gamma\tau.P$ stands for a process that takes an internal action and evolves to P, and in doing so resets the clocks specified by γ, if the clock constraint δ is satisfied.

In each of three processes explained above, if the clock constraint δ is not satisfied by the value of clocks at the time of transition, then, the process becomes inactive. An inactive process, represented by 0, is a process that does nothing. The operators + and | are used for nondeterministic *choice* and *composition* of processes, just as in π-calculus [14]. The *replication* $!P$, represents an infinite composition $P \mid P \mid \ldots$, just as in π-calculus [14]. The *restriction* $(z)P, z \in \mathcal{N}$, behaves as P with z local to P. Therefore, z cannot be used as a channel over which to communicate with other processes or the environment. $(c)P, c \in \Gamma$, behaves as P with c being a local clock in P. $[x = y]\, P$ evolves to P if x and y are the same name; otherwise, it becomes inactive. $[c = d]\, P$ evolves to P if c and d are the same clock name; otherwise, it becomes inactive.

Example 1. The timed π-calculus expression $x(\langle m, t_m, c\rangle).(c - t_m < 5)\bar{y}\langle n, t_n, c\rangle$ represents a process that is waiting for a message on channel x. The process upon receiving a name m with time-stamp t_m and its accompanying clock c on channel x, sends a name n with time-stamp t_n on channel y with the delay of at most 5 units of time since the time-stamp of m. The process will use the clock c to choose a time t_n on c such that $c - t_m < 5$.

In a process of the form $\delta\gamma x(\langle y, t, c\rangle).P$ the occurrences of y, t and c are binding occurrences, and the scope of the occurrences is P. In $(n)P, n \in \mathcal{N} \cup \Gamma$ the occurrence of n is a binding occurrence, and the scope of the occurrence is P.

Definition 1. *[14] An occurrence of n in a process is* free *if it does not lie within the scope of a binding occurrence of n. An occurrence of a name in a process is* bound *if it is not free. The set of bound names of P, $bn(P)$, contains all names which occur bound in P. The set of names occurring free in P is denoted $fn(P)$. We write $n(P)$ for the set $fn(P) \cup bn(P)$ of names of P.*

Intuitively, the free (non-clock) names of a process, represent its (public) links to other processes. For instance, if processes P and Q share the same free name x, then, the channel x is shared between these two processes.

Example 2. Let $P = x(\langle y, t, c\rangle).P'$ and $Q = (d)(d > 1)(d < 5)x\langle z, t', d\rangle.Q'$. Then, $fn(P) = \{x\}, bn(P) = \{y, t, c\}$, $fn(Q) = \{x, z, t'\}$, and $bn(Q) = \{d\}$. Since x appears as a free name in both P and Q, x is a channel that is shared between P and Q. This behavior can be represented for example in the parallel composition of P and Q: $(P \mid Q)$. Assume further that x is restricted, then this new behavior is represented by $(x)(P \mid Q)$.

Definition 2. *[14] A substitution is a function θ from a set of names $\mathcal{N}$ to $\mathcal{N}$. If $x_i\theta = y_i$ for all i with $1 \leq i \leq n$ (and $x\theta = x$ for all other names x), we write $\{y_1/x_1, \ldots, y_n/x_n\}$ for θ.*

The effect of applying a substitution θ to a process P is to replace each free occurrence of each name x in P by $x\theta$, with change of bound names to avoid name capture (to preserve the distinction of bound names from the free names). Substitution for clock names and time-stamps can be defined similarly.

2.3 Operational Semantics

A transition in the timed π-calculus is of the form $P_{(I,w)} \xrightarrow{\langle\delta,\alpha,\gamma\rangle} P'_{(I',w')}, w' > w$. This transition means that if δ is satisfied by the current interpretation I, P evolves into P' at time w', and in doing so performs the action α and resets the clocks specified by γ. After this transition the new interpretation I' is $[\gamma \mapsto 0](I + w' - w)$ and the time w' at which the action is performed is recorded. Note that w is the time of the last action before this transition. With abuse of notation, we have used γ as a set of clocks to be reset. We call the triple $\langle\delta, \alpha, \gamma\rangle$ a timed action, in which action α is defined by the following syntactic rule:

$$\alpha ::= \bar{x}\langle y, t, (c)\rangle \mid x(\langle y, t, c\rangle) \mid \tau \mid \bar{x}\langle(y), t, (c)\rangle$$

The first action is the *output action* $\bar{x}\langle y, t, (c)\rangle$. This action is used for sending a name y, time-stamp of y, t, and the clock that is used to generate t via channel x. The process that gives rise to this action can be of the form $(c)\bar{x}\langle y, t, c\rangle.P$. In this action x and y and t are free and c is bound.

The second action is the *input action* $x(\langle y, t, c\rangle)$. This action is used for receiving any name z with its time-stamp t_z, and a clock d via x. y, t and c are place holders in the receiving process for values that will be received as inputs. In this action x is free, while y, t and c are bound names.

The third action is the *silent action* τ, which is used to express performing an internal action. Silent actions can naturally arise from processes of the form $\tau.P$, or from communications within a process (e.g., rule COM in Table 1).

The last action is the *bound output action*. The expression $\bar{x}\langle(y), t, (c)\rangle$ is used by a process P for sending a private name y (y is bound in P). Bound output actions arise from output actions which carry (non-clock) names out of their scope. The process that gives rise to this action can be of the form: $(c)(y)\bar{x}\langle y, t, c\rangle.P$. In this action x and t are free, while y and c are bound.

Intuitively, a system specified by the set of timed π-calculus processes starts with an empty interpretation. As a process starts its execution, the set of its clocks are added to the clock interpretation. All the clocks of a process are mapped to 0 and $w = 0$, initially. As time advances the value of all clocks advances, reflecting the elapsed time. The behavior of a system can be understood as a finite or infinite sequence of timed events of the form $(\alpha_1, w_1), (\alpha_2, w_2), \ldots$, in which α_i is an action and w_i is a wall clock time at which α_i takes place.

The operational semantics of timed π-calculus, represented in Table 1, is expressed as a set of transition rules which are labeled by timed actions. We use $fn(\alpha)$ for set of free names of α, $bn(\alpha)$ for set of bound names of α, and $n(\alpha)$ for the union of $fn(\alpha)$ and $bn(\alpha)$.

The side condition in IN guarantees that either y is z or y is not a free name in P. The two rules COM and CLOSE take care of scope extending of local clocks and also other bound names such as private channel names. The rule COM can be understood as follows. When P's local clock, c, is received by Q, the restriction of c reappears, i.e., after the transition is performed, c will be a local clock accessible by both P and Q. Therefore, the scope of c has grown. The

rule for CLOSE can be understood similarly; however, the scopes of both local clock c and bound name z are extended after CLOSE is performed.

Example 3. Using OUT and OPEN we can derive:

$$(y)(c)\bar{x}\langle y,t,c\rangle.P \xrightarrow{\langle\epsilon,\bar{x}\langle(u),t,(c)\rangle,\epsilon\rangle} P\{u/y\}$$

For all u such that u is y or $u \notin fn(P)$. Using IN we have that

$$x(\langle z,t_z,e\rangle).Q \xrightarrow{\langle\epsilon,x(\langle u,t,c\rangle),\epsilon\rangle} Q\{u/z,t/t_z,c/e\}$$

For all u such that u is z or $u \notin fn(Q)$. By applying CLOSE we derive

$$((y)(c)\bar{x}\langle y,t,c\rangle.P \mid x(\langle z,t_z,e\rangle).Q) \xrightarrow{\langle\epsilon,\tau,\epsilon\rangle} (u)(c)(P\{u/y\} \mid Q\{u/z,t/t_z,c/e\})$$

Note that in the formal exposition, we idealize reality and assume that actions are instantaneous; e.g., if a process Q is ready to send a name $\langle z,t_z,c\rangle$ via channel x, then $x(\langle y,t_y,d\rangle).P$ performs the input action immediately and becomes $P\{z/y,t_z/t_y,c/d\}$ in doing so. This is reflected in rules COM, CLOSE, REP-COME and REP-CLOSE of Table 1. However, these assumptions do not put any constraints on the actual systems that are modeled using this formalism.

2.4 Passing Clocks and Channels

Link (channel) passing in timed π-calculus is handled in exactly the same manner as in π calculus, in the sense that a process P can send a public channel x to a process Q. However, if Q already has access to a private channel x before the transition, the latter must be renamed to avoid confusion: this is called scope intrusion [14]. If P has a private link x that it sends to Q, the scope of restriction will be extended, this is called scope extrusion [14]. In this case, if Q already has access to a public link x, then the name of the private link must be changed before the transition. As we mentioned earlier, all clocks in timed π-calculus are local clocks. Moreover, processes access disjoint sets of clocks. When a process P sends its (local) clock c to another process Q, the scope of c will be extended and both P and Q will have access to it.

Assume that $P = (c)\delta\gamma\bar{y}\langle x,t_x,c\rangle.P'$ and $Q = \delta'\gamma' y(\langle z,t_z,d\rangle).Q'$. Furthermore, assume that P sends $\langle x,t_x,c\rangle$ to process Q. This behavior can be captured by the following timed π-calculus transition.

$(P \mid Q)_{(I,w)} \xrightarrow{\langle\delta\delta',\tau,\gamma\gamma'\rangle} (c)(P' \mid Q'\{x/z,t_x/t_z,c/d\})_{(I',w')}$

This is analogous to scope extrusion of private links in π-calculus.

The notion of timed bisimilarity along with its algebraic rules and also structural congruence have been developed for the proposed timed π-calculus, which are not presented here due to lack of space, but can be found in [17].

3 Operational Semantics in Logic Programming

For a complete encoding of the operational semantics of timed π-calculus, we must account for the facts that: (i) clock constraints are posed over continuous time, (ii) infinite computations are defined in timed π-calculus (and also

$$\text{IN}\ \frac{}{\delta\gamma x(\langle z,t,c\rangle).P_{(I,w)} \xrightarrow{\langle \delta\{t'/t,d/c\},x(\langle y,t,d\rangle),\gamma\{d/c\}\rangle} P\{y/z,t'/t,d/c\}_{([\gamma\{d/c\}\mapsto 0]I+w'-w,w')}}\ y\notin fn((z)P), w'>w$$

$$\text{OUT}\ \frac{}{\delta\gamma\bar{x}\langle y,t,c\rangle.P_{(I,w)} \xrightarrow{\langle \delta,\bar{x}\langle y,t,(c)\rangle,\gamma\rangle} P_{([\gamma\mapsto 0]I+w'-w,w')}}\ w'>w$$

$$\text{TAU}\ \frac{}{\delta\gamma\tau.P_{(I,w)} \xrightarrow{\langle \delta,\tau,\gamma\rangle} P_{([\gamma\mapsto 0]I+w'-w,w')}}\ w'>w \qquad \text{SUM}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}}{(P+Q)_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}}$$

$$\text{PAR}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}}{(P\mid Q)_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} (P'\mid Q)_{(I',w')}}\ bn(\alpha)\cap fn(Q)=\emptyset$$

$$\text{COM}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\bar{x}\langle z,t,(c)\rangle,\gamma\rangle} P'_{(I',w')}\quad Q_{(I,w)} \xrightarrow{\langle \delta',x(\langle z,t,c\rangle),\gamma'\rangle} Q'_{(I',w')}}{(P\mid Q)_{(I,w)} \xrightarrow{\langle \delta\delta',\tau,\gamma\gamma'\rangle} (c)(P'\mid Q')_{(I',w')}}$$

$$\text{OPEN}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\bar{x}\langle y,t,(c)\rangle,\gamma\rangle} P'_{(I',w')}}{(y)P_{(I,w)} \xrightarrow{\langle \delta,\bar{x}\langle (u),t,(c)\rangle,\gamma\rangle} P'\{u/y\}_{(I',w')}}\ y\neq x\wedge u\notin fn((y)P')$$

$$\text{CLOSE}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\bar{x}\langle (z),t,(c)\rangle,\gamma\rangle} P'_{(I',w')}\quad Q_{(I,w)} \xrightarrow{\langle \delta',x(\langle z,t,c\rangle),\gamma'\rangle} Q'_{(I',w')}}{(P\mid Q)_{(I,w)} \xrightarrow{\langle \delta\delta',\tau,\gamma\gamma'\rangle} (z)(c)(P'\mid Q')_{(I',w')}}$$

$$\text{RES}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}}{(z)P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} (z)P'_{(I',w')}}\ z\notin n(\alpha), z\in\mathcal{N} \qquad \text{RES-CLK}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}}{(c)P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} (c)P'_{(I',w')}}\ c\in\Gamma$$

$$\text{MATCH}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}}{[x=x]P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}}\ x\in\mathcal{N}\text{ or }x\in\Gamma \qquad \text{REP}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} P'_{(I',w')}}{!P_{(I,w)} \xrightarrow{\langle \delta,\alpha,\gamma\rangle} (P'\mid !P)_{(I',w')}}$$

$$\text{REP-COM}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\bar{x}\langle z,t,(c)\rangle,\gamma\rangle} P'_{(I',w')}\ P_{(I,w)} \xrightarrow{\langle \delta',x(\langle z,t,c\rangle),\gamma'\rangle} P''_{(I',w')}}{!P_{(I,w)} \xrightarrow{\langle \delta\delta',\tau,\gamma\gamma'\rangle} (((c)(P'\mid P''))\mid !P)_{(I',w')}}$$

$$\text{REP-CLOSE}\ \frac{P_{(I,w)} \xrightarrow{\langle \delta,\bar{x}\langle (z),t,(c)\rangle,\gamma\rangle} P'_{(I',w')}\ P_{(I,w)} \xrightarrow{\langle \delta',x(\langle z,t,c\rangle),\gamma'\rangle} P''_{(I',w')}}{!P_{(I,w)} \xrightarrow{\langle \delta\delta',\tau,\gamma\gamma'\rangle} (((z)(c)(P'\mid P''))\mid !P)_{(I',w')}}$$

Table 1. Timed π-calculus Transition Rules

π-calculus) through the infinite replication operator '!', and (iii) we are dealing with mobile concurrent processes. We have developed an implementation of operational semantics of timed π-calculus using coinductive constraint logic programming over reals extended with coroutining in which, channels are modeled as streams and all the three aspects are handled faithfully. Our implementation consists of two phases: in the first phase, we develop the logical denotational semantics of timed π-calculus and show how this semantics yields an interpreter for timed π-calculus. In the second phase we use partial evaluation: given a timed π-calculus expression P and a partial evaluator for pure Prolog, the interpreter can be partially evaluated w.r.t. P to obtain a coroutined coinductive constraint logic program Q. Since Q is obtained automatically via partial evaluation of the interpreter, it is faithful to P, provided the partial evaluator is correct. In the first phase, we develop the logical denotational semantics of timed π-calculus

```
process(P) --> mprocess(P).
process(par(P1,P2)) --> [(],process(P1),[|],process(P2),[)].
process(rep(P)) --> [!],process(P).
process(((N),P)) --> [(],name(N),[)],[(],process(P),[)].
process(match(N1,N2,P)) --> [[],name(N1),[=],name(N2),[]],process(P).
mprocess([Prefix|Processes]) --> prefix(Prefix),[.],process(Processes).
mprocess([process(0)]) --> [0].
mprocess(sum(M1,M2)) --> [(],mprocess(M1),[+],mprocess(M2),[)].
prefix(in(CO,CN,N,T,C)) --> clockOp(CO),name(CN),[(],[<],name(N),[,],
                            time(T),[,],clock(C),[>],[)].
prefix(out(CO,CN,N,T,C)) --> clockOp(CO),name(CN),[<],name(N),[,],
                             time(T),[,],clock(C),[>].
prefix(tau(CO)) --> clockOp(CO),[tau].
clockOp(clockOp(C,R)) --> constraints(C),resets(R).
constraints([Constr|List]) --> constraint(Constr),constraints(List).
constraint((C,Op,R)) --> [(],clock(C),op(Op),const(R),[)].
constraint((C,T,Op,R)) --> [(],clock(C),[-],time(T),op(Op),const(R),[)].
constraint((T,C,Op,R)) --> [(],time(T),[-],clock(C),op(Op),const(R),[)].
resets([Reset|List]) --> reset(Reset),resets(List).
resets([]) --> [].                          constraints([]) --> [].
reset(C) --> [(],clock(C),[:=],[0],[)].
op(O) --> [O],{O = <; O = =<; O = >; O = >=; O = =}.
clock(C) --> [C], {atom(C); var(C)}.        time(T) --> [T],{atom(T)}.
name(N) -->  [N], {atom(N); var(N)}.        const(R) --> [R],{number(R)}.
```

Table 2. DCG for Timed π-calculus Grammar

which yields an executable specification of timed π-calculus. First, we express the grammar of timed π-calculus as a DCG in a straightforward manner as shown in Table 2. There is a one-to-one correspondence between rules in the grammar of timed π-calculus and rules in the DCG. The DCG is a logic program, and when executed in Prolog system, leads to an automatic parser for timed π-calculus. This parser parses a timed π-calculus expression representing a process, and produces a parse tree for it. For instance the query for parsing the expression $((d < 2)\bar{x}\langle w, t_1, d\rangle.0|(c := 0)x(\langle z, t_2, c\rangle).0)$:

```
process( T, [(,(,d,<,2,),x,<,w,',',t1,',',d,>,.,0,|,
               (,c,:=,0,),x,(,<,z,',',t2,',',c,>,),.,0,)], [] ).
```

will produce the parse tree shown below:

```
T = par([out(clockOp([(d,<,2)]), x, w, t1, d), process(0)],
        [in(clockOp([c]), x, z, t2, c), process(0)])
```

Next, we present the denotational semantics where the semantic algebra and the valuation functions are expressed as logic programs. The semantic algebra consists of two store domains: one for storing clocks, realized as an association list of the form `[(Clock, Value),...]`, and one for storing timed events, realized as a list (initially empty). The valuation functions, realized as `sem/10` predicate and shown in Table 3, take the parse tree pattern and the current stores and assign

```
% OUTPUT
sem([out(CO,ChN,N,T,C)|P],C1,E1,O1,C2,E2,O2,[],P,out(ChN,N,W,C)):-
  access(w,C1,W1), {W>W1}, clockOp(CO,C1,Ctemp,W), update(w,W,Ctemp,C2),
  add_events(out(ChN,N,W,C),E1,E2), add_events(out(ChN,N,W,C),O1,O2).
% INPUT
sem([in(CO,ChM,M,T,D)|P],C1,E1,O1,C2,E2,O2,Eq,P,in(ChN,N,W,C)):-
  O1 = [out(ChN,N,W,C)|O2], clockOp(CO,C1,C2,W),
  ((nonvar(ChN),nonvar(ChM))->ChN==ChM, Eq1=[]; ChN=ChM, Eq1=[ChN=ChM]),
  ((nonvar(N),  nonvar(M))  ->N==M,     Eq2=Eq1; N=M, Eq2=[N=M | Eq1]),
  ((nonvar(W),  nonvar(T))  ->W==T,     Eq3=Eq2; W=T, Eq3=[W=T | Eq2]),
  ((nonvar(C),  nonvar(D))  ->C==D,     Eq =Eq3; C=D, Eq =[C=D | Eq3]),
  add_events(in(ChN,N,W,C),E1,E2).
% TAU
sem([tau(CO)|P],C1,E1,O1,C2,E2,O1,[],P,tau(W)):-
  access(w,C1,W1), {W>W1}, clockOp(CO,C1,C2,W), add_events(tau(W),E1,E2).
% MATCH
sem(match(N1,N2,P),C1,E1,O1,C2,E2,O2,Eq,R,A):-
  ( N1==N2 -> sem(P,C1,E1,O1,C2,E2,O2,Eq,R,A)
              ;sem(P,C1,E1,O1,C2,E2,O2,Eq1,R,A ), Eq=[N1=N2|Eq1] ).
% SUM
sem(sum(P1,_P2),C1,E1,O1,C2,E2,O2,Eq,R,A):-
   sem(P1,C1,E1,O1,C2,E2,O2,Eq,R,A).
sem(sum(_P1,P2),C1,E1,O1,C2,E2,O2,Eq,R,A):-
   sem(P2,C1,E1,O1,C2,E2,O2,Eq,R,A).
% PAR
sem(par(P1,P2),C1,E1,O1,C2,E2,O2,Eq,par(Q1,P2),A):-
   sem(P1,C1,E1,O1,C2,E2,O2,Eq,Q1,A).
sem(par(P1,P2),C1,E1,O1,C2,E2,O2,Eq,par(P1,Q2),A):-
   sem(P2,C1,E1,O1,C2,E2,O2,Eq,Q2,A).
% COM
sem(par(P1,P2),C1,E1,O1,C2,E2,O2,Eq,nu(C,par(Q1,Q2)),tau(W)):-
   sem(P1,C1,E1,O1,Ctemp,Etemp,Otemp,Eq1,Q1,A1),
   sem(P2,Ctemp,Etemp,Otemp,C2,E2,O2,Eq2,Q2,A2),
   match(A1,A2,W,C,Eq4), append(Eq1,Eq2,Eq3), append(Eq3,Eq4,Eq).
% OPEN
sem(nu((N),P),C1,E1,O1,C2,E2,O2,Eq,Q,bound_out(ChN,M,W,C)):-
   sem(P,C1,E1,O1,C2,E2,O2,Eq,Q,out(ChN,M,W,C)),
   N \== ChN, N == M, not_in(N,Eq).
% CLOSE
sem(par(P1,P2),C1,E1,O1,C2,E2,O2,Eq,nu((C,N),par(Q1,Q2)),tau(W)):-
   sem(P1,C1,E1,O1,Ctemp,Etemp,Otemp,Eq1,Q1,A1),
   sem(P2,Ctemp,Etemp,Otemp,C2,E2,O2,Eq2,Q2,A2),
   match_bound(A1,A2,N,W,C,Eq4), append(Eq1,Eq2,Eq3), append(Eq3,Eq4,Eq).
% RES
sem(((N),P),C1,E1,O1,C2,E2,O2,Eq,((N),Q),A):-
   sem(P,C1,E1,O1,C2,E2,O2,Eq,Q,A), not_in_act(N,A), not_in(N,Eq).
% REP
sem(rep(P),C1,E1,O1,C2,E2,O2,Eq,par(Q rep(P)),A):-
   sem(P,C1,E1,O1,C2,E2,O2,Eq,Q,A).
% REP-COM
sem(rep(P),C1,E1,O1,C2,E2,O2,Eq,par(((C),par(P1,P2)),rep(P)),tau(W)):-
   sem(P,C1,E1,O1,Ctemp,Etemp,Otemp,Eq1,P1,A1),
   sem(P,Ctemp,Etemp,Otemp,C2,Etemp2,O2,Eq2,P2,A2),
   match(A1,A2,W,C,Eq4), append(Eq1,Eq2,Eq3), append(Eq3,Eq4,Eq),
   add_events(tau(W),Etemp2,E2).
% REP-CLOSE
sem(rep(P),C1,E1,O1,C2,E2,O2,Eq,par(((N,C),par(P1,P2)),rep(P)),tau(W)):-
   sem(P,C1,E1,O1,Ctemp,Etemp,Otemp,Eq1,P1,A1),
   sem(P,Ctemp,Etemp,Otemp,C2,Etemp2,O2,Eq2,P2,A2),
   match_bound(A1,A2,N,W,C,Eq4), append(Eq1,Eq2,Eq3), append(Eq3,Eq4,Eq),
   add_events(tau(W),Etemp2,E2).
```

Table 3. Denotational Semantics of Timed π-calculus in Logic Programming

```
not_in_act(X,in(ChN,N,W,C)) :- X\==ChN, X\==N.     not_in_act(X, tau(W)).
not_in_act(X,out(ChN,N,W,C)) :- X\==ChN, X\==N.
not_in_act(X,bound_out(ChN,N,W,C)) :- X\==ChN, X\==N.

not_in(X,[Y=Z|T]) :- X\==Y, X\==Z, not_in(X,T).    not_in(_X, []).

match(in(H1,N,T,D),out(H2,N2,T2,C),T,C,E) :- (H1==H2 -> E=[]; E=[H1=H2]).
match(out(H1,N,T,C),in(H2,N2,T2,D),T,C,E) :- (H1==H2 -> E=[]; E=[H1=H2]).

match_bound(in(H1,N,T,D),bound_out(H2,N2,T2,C),N,T,C,E) :-
(H1==H2 -> E=[]; E=[H1=H2]).
match_bound(bound_out(H1,N,T,C),in(H2,N2,T2,D),N,T,C,E) :-
(H1==H2 -> E=[]; E=[H1=H2]).
```

Table 4. Denotational Semantics of Timed π-calculus in Logic Programming

meaning to the parse tree pattern, while updating the stores. This denotational semantics can be viewed also as the logical encoding of the operational semantics of timed π-calculus. The encoded operational semantics generates the symbolic transition systems from process expressions using the `sem/10` predicate defined by the rules in Table 3. The syntax of timed π-calculus realized as a DCG along with the semantic algebras and valuation functions realized as a logic program lead to an interpreter for timed π-calculus. `not_in_act/2, not_in/2, match/5` and `match_bound/6` are predicates that handle the side conditions of Table 1 and defined in Table 4. Operations for creating, accessing and updating the stores and clocks are implemented through simple predicates: `access/3, update/4, add_events/3` and `clockOp/3`, which are not presented here due to lack of space.

If the above syntax and semantics rules along with the following logic program are loaded in a CLP system and the query: `?-main(M,Q,A)` is posed, then `Q=((d),par([process(0)],[process(0)]))`, `M=[Z=y,T=t1,E=d]`, `A=tau(B)`, `B>0,B<2`. The updated stores are not shown due to lack of space.

```
main(M, Q, A) :- process(P,[(,(,d,<,2,),x,<,y,',',t1,',',d,>,.,0,|,
                            (,c,:=,0,),x,(,<,Z,',',T,',',E,>,),.,0,)],[]),
        store(S), eventStore(E), outputs(O), sem(P,S,E,O,S2,E2,O2,M,Q,A).
store([(w,0),(c,0),(d,0)]).   eventStore([]).   outputs([]).
```

In the second phase, after obtaining an interpreter for timed π-calculus from its denotational specifications, we use partial evaluation to obtain a compiled code. It is well known in partial evaluation [9] that compiled code for a program $\mathcal{P}$ written in language $\mathcal{L}$ can be obtained by partially evaluating the interpreter for $\mathcal{L}$ w.r.t. the program $\mathcal{P}$. Using this technique with $\mathcal{L}$ being timed π-calculus and program $\mathcal{P}$ being timed π-calculus expressions, we obtain a code which is equivalent to direct encoding of $\mathcal{P}$ as a coroutined coinductive constraint logic program. Thus, in our formulation of the operational semantics of timed π-calculus, clock expressions and time constraints are modeled using *constraint logic programming over reals* [8], (rational) infinite computations, realized through the replication operator of timed π-calculus (and also π-calculus) are handled using *coinductive*

logic programming [20, 5], and finally concurrency is simulated by *coroutining* within logic programming computations. Therefore, timed π-calculus processes are modeled as coroutined coinductive constraint logic programs.

Coinductive constraint logic programming [18] is a paradigm that combines constraint logic programming (CLP) [8] and coinductive logic programming [5, 20, 19]. The operational semantics of coinductive CLP relies on the *coinductive hypothesis rule* and systematically computes elements of the *greatest fixed point* (*gfp*) of a program via backtracking. The coinductive hypothesis rule states that during execution, if the current resolvent R contains a call G' that unifies with an ancestor call G, and the set of accumulated constraints are satisfied, then the call G' succeeds; the new resolvent is $R'\theta$ where θ is the most general unifier of G and G', and R' is obtained by deleting G' from R. Regular constraint logic programming execution extended with the coinductive hypothesis rule is termed *co-constraint logic programming* (or co-CLP)[18]. In co-CLP, predicates can be declared as being either coinductive or inductive. Using co-CLP enables us to handle (i) real time and timing constraints, (ii) the infinite computation, realized by the replication operator of timed π-calculus.

The coroutining feature of logic programming deals with having logic program goals scheduled for execution as soon as some conditions are fulfilled. In LP the most commonly used condition is the instantiation (binding) of a variable. Scheduling a goal to be executed immediately after a variable is bound can be used to model the actions taken by processes as soon as a message is received in the channel specified by that variable. Coroutining can be practically realized through the delay/freeze construct supported in most Prolog systems.

4 Example: The generalized railroad crossing (GRC)

The GRC problem [7] describes a railroad crossing system with several tracks and an unspecified number of trains traveling through the tracks. The gate at the railroad crossing should be operated in a way that guarantees the *safety* and *utility* properties. The *safety* property stipulates that the gate must be down while there is a train in the crossing. The *utility* property states that the gate must be up (or going up) when there is no train in the crossing. We model the system in timed π-calculus, where each component is specified as a timed π-calculus process, and verify properties of the system.

The controller at the railroad crossing might receive various signals from trains in different tracks. In order to avoid signals from different trains being mixed, each train communicates through a private channel with the controller. A new channel is established for each approaching train to the crossing area through which the communication between the train and the controller takes place. For simplicity of presentation we consider only one track in this example. The behavior of the system, specified via three timed automata, is shown in Fig. 1. Note that the design of (1-track) GRC shown in Fig. 1 (originally from [1]) does not account for the delay between the sending of *approach* (*exit*) signal by a train and receiving it by the controller. Similarly the delay between sending *lower* (*raise*) by the controller and receiving it by the gate is not taken into account. In contrast, in our specification of GRC in timed π-calculus, we are considering

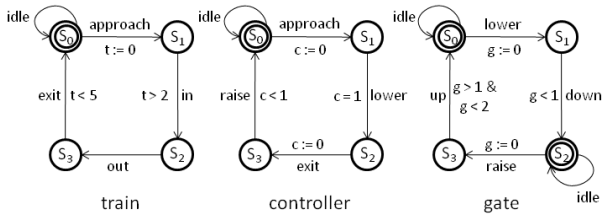


Fig. 1. Timed automata for train, controller, and gate in (1-track) GRC

the delays; therefore, all the time-related reasoning in the system is performed against *train*'s clock and the time-stamp of *approach* signal (sent by *train* to *controller*). The timed π-calculus expressions for processes of (1-track) GRC are presented in Table 4. In the expression for *train*, the two consecutive τ actions correspond to *train*'s internal actions *in* and *out*. Similarly, in the expression for *gate*, the two τ actions correspond to *gate*'s internal actions; the first τ represents *down*; while the second τ represents *up*. The result of partially evaluating the interpreter explained before, w.r.t. program in Table 4 is a program that is equivalent to the coroutined coinductive constraint logic program below.

```
:- coinductive(train).
train(X, Y, St, W, T) :-
 (H=approach, {T2=W}
 ;H=in, {W-T>2, T2=T}
 ;H=out, {T2=T}
 ;H=exit, {W-T<5, T2=T}),
 {W2>W}, t_trans(St, H, St2),
 freeze(X,train(Xs,Ys,St2,W2,T2)),
 ((H=approach; H=exit)->
   Y=[(H,W)|Ys]; Y=Ys), X=[(H,W)|Xs].

:- coinductive(controller).
controller([(H, W)| Xs], Y, Sc) :-
 freeze(Xs, controller(Xs, Ys, Sc3)),
 (H=approach, M=lower, {W2>W, W2-W=1}
 ;H=exit, M=raise, {W2>W, W2-W<1}),
 c_trans(Sc,  H, Sc2),
 c_trans(Sc2, M, Sc3),
 Y = [(M, W2)| Ys].

:- coinductive(gate).
gate([(H, W)| Xs], Sg) :-
 freeze(Xs, gate(Xs, Sg3)),
 ( H=lower, M=down, {W2>W, W2-W<1}
 ; H=raise, M=up, {W2>W, W2-W>1, W2-W<2} ),
 g_trans(Sg,  H, Sg2), g_trans(Sg2, M, Sg3).

g_trans(s0,lower,s1).
g_trans(s1,down, s2).
g_trans(s2,raise,s3).
g_trans(s3,up,   s0).
```

The first argument of `train/5` is the list of timed events (a list of events with their time-stamps) generated by *train*, the second argument is the list of events sent to *controller* and the third argument is *train*'s current state. `W` is the current wall clock time and `T` is *train*'s clock. Similarly, the first argument of `controller/3` is the list of timed events received from *train*; while the second argument is the list of events generated by *controller* and sent to *gate*. `Sc` is the current state of *controller*. Finally, the first argument of `gate/2` is the list of timed events received from *controller*; while the second argument is the current state of *gate*. `g_trans/3` specifies the internal transitions of *gate*; the internal transitions of *train* and *controller*, specified by `t_trans/3` and `c_trans/3` respectively, is omitted. The entire system will wait for *train* to generate the

$train \equiv$	$controller \equiv$
$!(ch)(t)\overline{ch1}\langle ch, t_c, t\rangle.$	$!ch1(\langle y, t_y, d\rangle).y(\langle x, t_x, e\rangle).(c)$
$(t := 0)\overline{ch}\langle approach, t_a, t\rangle.$	$([x = approach](e = 1)(c := 0)\overline{ch2}\langle lower, t_l, c\rangle +$
$(t > 2)\tau.\tau.$	$[x = exit](e - t_x < 1)(c := 0)\overline{ch2}\langle raise, t_r, c\rangle)$
$(t < 5)\overline{ch}\langle exit, t_e, t\rangle)$	
$gate \equiv$	
$!ch2(\langle x, t_x, g\rangle).([x = lower](g < 1)\tau + [x = raise](g > 1)(g < 2)\tau)$	
$main \equiv train \mid controller \mid gate$	

Table 5. The timed π-calculus expressions for components of (1-track) GRC

initial signals and send them to *controller*; as soon as *controller* receives these signals (the first argument of `controller/3` gets bound), it will send appropriate signals to *gate*. This composition of three processes is realized by the expression:

```
freeze(A, (freeze(C, gate(C,s0)),controller(B,C,s0))),train(A,B,s0,0,0).
```

Once the system is modeled as a coinductive coroutined CLP(R) program, the model can be used to verify interesting properties of the system by posing queries. Given a property `Q` to be verified, we specify its negation as a logic program, `notQ`. If the property `Q` holds, the query `notQ` will fail w.r.t. the logic program that models the system. If the query `notQ` succeeds, the answer provides a counterexample to why the property `Q` does not hold.

To prove the *safety* property, we define `unsafe/1` in which, the safety property is negated: we look for any possibility that a train is in the crossing area before the gate goes down, with the gate being up initially. The call to `unsafe/1` fails, which proves the safety of the system. `main(R)` represents the composition of three processes in which R is the time trace of the system after the execution is done. Similarly we check the *utility* property using `unutilized/1` defined below. `unutilized/1` looks for the possibility of a situation in which the gate is down without any train being in the crossing area. Likewise if a call to `unutilized/1` fails we know that the *utility* property is satisfied. We have also verified the *liveness* property by using `not_live/1` predicate, defined below. The *liveness* property states that once the gate goes down, it will not stay down forever. To verify this, we negate the liveness property and look for the possibility that *up* does not appear infinitely often in the accepting timed trace. Coinductive `co_not_member/2` succeeds if *up* does not appear in `R` infinitely often.

```
unsafe(R) :- main(R),
   append(C, [(in, _)| D], R),
   append(A, [(up, _)| B], C),
   not_member((down, _), B).

unutilized(R) :- main(R),
   append(A, [(down, _)| B], R),
   find_first_up(B, C),
   not_member((in, _), C).
% find_first_up/2 returns all the events after 'down' up to 'up'.

not-live(R) :- main(R), co_not_member((up, _), R).
```

5 Conclusions and Related Work

Since the π-calculus was proposed by Milner et al. [14], many researchers have extended it for modeling distributed real-time systems. Berger has introduced

timed π-calculus (π_t-calculus) [2], asynchronous π-calculus with timers and a notion of *discrete* time, locations, and message failure, and explored some of its basic properties. Olarte has studied temporal CCP (tcc) as a model of concurrency for mobile, timed reactive systems in his Ph.D thesis [15]. He has developed a process calculus called universal temporal CCP (utcc). His work can be seen as adding mobile operation to the tcc. In utcc, like tcc, time is conceptually divided into time intervals (or time units); therefore it is discretized. Lee et al. [12] introduced another timed extension of π-calculus called real-time π-calculus (πRT-calculus). They have introduced the time-out operator and considered a global clock, single observer as part of their design, as is common in other (static) real-time process algebras. They have used the set of natural numbers as the time domain, i.e., time is *discrete* and is strictly increasing. Ciobanu et al. [3] have introduced a model called timed distributed π-calculus in which they have considered timers for channels, by which they restrict access to channels. They use *decreasing* timers, and time is discretized in their approach also. Many other timed calculi have similar constructs which also discretize time [4, 10, 13]. In summary, all these approaches share some common features; they use a *discrete* time-stepping function or timers to increase/decrease the time-stamps after every action (they assume that every action takes exactly one unit of time). *In contrast, our approach for extending π-calculus with time faithfully treats time as continuous.*

Posse et al. [16] have proposed π_{klt}-calculus as a real-time extension of π-calculus and study a notion of time-bounded equivalence. They have developed an abstract machine for the calculus and developed an implementation based on this abstract machine for the π_{klt}-calculus in a language called `kiltera`. The replication operator of the original π-calculus is missing in this work.

The work of Yi [24] shows how to introduce time into Milner's CCS to model real-time systems. An extra variable t is introduced which records the time delay before a message on some channel α is available, and also a timer for calculating delays. The idea is to use delay operators to suspend activities. In our opinion, it is much harder to specify real-time systems using delays. Our approach provides a more direct way of modeling time in π-calculus via clocks, and also can be used to elegantly reason about delays.

There has been some efforts on implementing the operational semantics of π-calculus and its various extensions with time; the most notable one and closest one to our approach is the work of Yang et al. [23]. However, this work is different from our work as: (i) it is unable to model infinite processes and infinite replication. In our implementation we are using coinductive logic programming, a more recently developed concept, which allows such modeling, (ii) it models π-calculus but not timed π-calculus, (iii) it does not use Horn logic semantics and partial evaluation.

The proposed timed π-calculus is an expressive, natural model for describing real-time, mobile, concurrent processes and our logic-based implementation of the operational semantics of this calculus provides a framework for modeling and verification of real-time systems and CPS.

References

1. Rajeev Alur and David L. Dill. A theory of timed automata. *TCS*, 126(2):183–235, 1994.
2. Martin Berger. Towards abstractions for distributed systems. Technical report, Imperial College London, 2004.
3. Gabriel Ciobanu and Cristian Prisacariu. Timers for distributed systems. *Electr. Notes Theor. Comput. Sci.*, 164(3):81–99, 2006.
4. Pierpaolo Degano, Jean-Vincent Loddo, and Corrado Priami. Mobile processes with local clocks. In *LOMAPS*, pages 296–319. Springer-Verlag, 1996.
5. Gopal Gupta, Ajay Bansal, Richard Min, Luke Simon, and Ajay Mallya. Coinductive logic programming and its applications. In *ICLP*, pages 27–44. Springer, 2007.
6. Rajesh Gupta. Programming models and methods for spatiotemporal actions and reasoning in cyber-physical systems. In *NSF Workshop on CPS*, 2006.
7. Constance L. Heitmeyer and Nancy A. Lynch. The generalized railroad crossing: A case study in formal verification of real-time systems. In *IEEE RTSS*, pages 120–131, 1994.
8. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
9. Neil D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, September 1996.
10. Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. pages 282–298. Springer-Verlag, 2005.
11. Edward A. Lee. Cyber-physical systems: Design challenges. In *ISORC*, May 2008.
12. Jeremy Y. Lee and John Zic. On modeling real-time mobile processes. *Aust. Comput. Sci. Commun.*, 24(1):139–147, 2002.
13. Manuel Mazzara. Timing issues in web services composition. In *EPEW/WS-FM*, pages 287–302, 2005.
14. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts i and ii. *Inf. Comput.*, 100(1):1–77, September 1992.
15. Carlos Olate. *Universal Temporal Concurrent Constraint Programming.* PhD thesis, LIX, Ecole Polytechnique, 2009.
16. Ernesto Posse and Jürgen Dingel. Theory and implementation of a real-time extension to the *pi*-calculus. In *FMOODS/FORTE*, pages 125–139, 2010.
17. N. Saeedloei. *Modeling and Verification of Real-Time and Cyber-Physical Systems.* PhD thesis, University of Texas at Dallas, Richardson, Texas, 2011.
18. Neda Saeedloei and Gopal Gupta. Coinductive constraint logic programming. In *FLOPS*, pages 243–259, 2012.
19. L. Simon. *Coinductive Logic Programming.* PhD thesis, University of Texas at Dallas, Richardson, Texas, 2006.
20. Luke Simon, Ajay Bansal, Ajay Mallya, and Gopal Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, pages 472–483, 2007.
21. Qian Wang and Gopal Gupta. Provably correct code generation: A case study. *Electr. Notes Theor. Comput. Sci.*, 118:87–109, 2005.
22. Qian Wang, Gopal Gupta, and Michael Leuschel. Towards provably correct code generation via Horn logical continuation semantics. In *PADL*, pages 98–112, 2005.
23. Ping Yang, C. R. Ramakrishnan, and Scott A. Smolka. A logical encoding of the pi-calculus: model checking mobile processes using tabled resolution. *STTT*, 6(1):38–66, 2004.
24. Wang Yi. CCS + time = an interleaving model for real time systems. In *ICALP*, pages 217–228. Springer-Verlag New York, Inc., 1991.

Simplifying the Verification of Quantified Array Assertions via Code Transformation

Mohamed Nassim Seghir and Martin Brain

University of Oxford

Abstract. Quantified assertions pose a particular challenge for automated software verification tools. They are required when proving even the most basic properties of programs that manipulate arrays and so are a major limit for the applicability of fully automatic analysis. This paper presents a simple program transformation approach based on induction to simplify the verification task. The reduction may affect the code as well as the assertion to be verified. Experiments using an implementation of this technique show a significant improvement in performance and the range of programs that can be checked fully automatically.

1 Introduction

Arrays are a primitive data-structure in most procedural languages so it is vital for verification tools to be able to handle them. Most of the properties of interest for array manipulating programs are stated in terms of *all* elements of the array. For example, all elements are within a certain range, all elements are not equal to a target value or all elements are sorted. Handling quantified assertions is thus a prerequisite for supporting reasoning about arrays. Proving them requires reasoning combining quantified assertions and loops. Unfortunately, handling the combination of loops and quantified assertions is difficult and poses considerable practical challenges to most of the existing automatic tools such as SLAM [1], BLAST [11], MAGIC [5], ARMC [18] and SATABS [6]. One approach requires the user provides loop invariants [2, 7]. This is undesirable as writing loop invariants is a skilled, time intensive and error prone task. This paper aims to give a simple program transformation approach for handling some combinations of loops and quantified assertions, which include text book examples as well as real world examples taken from the Linux operating system kernel and the Xen hypervisor.

Transforming the program and assertions before applying existing verification techniques is a common and effective technique. It has a number of pragmatic advantages; it is often simple to implement, fast to use, re-uses existing technology and can be adapted to a wide range of tools. We present an approach that either removes loops or quantifiers and thus reduces the problem to a form solvable by existing tools. For example, if loops can be removed then decision procedures for quantified formulas [4] can be applied to the loop-free code. If quantifiers can be removed then the previously mentioned automatic tools can be used as they

are quite efficient in handling quantifier-free assertions. This work builds on the results presented in [19]; giving a formal basis for the technique, in particular:

1. Defining *recurrent fragments* and shows how they can be used to build inductive proofs of quantified assertions.
2. Giving an algorithm for locating recurrent fragments and transforming the program so that the resulting code is simpler than the original one.
3. Proving the soundness of the program transformation algorithm.

2 Overview

We illustrate our method by considering two sorting algorithms (insertion_sort and selection_sort) which represent challenging examples for automatic verification tools.

2.1 Pre-recursive case (insertion_sort)

Let us consider the insertion_sort example illustrated in Figure 1(a). This program sorts elements of array a in the range $[0..n[$. We want to verify the assertion specified at line 18, which states that elements of array a are sorted in ascending order. Let us call $\mathcal{L}$ the outer `while` loop of example (a) in Figure 1 together with the assignment at line 4 which just precedes the loop. We also call φ the assertion at line 18. In the notation $\mathcal{L}(1, n)$ the first parameter represents the initial value of the loop iterator i and the second parameter is its upper bound (here upper bound means strictly greater). Also $\varphi(n)$ denotes the parameterized form of φ through n. In a Floyd-Hoare style [12], the verification task is expressed as a Hoare triple

$$\{\}\ \mathcal{L}(1, n)\ \{\varphi(n)\} \tag{1}$$

In (1) the initial value 1 for the loop iterator is fixed but its upper bound n can take any value. To prove (1) via structural induction on the iterator range, where the *well-founded order* is range inclusion, it suffices to use induction on n. We proceed as follows

- prove that $\{\}\ \mathcal{L}(1, 1)\ \{\varphi(1)\}$ holds
- assume $\{\}\ \mathcal{L}(1, n-1)\ \{\varphi(n-1)\}$ and prove $\{\}\ \mathcal{L}(1, n)\ \{\varphi(n)\}$

By unrolling the outer `while` loop in Figure 1(a) backward, we obtain the code in Figure 1(b). One can observe that the code fragment from line 4 to 16 in Figure 1(b) represents $\mathcal{L}(1, n-1)$, we call it the *recurrent* fragment. If we denote by $\mathcal{C}$ the remaining code, i.e., from line 18 to 26 in Figure 1(b), then the Hoare triple (1) is rewritten to

$$\{\}\ \ \mathcal{L}(1, n-1); \mathcal{C}\ \{\varphi(n)\}$$

According to Hoare's rule of composition we have

```
1   void insertion_sort (int a [], int n)
2   {
3     int i, j, index;
4     i = 1;
5     while(i < n)
6     {
7       index = a[i];
8       j = i;
9       while ((j > 0) && (a[j−1] > index))
10      {
11        a[j] = a[j−1];
12        j = j − 1;
13      }
14      a[j] = index;
15      i++;
16    }
17
18    assert(∀ x y. (0 ≤ x < n
19          ∧ 0 ≤ y < n
20          ∧ x < y)
21          ⇒ a[x] ≤ a[y]);
22  }
```

(a)

```
1   void insertion_sort (int a [], int n)
2   {
3     int i, j, index;
4     i = 1;
5     while(i < n−1)
6     {
7       index = a[i];
8       j = i;
9       while ((j > 0) && (a[j−1] > index))
10      {
11        a[j] = a[j−1];
12        j = j − 1;
13      }
14      a[j] = index;
15      i++;
16    }
17
18    index = a[n−1];
19    j = n−1;
20    while ((j > 0) && (a[j−1] > index))
21    {
22      a[j] = a[j−1];
23      j = j − 1;
24    }
25    a[j] = index;
26    i = n;
27
28    assert(∀ x y. (0 ≤ x < n
29          ∧ 0 ≤ y < n
30          ∧ x < y)
31          ⇒ a[x] ≤ a[y]);
32  }
```

(b)

Fig. 1. Insertion sort program (a) and the corresponding pre-recursive form (b).

$$\frac{\{\}\ \ \mathcal{L}(1, n-1)\ \ \{\varphi(n-1)\} \qquad \{\varphi(n-1)\}\ \ \mathcal{C}\ \{\varphi(n)\}}{\{\}\ \ \mathcal{L}(1, n-1); \mathcal{C}\ \{\varphi(n)\}}$$

The first (left) premise of the rule represents the induction hypothesis, thus, assumed to be valid. It suffices then to prove the second premise to conclude the validity of (1). Hence, the verification problem in Figure 1(a) is reduced to the one illustrated in Figure 2, in addition to the verification of the basic case $\{\}\ \mathcal{L}(1,1)\ \{\varphi(1)\}$ which is trivial. The assumption at line 1 in Figure 2 represents the postcondition of the induction hypothesis. One can clearly see that the final code is much simpler than the original one. We start with a loop having two levels of nesting, we end up with a code containing a loop that has just one level of nesting, which means less loop invariants (fixed points) to compute.

```
1    assume(∀ x y. (0 ≤ x,y < n−1 ∧ x < y) ⇒ a[x] ≤ a[y]);
2
3    index = a[n−1];
4    j = n−1;
5    while ((j > 0) && (a[j−1] > index))
6    {
7      a[j] = a[j−1];
8      j = j − 1;
9    }
10   a[j] = index;
11   i = n;
12
13   assert(∀ x y. (0 ≤ x,y < n ∧ x < y) ⇒ a[x] ≤ a[y]);
```

Fig. 2. Result obtained by replacing the code fragment of the induction hypothesis with the corresponding post condition (which is here used as assumption).

2.2 Post-recursive case (selection_sort)

Now, we consider the example selection_sort which is illustrated in Figure 3(a). Initially, instead of the two assignments at lines 5 and 6, we had just a single assignment $i := 0$. Also in the assertion at line 23 we had 0 instead of k. We introduced the fresh variable k to allow the application of induction on the iterator initial value as will be illustrated.

As previously, we write $\mathcal{L}(k, n)$ to denote the outer loop together with the assignment at line 6 in Figure 3(a). Unlike the previous example, here by unrolling $\mathcal{L}(k, n)$ backward, the remaining (first) $n - 1$ iterations do not represent $\mathcal{L}(k, n - 1)$. In fact, the iterator j of the inner loop in $\mathcal{L}(k, n)$ has n as upper bound in the first $n-1$ iterations of $\mathcal{L}(k, n)$, whereas j has $n-1$ as upper bound in $\mathcal{L}(k, n - 1)$. However, by unrolling $\mathcal{L}(k, n)$ forward, we obtain the code in Figure 3(b). The code portion from line 20 to 35 represents $\mathcal{L}(k + 1, n)$. In this case, the recursion occurs at the end, i.e., $\mathcal{L}(k, n) = \mathcal{C}; \mathcal{L}(k + 1, n)$, where $\mathcal{C}$ is the code fragment from line 5 to 18 in Figure 3(b).

For both $\mathcal{L}(k, n)$ and $\mathcal{L}(k+1, n)$ the iterator upper bound n is fixed, but the iterator initial value varies (k and $k+1$). Thus, this time we apply induction on the iterator initial value. Hence, we prove the basic case $\{\}\ \mathcal{L}(n, n)\ \{\varphi(n)\}$, then we assume that $\{\}\ \mathcal{L}(k + 1, n)\ \{\varphi(k + 1)\}$ holds and prove $\{\}\ \mathcal{L}(k, n)\ \{\varphi(k)\}$. Replacing $\mathcal{L}(k, n)$ with $\mathcal{C}; \mathcal{L}(k + 1, n)$ in the last formula, we obtain

$$\{\}\ \mathcal{C}; \mathcal{L}(k + 1, n)\ \{\varphi(k)\} \tag{2}$$

Introducing the assumption $\varphi(k + 1)$ resulting from the induction hypothesis into (2), we have

$$\{\}\ \ \mathcal{C}; \mathcal{L}(k + 1, n)\ ; \mathsf{assume}(\varphi(k + 1))\ \{\varphi(k)\}$$

By moving the assumption into the postcondition we obtain

$$\{\}\ \ \mathcal{C}; \mathcal{L}(k + 1, n)\ \{\varphi(k + 1) \Rightarrow \varphi(k)\}.$$

```
1  void selection_sort (int a [], int n)
2  {
3    int i, j, k, s;
4
5    k = 0;
6    i = k;
7    while(i < n)
8    {
9      s = i;
10     for(j = i+1; j < n; ++j)
11     {
12       if(a[j] < a[s])
13       {
14         s = j;
15       }
16     }
17     t = a[i];
18     a[i] = a[s];
19     a[s] = t;
20     i++;
21   }
22
23   assert(∀ x y. (k ≤ x < n
24   ∧ k ≤ y < n
25   ∧ x < y)
26   ⇒ a[x] ≤ a[y]);
27 }
```

(a)

```
1  void selection_sort (int a [], int n)
2  {
3    int i, j, k, s;
4
5    k = 0;
6    i = k;
7    s = k;
8
9    for(j = 1; j < n; ++j)
10   {
11     if(a[j] < a[s])
12     {
13       s = j;
14     }
15   }
16   t = a[k];
17   a[k] = a[s];
18   a[s] = t;
19
20   i = k + 1;
21   while(i < n)
22   {
23     s = i;
24     for(j = i+1; j < n; ++j)
25     {
26       if(a[j] < a[s])
27       {
28         s = j;
29       }
30     }
31     t = a[i];
32     a[i] = a[s];
33     a[s] = t;
34     i++;
35   }
36
37   assert(∀ x y. (k ≤ x < n
38   ∧ k ≤ y < n
39   ∧ x < y)
40   ⇒ a[x] ≤ a[y]);
41 }
```

(b)

Fig. 3. Selection sort (a) and the corresponding post-recursive form (b).

The last formula is simply written

$$\{\}\ \mathcal{L}(k,n)\ \{\varphi(k+1) \Rightarrow \varphi(k)\}.$$

We have

$$\varphi(k) \equiv \forall x\ y.(k \leq x < n \wedge k \leq y < n \wedge x < y) \Rightarrow a[x] \leq a[y]$$

and

$$\varphi(k+1) \equiv \forall x\ y.(k+1 \leq x < n \wedge k+1 \leq y < n \wedge x < y) \Rightarrow a[x] \leq a[y]$$

formula $\varphi(k)$ can be split into two conjuncts

$$\varphi(k) \equiv \varphi(k+1) \wedge \varphi'(k)$$

such that

$$\varphi'(k) \equiv \forall y.\ k+1 \leq y < n \Rightarrow a[k] \leq a[y]$$

thus

$$\varphi'(k) \Rightarrow (\varphi(k+1) \Rightarrow \varphi(k)).$$

Meaning that it suffices to prove

$$\{\}\ \ \mathcal{C}; \mathcal{L}(k+1, n)\ \{\varphi'(k)\}$$

We can iterate the same steps applied to (2) and obtain

$$\{\}\ \ \mathcal{C}; \mathcal{L}(k+1, n)\ \{\varphi'(k+1) \Rightarrow \varphi'(k)\}.$$

One can check that the final postcondition $\varphi'(k+1) \Rightarrow \varphi'(k)$ is simply equivalent to $a[k] \leq a[k+1]$. We start with an assertion having two universally quantified variables and end up with an assertion which is quantifier-free. To the opposite of quantified assertions many existing automatic tools can handle quantifier-free assertions. This time the simplification does not affect the code but weakens the target assertion. Here (post-recursive), the application of induction is slightly different from the previous (pre-recursive) case. We assume that the target property $\varphi(k)$ holds for $k+1$ ($k+1 \leq n-1$) and prove it for k. We can always apply the classical reasoning scheme by rewriting $k+1$ to $n-p$ (p is fresh) as n is fixed. We then apply induction on p: we assume that $\varphi(p)$ holds for p ($p \geq 1$) and prove it for $p+1$.

Question: based on which criterion, we transform loops to pre- or post-recursive form? An answer to this question is given in section 4.

3 Preliminaries

3.1 Loops in canonical form

In our study, we consider loops $\mathcal{L}$ having one of the forms illustrated in Figure 4(a) and Figure 4(b). We say that they are in *canonical* form. The variable i is an *iterator*, it is incremented (decremented) by one at each loop iteration. In Figure 4(a), variable l represents the iterator initial value and variable u represents its upper bound (strictly greater). In Figure 4(b), variable u represents the iterator initial value and l its lower bound (strictly less). The iterator i is not modified within the loop body $\mathcal{B}$. If l and u are not numerical constants, they are also not modified within $\mathcal{B}$.

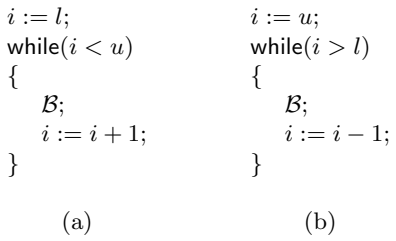


Fig. 4. Canonical form for loops.

3.2 Notations

Given the loop $\mathcal{L}$ in canonical form, $\mathcal{L}.i$ refers to the loop iterator and $\mathcal{L}.s$ is the iterator sign, it can be "+" (increase) or "−" (decrease). The notation $\mathcal{L}(t, b)$ represents a parameterization of $\mathcal{L}$ with the iterator initial value t and the iterator bound b (upper or lower). The iterator range is abstractly given by $[t..b[$. This range is not oriented, which means that t is not necessarily less than b. We just know that the left element of the range is the initial value and the right one is the bound. E.g., $[5..0[$ models the range which is concretely defined by $]0..5]$. The body $\mathcal{B}$ of the loop $\mathcal{L}$ is denoted by $\mathcal{L}.B$. We have the generic operator next which takes the loop $\mathcal{L}$ and a value n for the iterator $\mathcal{L}.i$ as parameters and returns the next value of the iterator. I.e., $\mathsf{next}(\mathcal{L}, n) = n + 1$ if $\mathcal{L}.i$ increases and $\mathsf{next}(\mathcal{L}, n) = n - 1$ if $\mathcal{L}.i$ decreases. We also have the generic operator prev which computes the previous value of the loop iterator. Note that $\mathsf{next}(\mathcal{L}, x)$ for $x \notin [t..b[$ is not defined as well as $\mathsf{prev}(\mathcal{L}, x)$ for $x \notin [t..b]$. In the latter case, $\mathsf{prev}(\mathcal{L}, b)$ gives the last value reached by the loop iterator. I.e., $b-1$ $(b+1)$ if $\mathcal{L}.i$ increases (decreases), assuming that the loop is executed at least once. Finally, the projection of loop $\mathcal{L}$ on the range $[i'..b'[$, such that $[i'..b'[\subseteq [i..b[$, is denoted by $\mathcal{L}([i'..b'[)$. It represents the execution of $\mathcal{L}$ when the loop iterator takes its values in the range $[i'..b'[$. Note that for a given loop $\mathcal{L}(i, b)$, $\mathcal{L}(i', b')$ is different from $\mathcal{L}([i'..b'[)$. The first notation represents the loop obtained by replacing each occurrence of the symbols i and b with i' and b' respectively, which means that changes induced by the substitution can also affect the loop body $\mathcal{L}.B$. However, the loop projection is simply the iterative execution of the loop body $\mathcal{L}.B$ such that $\mathcal{L}.i$ ranges in $[i'..b'[$.

3.3 Array quantified assertions

Let us have the following grammar for predicates

$$\begin{aligned} \mathsf{pred} &::= \mathsf{pred} \wedge \mathsf{pred} \mid \mathsf{pred} \vee \mathsf{pred} \mid \neg \mathsf{pred} \mid \mathsf{e} > \mathsf{e} \mid \mathsf{e} = \mathsf{e} \\ \mathsf{e} &::= \mathsf{int} \mid \mathsf{id} \mid \mathsf{int} * \mathsf{e} \mid \mathsf{id}[\mathsf{i}] \mid \mathsf{e} + \mathsf{e} \\ \mathsf{i} &::= \mathsf{int} \mid \mathsf{id} \mid \mathsf{int} * \mathsf{i} \mid \mathsf{i} + \mathsf{i} \end{aligned}$$

In the grammar above, id stands for an identifier and int represents an integer constant.

We consider universally quantified assertions φ of the form

$$\varphi \equiv \forall x_1, \ldots, x_k \in [l..u[.\ \psi(x_1, \ldots, x_k)$$

such that ψ is generated via the above grammar under the condition that each of the variables $x_1, \ldots, x_k$ must occur in an expression generated by the last rule (head i) of the grammar. In other words, there must be at least one occurrence of each of the variables $x_1, \ldots, x_k$ in the index expression of an array that appears in ψ.

4 Source-to-source transformation

As seen in Section 2, the base of our program transformation is finding the recurrent fragment. In what follows, we formalize a criterion for identifying the recurrent fragment and thus transforming loops to pre- or post-recursive forms. We also present an algorithm which is based on the proposed criterion to soundly transform verification tasks.

4.1 Recurrent Fragments

Given a loop $\mathcal{L}(t, b)$ whose iterator has the range $[t..b[$ of length $|b - t|$, our goal is to identify the code fragment X in $\mathcal{L}(t, b)$ such that $\exists\ t', b' \in [t..b[$, $X = \mathcal{L}(t', b')$ and $|b' - t'| = |b - t| - 1$. It means that (t', b') is either $(\mathsf{next}(\mathcal{L}, t), b)$ or $(t, \mathsf{prev}(\mathcal{L}, b))$, which implies that X is either $\mathcal{L}[\mathsf{next}(\mathcal{L}, t)..b[$ or $\mathcal{L}[t..\mathsf{prev}(\mathcal{L}, b)[$. Thus, the relation between $\mathcal{L}(t, b)$ and X follows one of the forms in (3)

$$\mathcal{L}(t, b) = \begin{cases} \mathcal{C};\ X \\ \text{or} \\ X;\ \mathcal{C} \end{cases} \tag{3}$$

where $\mathcal{C}$ is a code fragment and ";" is the sequencing operator. We call X the *recurrent* fragment.

Let us consider the case where $X = \mathcal{L}(\mathsf{next}(\mathcal{L}, t), b)$, it means that the iterator $\mathcal{L}.i$ in X ranges over $[\mathsf{next}(\mathcal{L}, t)..b[$. The code fragment in $\mathcal{L}(t, b)$ that potentially matches with X can only be the last $|b - t| - 1$ iterations, which corresponds to $\mathcal{L}([\mathsf{next}(\mathcal{L}, t)..b[)$ the projection of $\mathcal{L}(t, b)$ on $[\mathsf{next}(\mathcal{L}, t)..b[$. In this case, we have $\mathcal{L}(t, b) = \mathcal{C}; \mathcal{L}(\mathsf{next}(\mathcal{L}, t), b)$ such that $\mathcal{C}$ corresponds to the initial iteration where the loop iterator $\mathcal{L}.i$ is equal to t. It remains now to check whether the equation below holds

$$\mathcal{L}(\mathsf{next}(\mathcal{L}, t), b) = \mathcal{L}([\mathsf{next}(\mathcal{L}, t)..b[) \tag{4}$$

The code corresponding to the left and right side of the above equation is respectively represented in Figure 5(a) and Figure 5(b).

The symbol $\bowtie$ in Figure 5(a) and Figure 5(b) represents a relational operator which can be either " $<$ " or " $>$ ", depending on whether the loop iterator is increasing or decreasing. We want to find the condition under which the piece of code in Figure 5(a) and the one in Figure 5(b) are equivalent. A possible

```
L.i := next(L, t);
while(L.i ⋈ b)
{
    B[next(L, t)/t];
    L.i := next(L, L.i);
}
```

(a)

```
L.i := next(L, t);
while(L.i ⋈ b)
{
    B;
    L.i := next(L, L.i);
}
```

(b)

Fig. 5. Code fragments corresponding to $\mathcal{L}(\mathsf{next}(\mathcal{L}, t), b)$ (left) and $\mathcal{L}([\mathsf{next}(\mathcal{L}, t)..b[)$ (right).

condition for equivalence is that t is equal to $\mathsf{next}(\mathcal{L}, t)$ the value with which it is replaced (Figure 5(a)). However, we know that the equality $t = \mathsf{next}(\mathcal{L}, t)$ is not possible, thus, we go for a stronger condition and simply choose identity as equivalence relation between programs. It means that the code fragments in Figure 5(a) and Figure 5(b) must be syntactically identical. This requires that changes induced by the substitution in Figure 5(a) must not affect the loop body, i.e., $\mathcal{B}[\mathsf{next}(\mathcal{L}, t)/t] = \mathcal{B}$. This is only true if $\mathcal{B}$ does not contain t. Hence a sufficient condition for the transformation to post-recursive form is given by the following proposition which is valid by construction.

Proposition 1. *(Transformation condition) $\mathcal{L}(t, b)$ is transformable to post-recursive form if the variable t representing the symbolic initial value of the loop iterator does not appear in $\mathcal{L}.B$ the body of $\mathcal{L}(t, b)$.*

A similar reasoning is applied to show that the loop $\mathcal{L}(t, b)$ is transformable to pre-recursive form under the condition that the *variable b representing the symbolic bound of the loop iterator must not appear in the body $\mathcal{L}.B$ of the loop $\mathcal{L}(t, b)$.*

4.2 Transformation algorithm

Based on the previous result (proposition 1) concerning the criterion for loop transformation, we present algorithm Transform (Algorithm 1) which takes a Hoare triple (verification task) H as argument and returns another Hoare triple H', such that, if H' is proven to be valid then H is valid. The algorithm may also return H unchanged if neither the pre- nor the post-recursive transformation criterion is fulfilled. The induction is applied either on the iterator initial value t or its bound b. Therefore we first test, at line 1 of the algorithm, whether φ contains at least one of the symbols t or b, if not then the Hoare triple is not transformed. Function lds takes an expression as parameter and returns the set of symbols in that expression. If the test at line 4 of the algorithm is true, the loop $\mathcal{L}$ is transformed to the pre-recursive form. In this case the reasoning scheme presented in section 2.1 is applied. The induction is then performed on

the iterator bound b, that is why we have the additional condition (at line 4) that b must not be a numerical constant. Following the reasoning scheme of section 2.1, the code fragment in the returned Hoare triple represents the last iteration of the loop, i.e., the loop iterator is assigned $b-1$ if it is increasing (line 6) or it assigned $b+1$ if it is decreasing (line 8). The precondition of the Hoare triple in both cases is simply the post condition of the induction hypothesis, i.e., b in φ is replaced with $b-1$ ($b+1$). If the criterion for the transformation to post-recursive form is true (line 10) the reasoning scheme illustrated in section 2.2 is applied. In this case, the code in the Hoare triple remains unchanged but the postcondition is weakened using the postcondition of the induction hypothesis. This is illustrated in the return statements at lines 12 and 14. Here the induction is applied on the initial value t of the iterator which must not be a numerical constant (line 10).

Back to the example of Figure 1(a), i represents an increasing iterator for the outer loop, its upper bound n does not appear in the loop body, thus, the obtained result (Figure 1(b)) corresponds to the return statement at line 6 of algorithm Transform. Concerning the example of Figure 3(a), n the upper bound of iterator i appears in the loop body, thus, the loop cannot be transformed to pre-recursive form. However k the initial value for i does not appear in the loop body, thus, the loop is transformed to post-recursive form (return statement at line line 12 of algorithm Transform).

For simplicity the algorithm performs just one transformation step. However it can be extended to an iterative version that performs a series of successive transformations by providing the result as input to the next iteration. In this case, a termination criterion is the non decrease of the number of quantifiers in the target assertion.

Proposition 2. *(Soundness) Let us have the Hoare triple* $H = \{\ \}\ \mathcal{L}(t,b)\ \{\varphi\}$ *and* $H' = \mathsf{Transform}(H)$*. If* H' *is true for* $|b-t| \geq 1$ *then* $H' \Rightarrow H$ *for* $|b-t| \geq 1$*, i.e., whenever* H' *is proven to be valid then* H *is also valid.*

Proof. Let us first consider the case where the loop iterator is increasing and the loop is transformable to pre-recursive form. According to the assumption, H' is true for $|b-t| = 1$, thus H' holds for $b = t+1$, i.e., $\{\}\ \mathcal{L}.i := t;\ \mathcal{L}.B\ \{\varphi[t+1/b]\}$. We also know that H' is valid for an arbitrary b ($|b-t| \geq 1$) such that

$$H' = \{\varphi[b-1/b]\}\ \mathcal{L}.i := b-1;\ \mathcal{L}.B\ \{\varphi\}$$

thus, all the following Hoare triples are valid

$$\begin{aligned}
H'_1 &= \{\varphi[t+1/b]\}\ \mathcal{L}.i := t+1;\ \mathcal{L}.B\ \{\varphi[t+2/b]\}\\
&\dots \quad \dots\\
H'_{b-2-t} &= \{\varphi[b-2/b]\}\ \mathcal{L}.i := b-2;\ \mathcal{L}.B\ \{\varphi[b-1/b]\}\\
H'_{b-1-t} &= \{\varphi[b-1/b]\}\ \mathcal{L}.i := b-1;\ \mathcal{L}.B\ \{\varphi\}
\end{aligned}$$

in addition to

$$H'_0 = \{\}\ \mathcal{L}.i := t;\ \mathcal{L}.B\ \{\varphi[t+1/b]\}.$$

According to Hoare's rule of composition, we have

Algorithm 1: Transform

Input: $\{\ \}\ \mathcal{L}(t,b)\ \{\varphi\}$ Hoare triple
Output: Hoare triple

```
1  if b ∉ lds(φ) ∧ t ∉ lds(φ) then
2      return ({ } L(t,b) {φ});
3  end
4  if ((b ∉ lds(L.B)) ∧ (b is not numerical constant)) then
5      if L.s = "+" then
6          return ({φ[b − 1/b]} L.i := b − 1; L.B {φ});
7      else
8          return ({φ[b + 1/b]} L.i := b + 1; L.B {φ});
9      end
10 else if ((t ∉ lds(L.B)) ∧ (t is not numerical constant)) then
11     if L.s = "+" then
12         return ({ } L(t,b) {φ[t + 1/t] ⇒ φ});
13     else
14         return ({ } L(t,b) {φ[t − 1/t] ⇒ φ});
15     end
16 else
17     return ({ } L(t,b) {φ});
18 end
```

$$\frac{H'_0 \qquad H_1 \qquad \dots \qquad H'_{b-2-t} \qquad H'_{b-1-t}}{\{\}\ \mathcal{L}.i := t;\ \mathcal{L}.B\ ;\mathcal{L}.i := t+1;\ \mathcal{L}.B\ \dots\ \mathcal{L}.i := b-1;\ \mathcal{L}.B\ \{\varphi\}}$$

The conclusion of the rule represents $\{\ \}\ \mathcal{L}(t,b)\ \{\varphi\}$ (which is H), thus the rule is simply written

$$\frac{H'_0 \qquad H'_1 \qquad \dots \qquad H'_{b-2-t} \qquad H'_{b-1-t}}{H}$$

Hence, H is valid if H' is valid. In a similar way, we can prove the case where the iterator decreases.

Now, we assume that H is transformable to the post-recursive case and the iterator is decreasing, i.e.,

$$H' = \{\ \}\ \mathcal{L}(t,b)\ \{\varphi[t-1/t] \Rightarrow \varphi\}.$$

The Hoare triple H' is assumed to be valid for an arbitrary value of t s.t., $|b-t| \geq 1$, thus we have

$$\{\ \}\ \mathcal{L}(t-1,b)\ \{\varphi[t-2/t] \Rightarrow \varphi[t-1/t]\}.$$

and we know that (post-recursive form) $\mathcal{L}(t,b) = \mathcal{L}.i := t;\ \mathcal{L}.B;\ \mathcal{L}(t-1,b)$, thus

$$\{\ \}\ \mathcal{L}(t,b)\ \{(\varphi[t-2/t] \Rightarrow \varphi[t-1/t]) \wedge (\varphi[t-1/t] \Rightarrow \varphi)\}$$

or simply

$$\{\ \}\ \mathcal{L}(t,b)\ \{\varphi[t-2/t] \Rightarrow \varphi\}.$$

By reiterating the previous rewriting step (post-recursive form)

$$\mathcal{L}(t,b) = \mathcal{L}.i := t;\ \mathcal{L}.B; \ldots;\ \mathcal{L}.i := b+1;\ \mathcal{L}.B$$

taking into account the assumption that H' holds for $|b-t| = 1$ i.e., $\{\ \}\ \mathcal{L}.i := b+1;\ \mathcal{L}.B\ \{\varphi[b+1/t]\}$ we obtain

$$\{\ \}\ \mathcal{L}(t,b)\ \{\varphi'\}$$

such that

$$\varphi' \equiv \varphi(b+1/t) \wedge (\varphi[b+1/t] \Rightarrow \varphi[b+2/t]) \wedge \ldots \wedge (\varphi[t-1/t] \Rightarrow \varphi).$$

Via modus ponens we obtain $\varphi' \Rightarrow \varphi$, thus $\{\ \}\ \mathcal{L}(t,b)\ \{\varphi\}$ which means that H is valid. Analogically, we can prove the case where the iterator increases.

5 Implementation and Experiments

We have implemented our transformation technique in the software model checker ACSAR [20]. We recall the previously obtained results [19] to show the practical performance enhancement that our approach can bring. Moreover, this transformation allows to verify challenging examples which are out of the scope of many automatic verification tools, in particular sorting algorithms.

Program	Transform	Time (s)	
		S	T
string_copy	PS/PR$^\bullet$	0.39	0.41
scan	PS/PR$^\bullet$	0.27	0.14
array_init	PS/PR$^\bullet$	0.50	0.13
loop1	PS/PR$^\bullet$	0.51	0.21
copy1	PS/PR$^\bullet$	0.70	0.23
num_index	PS/PR$^\bullet$	0.68	0.21
dvb_net_feed_stop*	PS/PR$^\bullet$	3.41	0.30
cyber_init*	PS$^\bullet$/PR	9.47	5.60
perfc_copy_info**	PS/PR$^\bullet$	10.57	1.50
do_enoprof_op**	PS/PR$^\bullet$	8.9	0.54
selection_sort	PS	409.87	173.50
insertion_sort	PR	-	145.60
bubble_sort	PS	-	188.90

Table 1. Experimental results for academic and industrial examples.

The results of our experiments are illustrated in Table 1. The column "Transform" indicates the type of transformation which is applicable, "PR" stands for

pre-recursive and "PS" stands for post-recursive. If both transformations are applicable, we choose the one that delivers the best results and mark it with the superscript $^\bullet$ as illustrated in the table. The column "Time" is divided into two columns "S" which stands for the simple (our previous [21]) approach and "T" which represents the modular approach based on code transformation. Our benchmarks are classified in three categories. The first category (upper part of the table) concerns academic examples taken from the literature. The second class of examples covers typical use of arrays in real world system code. The programs are code fragments taken from the Linux kernel (superscript $*$) and drivers code as well as the Xen hypervisor[1] code (superscript $**$). The last category of benchmarks (sorting algorithms), represents the most challenging examples in terms of complexity. The selection_sort example is handled by both approaches (simple and modular). However, the difference in terms of execution time is considerable. For bubble_sort and insertion_sort, the simple technique seems to diverge as it is unable to terminate within a fixed time bound. However, the modular approach is able to handle both examples in a fairly acceptable time regarding the complexity of the property. To the best of our knowledge, apart from the method presented in [22], no other technique in the literature can handle all these three sorting examples automatically. Please refer to [19] for more details about our experimental study.

6 Related Work

The verification of array quantified assertions received a lot of consideration in recent years. Various ideas and techniques have been developed to verify such properties. However, the idea of performing modular reasoning based on assertion or code decomposition has not yet been addressed. Moreover, the modularity aspect is orthogonal to issues investigated by methods discussed in this section, indeed our technique can also be combined with these methods. Lahiri and Bryant proposed an extension of predicate abstraction to compute universally quantified invariants [15]. Their technique is based on *index predicates* which are predicates that contain free index variables. These index variables are implicitly universally quantified at each program location. In a later work, they described heuristics for inferring index predicates using counterexamples [16]. This approach requires the implementation of adequate predicate transformers. Our method reuses the existing domain of quantifier-free predicates, therefore, it does not require the implementation of specialized domain transformers. In the same category, Jhala and McMillan proposed *range predicates* [13]: predicates that refer to an implicitly quantified variable that ranges over array intervals. An axiom-based algorithm is applied to infer new range predicates as Craig interpolants for the spurious counterexamples. This approach does not handle properties that require quantification over more than one variable. Our approach

[1] A hypervisor is a software that permits hardware virtualization. It allows multiple operating systems to run in parallel on a computer. The Xen hypervisor is available at `http://www.xen.org/`

does not have this restriction. Template-based techniques [3,9,22] consist of providing templates that fix the form of potential invariants. The analysis then searches for an invariant that instantiates the template parameters. Srivastava and Gulwani combined this approach with predicate abstraction to verify properties of arrays with alternating quantifiers [22]. Such properties are out of the scope for our method. However, finding the appropriate template is itself a complicated task, therefore, the template is in general manually provided. Automatic methods to discover relevant templates have not yet been proposed. Recently, a new family of interesting methods based on first-order theorem provers has emerged. McMillan proposed an approach for the computation of universally quantified interpolants by adapting a saturation prover [17]. The technique presented by Kovacs and Voronkov allows the generation of first-order invariants over arrays that contain alternations of quantifiers [14]. Their analysis is based on extracting predicates that encode array updates within a loop. A saturation-based theorem prover is applied to these predicates to infer quantified invariants. The current state of these methods is still limited due to the lack of support for arithmetic theories in the underlying theorem provers. Abstract interpretation has also received its part of interest in the verification of quantified assertions. Gopan *et al.* [8] proposed an idea based on partitioning an array into several symbolic intervals and associating a symbolic variable to each interval. Halbwachs and Péron extended this technique by allowing relational properties between abstract variables which are associated to array intervals [10]. Despite restrictions, their technique seems to handle several interesting properties. However, as for many abstract interpretation based methods, their approach requires the implementation of the appropriate abstract domain as well as the corresponding abstract transformers. This makes their approach less flexible to integrate in state of the art software verification tools.

7 Conclusion

Improving the ability of tools to check quantified properties over arrays is a key issue in automated program verification. This paper defines the concept of *recurrent fragments* and shows, for the case of loops, how the recurrence scheme can be lifted to give an automatic inductive proof. Pattern matching can then be used to locate suitable loops and thus simplify the verification task. An algorithm based on this idea shows that it can be implemented as a source-to-source translation making it low cost (as no fixed points are computed) and allowing it to be used as a front-end for a variety of tools. Experimental results show that this reduces verification time and allows us to verify challenging programs which were out of scope for previous methods. Although this work focuses on simple loop iteration schemes and syntactic matching, the idea of recurrent fragments and linking the code structure to the induction scheme is much wider. Future work includes more flexible and robust matching, handling wider ranges of iteration scheme and generalizing recurrent fragments to handle recursive and parametric program fragments.

References

1. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, pages 1–3, 2002.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. J. 0002, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, pages 364–387, 2005.
3. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. In *VMCAI*, volume 4349 of *LNCS*, pages 378–394, 2007.
4. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *VMCAI*, pages 427–442, 2006.
5. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, pages 385–395, 2003.
6. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Satabs: Sat-based predicate abstraction for ansi-c. In *TACAS*, pages 570–574, 2005.
7. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *ICSE Companion*, pages 429–430, 2009.
8. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.
9. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *POPL*, pages 235–246, 2008.
10. N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.
11. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.
12. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
13. R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.
14. L. Kovacs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *FASE*, 2009.
15. S. K. Lahiri and R. E. Bryant. Constructing quantified invariants via predicate abstraction. In *VMCAI*, pages 267–281, 2004.
16. S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.
17. K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. In *TACAS*, volume 4963 of *LNCS*, pages 413–427, 2008.
18. A. Podelski and A. Rybalchenko. ARMC: the logical choice for software model checking with abstraction refinement. In *PADL*, volume 4354 of *LNCS*, pages 245–259, 2007.
19. M. N. Seghir. An assume guarantee approach for checking quantified array assertions. In *AMAST*, pages 226–235, 2010.
20. M. N. Seghir and A. Podelski. ACSAR: Software model checking with transfinite refinement. In *SPIN*, pages 274–278, 2007.
21. M. N. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. In *SAS*, pages 3–18, 2009.
22. S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, pages 223–234, 2009.

Proving Properties of Co-logic Programs with Negation by Program Transformations

Hirohisa Seki *

Dept. of Computer Science, Nagoya Inst. of Technology,
Showa-ku, Nagoya, 466-8555 Japan
seki@nitech.ac.jp

Abstract. A framework for unfold/fold transformation of (constraint) co-logic programs has been proposed recently, which can be used to prove properties of co-logic programs, thereby allowing us to reason about infinite sequences of events such as behavior of reactive systems. The main problem with this approach is that only definite co-logic programs are considered, thus representing a rather narrow class of co-logic programs. In this paper we consider the "negation technique" by Sato-Tamaki tailored to co-logic programs. The negation technique is a program transformation which, given a program for predicate $p(X)$, derives a program which computes its negation $\neg p(X)$, when the program satisfies certain conditions. We show that the negation technique can be used for co-logic programs, and its application is correct under the alternating fixpoint semantics of co-logic programs. We show by examples how the negation technique, when incorporated into the previous framework for unfold/fold transformation, allows us to represent and reason about a wider class of co-logic programs. We also discuss the difference between the negation technique applied to co-logic programs and the conventional negative unfolding applied to stratified programs.

1 Introduction

Co-logic programming (co-LP) is an extension of logic programming recently proposed by Gupta et al. [5] and Simon et al. [22, 23], where each predicate in definite programs is annotated as either *inductive* or *coinductive*, and the declarative semantics of co-logic programs is defined by an alternating fixpoint model: the least fixpoints for inductive predicates and the greatest fixpoints for coinductive predicates. Predicates in co-LP are defined over infinite structures such as infinite trees or infinite lists as well as finite ones, and co-logic programs allow us to represent and reason about properties of programs over such infinite structures. Co-LP therefore has interesting applications to reactive systems and verifying properties such as safety and liveness in model checking and so on.

A framework for unfold/fold transformation of (constraint) co-logic programs has been proposed recently [20]. The main problem with this approach is that

* This work was partially supported by JSPS Grant-in-Aid for Scientific Research (C) 24500171 and the Kayamori Foundation of Information Science Advancement.

only definite co-logic programs are considered, thus representing a rather narrow class of co-logic programs.

In this paper we consider the "negation technique" by Sato and Tamaki tailored to co-logic programs. The negation technique [18] is a program transformation which, given a program for predicate $p(X)$, derives a program which computes its negation $\neg p(X)$, when the program satisfies certain conditions. We show that the negation technique can be used for co-logic programs, and its application is correct under the alternating fixpoint semantics of co-logic programs.

One of the motivations of this paper is to further study the applicability of techniques based on unfold/fold transformation not only to program development originally due to Burstall and Darlington [1], but also for proving properties of programs, which goes back to Kott [9] in functional programs. We show by examples how the negation technique, when incorporated into the previous framework for unfold/fold transformation, allows us to represent and reason about a wider class of co-logic programs. We also discuss the difference between the negation technique applied to co-logic programs and the conventional negative unfolding applied to stratified programs.

The organization of this paper is as follows. In Section 2, we summarise some preliminary definitions on co-logic programs and the previous framework for our unfold/fold transformation of co-logic programs. In Section 3, we present the negation technique for co-logic programs. In Section 4, we explain by examples how our transformation-based verification method proves properties of co-logic programs. Finally, we discuss about the related work and give a summary of this work in Section 5. [1]

Throughout this paper, we assume that the reader is familiar with the basic concepts of logic programming, which are found in [10].

2 A Framework for Transforming Co-Logic Programs

In this section, we recall some basic definitions and notations concerning co-logic programs. The details and more examples are found in [5, 22, 23]. We also explain some preliminaries on constraint logic programming (CLP) (e.g., [8] for a survey), and our framework for transformation of co-LP.

Since co-logic programming can deal with infinite terms such as infinite lists or trees like $f(f(\dots))$ as well as finite ones, we consider the *complete* (or *infinitary*) Herbrand base [10, 7], denoted by HB^*_P, where P is a program. Fig. 1 (left) shows an example of unfolding, one of basic transformations in our transformation system. The unfolding rule introduces a constraint consisting of equations $X = Y \wedge Y = [a|X]$, which shows the necessity of dealing with infinite terms. The theory of equations and inequations in this case is studied by Colmerauer [3]. Throughout this paper, we assume that there exists at least one constant and one function symbol of arity ≥ 1, thus HB^*_P is non-empty.

[1] Due to space constraints, we omit most proofs and some details, which will appear in the full paper.

$P_k : p \leftarrow q(X, [a|X])$ (1)

$q(Y, Y) \leftarrow r(Y)$ (2)

Unfold :

$P_{k+1} = (P_k \setminus \{(1)\}) \cup \{(3)\},$

$p \leftarrow c \,[]\, r(Y)$ (3)

where

$c \equiv (X = Y \wedge Y = [a|X])$

$P'_k : p \leftarrow X_1 = X, X_2 = [a|X] \,[]\, q(X_1, X_2)$ (1′)

$q(Y_1, Y_2) \leftarrow Y_1 = Y_2, Y_1 = Y_3 \,[]\, r(Y_3)$ (2′)

Unfold :

$P'_{k+1} = (P'_k \setminus \{(1')\}) \cup \{(3')\},$

$p \leftarrow c' \,[]\, r(Y_3)$ (3′)

where

$c' \equiv (X_1 = X \wedge X_2 = [a|X] \wedge X_1 = Y_1 \wedge X_2 = Y_2 \wedge Y_1 = Y_2 \wedge Y_1 = Y_3)$

Fig. 1. Example of Unfolding (left) and its CLP form (right)

Fig. 1 (right) shows the counterparts to the clauses in the figure (left) in standard CLP form. In general, let $\gamma : p(\tilde{t}_0) \leftarrow p_1(\tilde{t}_1), \ldots, p_n(\tilde{t}_n)$ be a (logic programming) clause, where $\tilde{t}_0, \ldots, \tilde{t}_n$ are tuples of terms. Then, γ is mapped into the following *pure* CLP clause:

$$p(\tilde{x}_0) \leftarrow \tilde{x}_0 = \tilde{t}_0 \wedge \tilde{x}_1 = \tilde{t}_1 \wedge \cdots \wedge \tilde{x}_n = \tilde{t}_n \,[]\, p_1(\tilde{x}_1), \ldots, p_n(\tilde{x}_n),$$

where $\tilde{x}_0, \ldots, \tilde{x}_n$ are tuples of new and distinct variables, and $[]$ means conjunction ("$\wedge$"). We will use the conventional representation of a (logic programming) clause as a shorthand for a pure CLP clause, for the sake of readability.

In the following, for a CLP clause γ of the form: $H \leftarrow c \,[]\, B_1, \ldots, B_n$, the head H and the body $B_1, \ldots, B_n$ are denoted by $hd(\gamma)$ and $bd(\gamma)$, respectively. We call c the *constraint* of γ. A conjunction $c \,[]\, B_1, \ldots, B_n$ is said to be a *goal* (or a *query*). The predicate symbol of the head of a clause is called the *head predicate* of the clause.

The set of all clauses in a program P with the same predicate symbol p in the head is called the definition of p and denoted by $Def(p, P)$. We say that a predicate p *depends on* a predicate q in P, iff either (i) $p = q$, (ii) there exists in P a clause of the form: $p(\ldots) \leftarrow c \,[]\, B$ such that predicate q occurs in B or (iii) there exists a predicate r such that p depends on r in P, and r depends on q in P. The *extended definition* [14] of p in P, denoted by $Def^*(p, P)$, is the conjunction of the definitions of all the predicates on which p depends in P.

2.1 Syntax and Semantics of Co-logic programs

A *co-logic program* is a constraint definite program, where predicate symbols are annotated as either inductive or coinductive.[2] Let $\mathcal{P}$ be the set of all predicates in a co-logic program P, and we denote by $\mathcal{P}^{in}$ ($\mathcal{P}^{co}$) the set of inductive (coinductive) predicates in $\mathcal{P}$, respectively. There is one restriction on co-LP, referred to as the *stratification restriction*: Inductive and coinductive predicates are not allowed to be mutually recursive. An example which violates the stratification

[2] We call an atom, A, an *inductive* (a *coinductive*) atom when the predicate of A is an inductive (a coinductive) predicate, respectively.

restriction is $\{p \leftarrow q;\ q \leftarrow p\}$, where p is inductive, while q is coinductive. When P satisfies the stratification restriction, it is possible to decompose the set $\mathcal{P}$ of all predicates in P into a collection (called a *stratification*) of mutually disjoint sets $\mathcal{P}_0, \ldots, \mathcal{P}_r$ $(0 \leq r)$, called *strata*, so that, for every clause

$$p(\tilde{x}_0) \leftarrow c \,[\!]\, p_1(\tilde{x}_1), \ldots, p_n(\tilde{x}_n),$$

in P, we have that $\sigma(p) \geq \sigma(p_i)$ if p and p_i have the same inductive/coinductive annotations, and $\sigma(p) > \sigma(p_i)$ otherwise, where $\sigma(q) = i$, if the predicate symbol q belongs to $\mathcal{P}_i$. The following is an example of co-logic programs.

Example 1. [23]. Suppose that predicates *member* and *drop* are annotated as inductive, while predicate *comember* is annotated as coinductive.

$$\begin{array}{ll} member(H, [H|\_]) \leftarrow & drop(H, [H|T], T) \leftarrow \\ member(H, [\_|T]) \leftarrow member(H, T) & drop(H, [\_|T], T_1) \leftarrow drop(H, T, T_1) \end{array}$$

$$comember(X, L) \leftarrow drop(X, L, L_1), comember(X, L_1)$$

The definition of *member* is a conventional one, and, since it is an inductive predicate, its meaning is defined in terms of the least fixpoint. Therefore, the prefix ending in the desired element H must be finite. The similar thing also holds for predicate *drop*.

On the other hand, predicate *comember* is coinductive, whose meaning is defined in terms of the greatest fixpoint (see the next section). Therefore, it is true if and only if the desired element X occurs an infinite number of times in the list L. Hence it is false when the element does not occur in the list or when the element only occurs a finite number of times in the list. □

Semantics of Co-Logic Programs The declarative semantics of a co-logic program is a stratified interleaving of the least fixpoint semantics and the greatest fixpoint semantics.

In this paper, we consider the complete Herbrand base HB_P^* as the set of elements in the domain of a *structure* $\mathcal{D}$ (i.e., a complete Herbrand interpretation [10]).

Given a structure $\mathcal{D}$ and a constraint c, $\mathcal{D} \models c$ denotes that c is true under the interpretation for constraints provided by $\mathcal{D}$. Moreover, if θ is a ground *substitution* (i.e., a mapping of variables on the domain $\mathcal{D}$, namely, HB_P^* in this case) and $\mathcal{D} \models c\theta$ holds, then we say that c is *satisfiable*, and θ is called a *solution* (or ground *satisfier*) of c, where $c\theta$ denotes the application of θ to the variables in c. We refer to [3] for an algorithm for checking constraint satisfiability.

Let P be a co-logic program with a stratification $\mathcal{P}_0, \ldots, \mathcal{P}_r$ $(0 \leq r)$. Let Π_i $(0 \leq i \leq r)$ be the set of clauses whose head predicates are in $\mathcal{P}_i$. Then, $P = \Pi_0 \cup \ldots \cup \Pi_r$. Similar to the "immediate consequence operator" T_P in the literature, our operator $T_{\Pi,S}$ assigns to every set I of ground atoms a new set

$T_{\Pi,S}(I)$ of ground atoms as

$$T_{\Pi,S}(I) = \{A \in HB^*_\Pi \mid \text{there is a ground substitution } \theta \text{ and a clause in } \Pi$$
$$H \leftarrow c \,[]\, B_1, \cdots, B_n,\ n \geq 0, \text{ such that}$$
$$\text{(i) } A = H\theta, \text{ (ii) } \theta \text{ is a solution of } c, \text{ and}$$
$$\text{(iii) for every } 1 \leq i \leq n, \text{ either } B_i\theta \in I \text{ or } B_i\theta \in S\}.$$

In the above, the atoms in S are treated as facts. S is intended to be a set of atoms whose predicate symbols are in lower strata than those in the current stratum Π. We consider $T_{\Pi,S}$ to be the operator defined on the set of all subsets of HB^*_Π, ordered by standard inclusion. Next, two subsets $T^{\uparrow\alpha}_{\Pi,S}$ and $T^{\downarrow\alpha}_{\Pi,S}$ of the complete Herbrand base are defined as:

$$T^{\uparrow 0}_{\Pi,S} = \phi \text{ and } T^{\downarrow 0}_{\Pi,S} = HB^*_\Pi \ ;$$

$$T^{\uparrow n+1}_{\Pi,S} = T_{\Pi,S}(T^{\uparrow n}_{\Pi}) \text{ and } T^{\downarrow n+1}_{\Pi,S} = T_{\Pi,S}(T^{\downarrow n}_{\Pi,S}) \text{ for a successor ordinal } n;$$

$$T^{\uparrow\alpha}_{\Pi,S} = \cup_{z<\alpha} T^{\uparrow z}_{\Pi,S} \text{ and } T^{\downarrow\alpha}_{\Pi,S} = \cap_{z<\alpha} T^{\downarrow z}_{\Pi,S}, \text{ for a limit ordinal } \alpha.$$

Finally, the model $M(P)$ of a co-logic program $P = \Pi_0 \cup \ldots \cup \Pi_r$ is defined inductively as follows: First, for the bottom stratum Π_0, let $M(\Pi_0) = T^{\uparrow\omega}_{\Pi_0,\emptyset}$, if $\mathcal{P}_0$ is inductive; $gfp(T_{\Pi_0,\emptyset})$ otherwise.

Next, for $k > 0$, let:

$$M(\Pi_k) = \begin{cases} T^{\uparrow\omega}_{\Pi_k, M_{k-1}}, & \text{if } \mathcal{P}_i \text{ is inductive,} \\ gfp(T_{\Pi_k, M_{k-1}}), & \text{if } \mathcal{P}_i \text{ is coinductive.} \end{cases}$$

where M_{k-1} is the model of lower strata than Π_k, i.e., $M_{k-1} = \cup_{i=0}^{k-1} M(\Pi_i)$. Then, the *model* of P is $M(P) = \cup_{i=0}^{r} M(\Pi_i)$, the union of all models $M(\Pi_i)$.

2.2 Transformation Rules for Co-Logic Programs

We first explain our transformation rules, and then give some conditions imposed on the transformation rules which are necessary for correctness of transformation. Our transformation rules are formulated in the framework of CLP, following that by Etalle and Gabbrielli [4].

A sequence of programs $P_0, \ldots, P_n$ is said to be a *transformation sequence* with a given initial program P_0, if each P_{k+1} $(0 \leq k \leq n-1)$ is obtained from P_k by applying one of the following transformation rules R1-R4.

Our motivation of this paper is to use program transformation rules for proving properties of a given system represented in a co-logic program P_0. We thus assume that there exist two kinds of predicate symbols appearing in a transformation sequence: *base* predicates and *defined* predicates. A base predicate is defined in P_0, and it is intended to represent the structure and the behaviours of the given system. Its definition is therefore assumed to remain unchanged during program transformation. On the other hand, a defined predicate, which is introduced by the following *definition introduction* rule, is intended to represent a property of the given system such as safety and liveness properties. Its definition

will be changed during program transformation so that its truth value in $M(P_n)$ will be easily known.

R1. Definition Introduction Let δ be a clause of the form: $newp(\tilde{X}) \leftarrow c [\!] B$, where (i) *newp* is a *defined* predicate symbol not occurring in P_k, (ii) c is a constraint, and (iii) B is a conjunction of atoms whose predicate symbols are all base predicates appearing in P_0. Moreover, δ satisfies the following conditon called the *annotation rule*: *newp* is annotated as *inductive* iff there exists at least one inductive atom in B, while *newp* is annotated as *coinductive* iff every predicate symbol occurring in B is annotated as *coinductive*.
By *definition introduction*, we derive from P_k the new program $P_{k+1} = P_k \cup \{\delta\}$. For $n \geq 0$, we denote by $Defs_n$ the set of clauses introduced by the definition introduction rule during the transformation sequence $P_0, \ldots, P_n$. In particular, $Defs_0 = \emptyset$.

As mentioned above, the role of the predicate annotations such as "base" and "defined" is to specify the conditions for applying our transformation rules; they are orthogonal to the predicate annotations of "inductive" and "coinductive", which are used to represent the intended semantics of each predicate in co-logic programs. In particular, since the definition of $newp(\tilde{X})$ is non-recursive, its meaning is determined only by the base predicates B in P_k, irrelevant of whether its annotation is inductive or coinductive. However, some application of transformation rules to δ could possibly derive a clause in P_n $(n > k)$ of the form: $newp(\tilde{X}) \leftarrow newp(\tilde{X})$. Then, the annotation of predicate *newp* will matter, which is determined according to the above-mentioned annotation rule.

Our transformation rules include two basic rules: *unfolding* (R2) and *folding* (R3). They are the same as those in Etalle and Gabbrielli [4], which are CLP counterparts of those by Tamaki and Sato [24] for definite programs, so we omit these definitions here;

The following *replacement rule* allows us to make simple the definition of a defined predicate. The notion of *useless* predicates is originally due to Pettorossi and Proietti [14], where the rule is called *clause deletion rule*.

R4. Replacement Rule We consider the following two rules depending on the annotation of a predicate.

- The set of the *useless* inductive predicates of a program P is the maximal set U of inductive predicates of P such that a predicate p is in U if, for the body of each clause of $Def(p, P)$, it has an inductive atom whose predicate is in U. By applying the *replacement rule* to P_k w.r.t. the useless inductive predicates in P_k, we derive the new program P_{k+1} from P_k by removing the definitions of the useless inductive predicates.
- Let $p(\tilde{t})$ be a coinductive atom, and γ be a clause (modulo variance $\simeq$) in P_k of the form: $p(\tilde{t}) \leftarrow p(\tilde{t})$. By applying the *replacement rule* to P_k w.r.t. $p(\tilde{t})$, we derive from P_k the new program $P_{k+1} = (P_k \setminus \{\gamma\}) \cup \{p(\tilde{t}) \leftarrow\}$.

2.3 Correctness of the Transformation Rules

In order to preserve the alternating fixpoint semantics of a co-logic program, we will impose some conditions on the application of folding rule. The conditions on folding are given depending on whether the annotation of the head predicate *newp* of a folded clause is inductive or not.

First, let *newp* be an *inductive* head predicate of a clause δ introduced by rule R1 (definition introduction). We call an inductive predicate p *primitive*, if, for some coinductive predicate q on which *newp* depends in $P_0 \cup \{\delta\}$, q depends on p in P_0; put simply, *newp* depends on an inductive predicate p through some coinductive predicate q. We denote by $\mathcal{P}_{pr}$ the set of all the primitive predicates, i.e., $\mathcal{P}_{pr} = \{p \in \mathcal{P}^{in} \mid \exists \delta \in \mathit{Defs}_n \exists q \in \mathcal{P}^{co},$ *newp* is the head predicate of δ s.t. *newp* depends on q in $P_0 \cup \{\delta\}$, and q depends on p in $P_0.\}$. We call an inductive predicate p *non-primitive*, if it is not primitive. We call an atom with non-primitive (primitive) predicate symbol a non-primitive (primitive) atom, respectively.

Let $P_0, \ldots, P_n$ be a transformation sequence, and δ be a clause first introduced by rule R1 in Defs_i $(0 \leq i \leq n)$, Then, we mark it "*not TS-foldable*". Let γ be a clause in P_{k+1} $(0 \leq k < n)$. γ inherits its mark when γ is not involved in the derivation from P_k to P_{k+1}. γ is marked "*TS-foldable*", only if γ is derived from $\beta \in P_k$ by unfolding β of the form: $A \leftarrow c \,[\!]\, H, G$ w.r.t. H and H is a *non-primitive* inductive atom. Otherwise, γ inherits the mark of β. Similarly, γ inherits the mark of β, if γ is derived from $\beta \in P_k$ by folding.

When there are no coinductive predicates in $P_0 \cup \mathit{Defs}_n$, the above TS-foldable condition coincides with the one by Tamaki-Sato [24].

Next, let *newp* be an *coinductive* defined predicate of a clause δ introduced by rule R1. The next notion is due to [21], which is originally introduced to give a condition on folding to preserve the finite failure set.

Let $P_0, \ldots, P_n$ be a transformation sequence, and δ be a clause first introduced by rule R1 in Defs_i $(0 \leq i \leq n)$. Then, each atom in the body $bd(\delta)$ is marked *inherited* in δ. Let γ be a clause in P_{k+1} $(0 \leq k < n)$. When γ is not involved in the derivation from P_k to P_{k+1}, each atom B in $bd(\gamma)$ is marked *inherited*, if so is it in γ in P_k. Otherwise, i.e., suppose that γ is derived by applying to some clause β in P_k either unfolding rule or folding rule. Then, each atom B in $bd(\gamma)$ is marked *inherited*, if it is not an atom introduced to $bd(\gamma)$ by that rule, and it is marked inherited in β in P_k. Moreover, the application of folding is said to be *fair*, if there is no folded atom in $bd(\beta)$ which is marked inherited. Intuitively, an atom marked inherited is an atom such that it was in the body of some clause in Defs_i $(0 \leq i \leq n)$, and no unfolding has been applied to it.

We are now in a position to state the conditions imposed on folding and the correctness of our transformation rules.

Conditions on Folding Let $P_0, \ldots, P_n$ be a transformation sequence. Suppose that P_k $(0 < k \leq n)$ is derived from P_{k-1} by folding $\gamma \in P_{k-1}$. The application of folding is said to be *admissible* if the following conditions are satisfied:
(1) P_k satisfies the stratification restriction,

(2) if $hd(\gamma)$ is an inductive atom, then γ is marked "TS-foldable" in P_{k-1}, and
(3) if $hd(\gamma)$ is a coinductive atom, then the application of folding is fair. □

Proposition 1. Correctness of Transformation [20]
Let P_0 be an initial co-logic program, and $P_0, \ldots, P_n$ $(0 \leq n)$ a transformation sequence, where every application of folding is admissible. Then, the transformation sequence is correct, i.e., $M(P_0 \cup Defs_n) = M(P_n)$. □

3 The Negation Technique in Co-Logic Programs

To deal with a negative literal in a goal, we use the *negation technique* by Sato and Tamaki [18]. The negation technique is a procedure to derive a set S' of definite clauses from a given definite program S such that (i) each predicate symbol p in S has one-to-one correspondence with a new predicate symbol, not_p, in S' with the same arity, and (ii) for any ground atom $p(\tilde{t})$ and $not\_p(\tilde{t})$,

$$M(S) \models \neg p(\tilde{t}) \text{ iff } M(S') \models not\_p(\tilde{t}) \qquad (*)$$

If S' satisfies the above conditions, it is called a *complement program* of S. Moreover, when "iff" in $(*)$ is replaced with "if", it is called a *dual program* of S. In [18], S is a definite program, and $M(S)$ is its least Herbrand model. In the following, we show the negation technique for co-logic programs under the alternating fixpoint semantics.

We explain the negation technique by using an example for saving space. Consider *drop* in Example 1, and let S be the definition of *drop*. We will derive the definition of *not_drop* (*not_d* for short) by applying the following steps:

(Step 1) Consider the completed definition of *drop*. In this case, we have:

$$drop(A,B,C) \leftrightarrow (\exists H,T)(\langle A,B,C\rangle = \langle H,[H|T],T\rangle) \ \vee \\ (\exists H,X,T,T_1)(\langle A,B,C\rangle = \langle H,[X|T],T_1\rangle \wedge drop(H,T,T_1)) \qquad (1)$$

We denote by $\tilde{V} = \tilde{t}_1$ ($\tilde{V} = \tilde{t}_2$) the first (second) equation in (1), respectively.
(Step 2) Negate both sides of the completed definition, and every negative occurrence $\neg p(\tilde{t})$ is replaced by $not\_p(\tilde{t})$. In this case, we have:

$$not\_d(A,B,C) \leftrightarrow (\forall H,T)(\tilde{V} \neq \tilde{t}_1) \wedge (\forall H,X,T,T_1)(\tilde{V} \neq \tilde{t}_2 \vee not\_d(H,T,T_1)) \qquad (2)$$

(Step 3) Transform every conjunct on the right-hand side of the result of (Step 2) which is of the form: $(\forall \tilde{X})(\langle \tilde{A} \neq \tilde{t}\rangle \vee not\_p_1(\tilde{t}_1) \vee \cdots \vee not\_p_m(\tilde{t}_m))$ $(m \geq 1)$ to $(\forall \tilde{X})\langle \tilde{A} \neq \tilde{t}\rangle \vee (\exists \tilde{X})(\langle \tilde{A} = \tilde{t}\rangle \wedge not\_p_1(\tilde{t}_1)) \vee \cdots \vee (\exists \tilde{X})(\langle \tilde{A} = \tilde{t}\rangle \wedge not\_p_m(\tilde{t}_m))$. Note that, when some clause in S has an existential variable, this transformation is not valid. In this case we obtain from (2):

$$not\_d(A,B,C) \leftrightarrow (\forall H,T)(\tilde{V} \neq \tilde{t}_1) \ \wedge \\ \{(\forall H,X,T,T_1)(\tilde{V} \neq \tilde{t}_2) \vee (\exists H,X,T,T_1)(\tilde{V} = \tilde{t}_2 \wedge not\_d(H,T,T_1))\} \qquad (3)$$

(Step 4) Transform the right-hand side to a disjunctive form. In this case, we obtain from (3):

$$not\_d(A,B,C) \leftrightarrow ((\forall H,T)(\tilde{V} \neq \tilde{t}_1) \wedge (\forall H,X,T,T_1)(\tilde{V} \neq \tilde{t}_2)) \vee \{(\forall H,T)(\tilde{V} \neq \tilde{t}_1) \wedge (\exists H,X,T,T_1)(\tilde{V} = \tilde{t}_2 \wedge not\_d(H,T,T_1))\} \quad (4)$$

(Step 5) Transform the completed definition given as the result of (Step 4) to a set of clauses, and then simplify constraints assuming that each predicate p is typed. Annotate the derived predicate as "coinductive" (resp. "inductive) if the annotation of the original predicate is inductive (resp. coinductive). In this case, we obtain from (4), assuming that B and C are lists:

$$not\_drop(A,[\,],C) \leftarrow$$
$$not\_drop(H,[X|T],T_1) \leftarrow X \neq H, not\_drop(H,T,T_1)$$
$$not\_drop(H,[H|T],T_1) \leftarrow T \neq T_1, not\_drop(H,T,T_1)$$

not_drop is annotated as "coinductive", since predicate *drop* is inductive.
(Step 6) Apply to all predicates in $Def^*(p,S)$ (Step 1) to (Step 5), and let S' is the set of the resulting clauses. We call S' the *result* of the negation technique applied to p in S.

The above transformations themselves are exactly the same as those in the original negation technique [18] for definite programs; the only difference is the necessity of predicate annotations in (Step 5). In the original negation technique, an extra condition such as the finiteness of SLD-tree for $p(\tilde{t})$ is necessary to obtain a *complementary* program, while such a condition is not needed here.

Proposition 2. Correctness of the Negation Technique
Let S be a co-logic program. If every clause in S has no existential variable, then the procedure of the negation technique (Step 1) to (Step 6) gives a complementary co-logic program S', i.e., for any ground term $\tilde{t}$,

$$M(S) \models \neg p(\tilde{t}) \text{ iff } M(S') \models not\_p(\tilde{t}).$$

Proof. (Sketch) Let S' be the definition of *not_drop* obtained in (Step 5) in the above. Since the general case will be shown similarly, we explain an outline of the proof that S' is a complementary program of S (i.e., the definition of *drop*).

We first note that, when S satisfies the stratification restriction, then so does S'. Let I' be the greatest fixpoint $gfp(S')$ of S', noting that *not_drop* is annotated as coinductive. Since I' is a fixpoint, it satisfies the IFF-definition of (4), and it also satisfies (3) and (2) in turn. We define:

$$I = \{drop(\tilde{t}) \mid not\_drop(\tilde{t}) \notin I'.\ \tilde{t} \text{ is a sequence of ground terms.}\}$$

Then, we can show that I satisfies (1) and is the least fixpoint model of S. □

Remark 1. In [18], Sato and Tamaki described an extension of the negation technique to the case where a clause in S has an existential variable. Namely, even when there are clauses with existential variables, (Step 3) in the negation technique is still valid, if the clauses have a *functional part* [18]: that is, let γ be a clause in P of the form: $p_0(X) \leftarrow p_1(X,Y),\ldots,p_m(X,Y)$. Then, we

say that γ with an existential variable Y has a *functional part* p_1 iff $p_1(X,Y)$ defines a partial function from X to Y (that is, whenever $M(P) \models p_1(a,b)$ and $M(P) \models p_1(a,b')$, then $b = b'$ holds for any ground terms a, b, b').

This is also true in co-logic programs, and we show it by the following example. The definition of predicate *comember* in Ex. 1 has an existential variable (i.e., L_1); we cannot apply the negation technique to $\neg comember(X,L)$. Moreover, it does not have a functional part, since $drop(X,L,L_1)$ does not define a partial function. Instead, we consider the following definition of $comember'$:

1. $comember'(X,L) \leftarrow drop'(X,L,L_1), comember'(X,L_1)$
2. $drop'(H,[H|T],T) \leftarrow$
3. $drop'(H,[H_1|T],T_1) \leftarrow H \neq H_1, drop'(H,T,T_1)$

where $drop'(X,L,L_1)$ now defines a partial function from (X,L) to L_1.

Applying the negation technique to $\neg comember'(X,L)$ and simple unfold/fold transformations, we obtain the following clauses:

4. $not\_comem(X,L) \leftarrow not\_d(X,L)$
5. $not\_comem(X,L) \leftarrow drop'(X,L,L_1), not\_comem(X,L_1)$
6. $not\_d(H,[]) \leftarrow$
7. $not\_d(H,[H_1|T]) \leftarrow H \neq H_1, not\_d(H,T)$

where we denote $not\_comember'$ by not_comem for short, and not_d is a coinductive predicate. We will use this in Example 3. □

In conventional program transformation for logic programs, *negative unfolding* ([14, 19] for example) is applied to a negative literal in the body of a clause. The following example will show the difference between the negation technique in co-LP and negative unfolding in (locally) stratified programs, which reflects the differences underlying the two different semantics.

Example 2. Consider the following stratified program $P_0 = \{1,2,3\}$:

1. $p \leftarrow \neg q(X)$
2. $q(X) \leftarrow q(X)$
3. $q(X) \leftarrow r$
4. $p \leftarrow \neg q(X), \neg r$ (neg. unfold 1)
5. $p \leftarrow p, \neg r$ (fold 4)

We first consider a *wrong* transformation sequence [14]: we apply to clause 1 negative unfolding w.r.t. $\neg q(X)$, followed by folding the resulting clause. Let $P_2 = (P_0 \setminus \{1\}) \cup \{5\}$. Then, we have that $PERF(P_0) \models p$, while $PERF(P_2) \not\models p$. Thus, the above transformation does not preserve the perfect model semantics; put somewhat simply, folding immediately after negative unfolding is not allowed as noted in [14].

Next, we consider the use of the negation technique in unfold/fold transformation of co-LP. We apply the negation technique to $\neg q(X)$ in P_0, obtaining $P_0' = \{1', 2', 3'\}$:

1′. $p \leftarrow not\_q(X)$
2′. $not\_q(X) \leftarrow not\_q(X), not\_r$
3′. $not\_r \leftarrow$
4′. $p \leftarrow not\_q(X), not\_r$ (unfold 1′)
5′. $p \leftarrow p, not\_r$ (fold 4′)

The annotations of $not\_q(X)$ and not_r are coinductive, since the semantics of $q(X)$ and r are given by the least fixpoints in the perfect model semantics. On

1. $state(\mathrm{s0}, [\mathrm{s0}, \mathrm{is1}|T]) \leftarrow enter, work, state(\mathrm{s1}, T)$
2. $state(\mathrm{s1}, [\mathrm{s1}|T]) \leftarrow exit, state(\mathrm{s2}, T)$
3. $state(\mathrm{s2}, [\mathrm{s2}|T]) \leftarrow repeat, state(\mathrm{s0}, T)$
4. $state(\mathrm{s0}, [\mathrm{s0}|T]) \leftarrow error, state(\mathrm{s3}, T)$
5. $state(\mathrm{s3}, [\mathrm{s3}|T]) \leftarrow repeat, state(\mathrm{s0}, T)$
6. $work \leftarrow work$
7. $work \leftarrow$
8. $enter \leftarrow$
9. $exit \leftarrow$
10. $repeat \leftarrow$
11. $error \leftarrow$

Fig. 2. Example: a self-correcting system [22]

the other hand, we consider p a defined predicate, annotated "coinductive" according to the annotation rule (R1). Then, we consider the above transformation sequence: we apply to clause $1'$ unfolding w.r.t. $not\_q(X)$, followed by folding the resulting clause. Let $P_2' = (P_0' \setminus \{1'\}) \cup \{5'\}$. Then, we have that $M(P_2') \models p$. The above transformation thus preserves the alternating fixpoint semantics. □

The above example suggests that the negation technique in co-LP, when used with unfold/fold transformations, would circumvent a restriction imposed on the use of negative unfolding, thereby making amenable to subsequent transformations, which will hopefully lead to a successful proof.

4 Proving Properties of Co-Logic Programs with Negation

In this section, we explain by examples how the negation technique in Sect. 3 will be utilized to prove properties of co-logic programs.

Let P be a co-logic program and *prop* be a predicate specifying a property of interest which is defined in terms of the predicates in P. Then, in order to check whether or not $M(P) \models \exists X\, prop(X)$, our transformation-based verification method is simple: we first introduce a defined predicate f defined by clause δ of the form: $f \leftarrow prop(X)$, where the annotation of predicate f is determined according to the annotation rule in R1. We then apply the transformation rules for co-logic programs given in Sect. 2.2 to $P_0 = P$ as an initial program, constructing a transformation sequence $P_0, P_1 = P_0 \cup \{\delta\}, \ldots, P_n$ so that the truth value of f in $M(P_n)$ will be easily known. In particular, if the definition of f in P_{n-1} consists of a single self-recursive clause $f \leftarrow f$, we will apply the replacement rule to it, obtaining P_n from P_{n-1}, where $Def(f, P_n) = \emptyset$ (i.e., $M(P_n) \models \neg f$) if f is inductive, $Def(f, P_n) = \{f \leftarrow\}$ (i.e., $M(P_n) \models f$) otherwise.

The first example due to [22] is on proving a liveness property of a self-correcting system in Fig. 2.

Example 3. Let P_s be the clauses $1 - 11$ in Fig. 2, which encodes the self correcting system in the figure. The system consists of four states $\mathrm{s0}, \mathrm{s1}, \mathrm{s2}$ and $\mathrm{s3}$. It starts in state s0, enters state s1, performs a finite amount of work in state s1.

This inner loop state in s1 is denoted by is1 (clause 1). The system then exits to state s2. From state s2 the system transitions back to state s0, and repeats the entire loop again, an infinite number of times. However, the system might encounter an error, causing a transition to state s3; corrective action is taken, returning back to s0 (this can also happen infinitely often).

The above system uses two different kinds of loops: an outermost infinite loop and an inner finite loop. The outermost infinite loop is represented by coinductive predicate *state*, while the inner finite loop is represented by inductive predicate *work*. The program P_s satisfies the stratification restriction.

Suppose that we want to prove a property of the system, ensuring that the system must traverse through the work state s2 infinitely often. The counterexamples to the property can be specified as: $\exists T\ state(\mathrm{s0}, T), \neg comember(\mathrm{s2}, T)$, meaning that the state s2 is not present infinitely often in the infinite list T.

To express the counterexample, we first introduce the following clause:

12. $f \leftarrow state(\mathrm{s0}, T), not\_comem(\mathrm{s2}, T)$

where f is an inductive defined predicate, and *not_comem* is an inductive predicate defined in Remark 1. Let P_0 be P_s together with the extended definition of *not_comem*. Then, we can consider the following transformation sequence:

13. $f \leftarrow state(\mathrm{s0}, T'), not\_d(\mathrm{s2}, T')$ (unfold$^+$ 12)
14. $g \leftarrow state(\mathrm{s0}, T), not\_d(\mathrm{s2}, T)$ (def. intro.), $g \in \mathcal{P}^{co}$
15. $f \leftarrow g$ (fold 13)
16. $g \leftarrow state(\mathrm{s0}, T'), not\_d(\mathrm{s2}, T')$ (unfold$^+$ 14)
17. $g \leftarrow g$ (fold 14)
18. $g \leftarrow$ (replacement 17)
19. $f \leftarrow$ (unfold 15)

This means that $M(P_0 \cup \{12, 14\}) \models f$, which implies that there exists a counterexample to the original property; in fact, $T = [\mathrm{s0}, \mathrm{s3}|T]$ satisfies the body of clause 12.

In the previous approach [20], we introduced a predicate $absent(X, T)$ in advance, which corresponds to $\neg comember(X, T)$. Then, we perform a sequence of transformations similar to the above. By contrast, we derive predicate $not\_comem(X, T)$ here by using the negation technique. The current approach is therefore more amenable to automatic proof of properties of co-LP. □

Example 4. Adapted from [16]. We consider regular sets of infinite words over a finite alphabet. These sets are denoted by ω-regular expressions whose syntax is defined as follows:

$e ::= a \mid e_1 e_2 \mid e_1 + e_2 \mid e^\omega$ with $a \in \Sigma$ (regular expressions)

$f ::= e^\omega \mid e_1 e_2^\omega \mid f_1 + f_2$ (ω-regular expressions)

Given a regular (or an ω-regular) expression r, by $\mathcal{L}(r)$ we indicate the set of all words in Σ^* (or Σ^ω, respectively) denoted by r. In particular, given a regular expression e, we have that $\mathcal{L}(e^\omega) = \{w_0 w_1 \cdots \in \Sigma^\omega \mid \text{for } i \geq 0, w_i \in \mathcal{L}(e) \subseteq \Sigma^*\}$.

Now we introduce a co-logic program, called P_f, which defines the predicate ω-*acc* such that for any ω-regular expression f, for any infinite word w, $\omega\text{-}acc(f, w)$ holds iff $w \in \mathcal{L}(f)$. The ω-program P_f consists of the clauses in Fig. 3, together with the clauses defining the predicate *symb*, where $symb(a)$

1. $acc(E, [E]) \leftarrow symb(E)$
2. $acc(E_1E_2, X) \leftarrow$
   $app(X_1, X_2, X), acc(E_1, X_1), acc(E_2, X_2)$
3. $acc(E_1 + E_2, X) \leftarrow acc(E_1, X)$
4. $acc(E_1 + E_2, X) \leftarrow acc(E_2, X)$
5. $acc(E^*, [\,]) \leftarrow$
6. $acc(E^*, X) \leftarrow$
   $app(X_1, X_2, X), acc(E, X_1), acc(E^*, X_2)$
7. $\omega\text{-}acc(F_1 + F_2, X) \leftarrow \omega\text{-}acc(F_1, X)$
8. $\omega\text{-}acc(F_1 + F_2, X) \leftarrow \omega\text{-}acc(F_2, X)$
9. $\omega\text{-}acc(E^\omega, X) \leftarrow$
   $app(X_1, X_2, X), acc(E, X_1), \omega\text{-}acc(E^\omega, X_2)$
10. $app([\,], Y, Y) \leftarrow$
11. $app([S|X], Y, [S|Z]) \leftarrow app(X, Y, Z)$

Fig. 3. Example: A Program P_f Which Accepts ω-languages

holds iff $a \in \Sigma$. Clauses 1-6 specify that, for any finite word w and regular expression e, $acc(e, w)$ holds iff $w \in \mathcal{L}(e)$. Similarly, clauses 7-9 specify that, for any infinite word w and ω-regular expression f, $\omega\text{-}acc(f, w)$ holds iff $w \in \mathcal{L}(f)$. The annotation of $\omega\text{-}acc$ is coinductive, while the other predicates are inductive.

Now, let us consider the ω-regular expressions $f_1 \equiv_{def} a^\omega$ and $f_2 \equiv_{def} (b^*a)^\omega$. The following two clauses:
12. $expr_1(X) \leftarrow \omega\text{-}acc(a^\omega, X)$
13. $expr_2(X) \leftarrow \omega\text{-}acc((b^*a)^\omega, X)$
together with program P_f, define the predicates $expr_i$ $(i = 1, 2)$ such that, for every infinite word w, $expr_i(w)$ holds iff $w \in \mathcal{L}(f_i)$. Moreover, we introduce predicate $not\_expr_2(X)$ defined as: for any ground term t,
14. $M(P_f \cup \{13\}) \models \neg expr_2(t)$ iff $M(P_f \cup \{13\}) \models not\_expr_2(t)$.
If we introduce a clause:
15. $not\_contained(X) \leftarrow expr_1(X), not\_expr_2(X)$,
then we have that $\mathcal{L}(f_1) \subseteq \mathcal{L}(f_2)$ iff $M(P_f \cup \{12, 13, 14, 15\}) \not\models \exists X\, not\_contained(X)$.
We note that the negation technique is not applicable to $\omega\text{-}acc(E^\omega, T)$; clause 9 has existential variables and, unlike the previous example, it does not have a functional part. The definition of $not\_\omega\text{-}acc$ is therefore not given in an explicit clausal form at this moment. Instead, we will apply the negation technique 'on the fly' in the following.

To check the above containment, we introduce the following clause:
16. $g \leftarrow not\_contained(X)$
where g is an inductive predicate. We can then consider the following transformation sequence:
17. $g \leftarrow \omega\text{-}acc(a^\omega, X), not\_\omega\text{-}acc((b^*a)^\omega, X)$ (unfold$^+$ 16)
18. $g_1 \leftarrow \omega\text{-}acc(a^\omega, X), not\_\omega\text{-}acc((b^*a)^\omega, X)$ (def. intro.), $g \in \mathcal{P}^{in}$
19. $g \leftarrow g_1$ (fold 17)
20. $g_1 \leftarrow \omega\text{-}acc(a^\omega, T), not\_\omega\text{-}acc((b^*a)^\omega, [a|T])$ (unfold$^+$ 18)

From P_f and a goal $\omega\text{-}acc((b^*a)^\omega, [a|T])$, we get the following resultant [11]:
$\omega\text{-}acc((b^*a)^\omega, [a|T]) \leftarrow \omega\text{-}acc((b^*a)^\omega, T)$.
Since the above clause does not contain an existential variable, we have now from the negation technique:
21. $not\_\omega\text{-}acc((b^*a)^\omega, [a|T]) \leftarrow not\_\omega\text{-}acc((b^*a)^\omega, T)$

Using the above clause, we continue the above-mentioned transformation:

22. $g_1 \leftarrow \omega\text{-}acc(a^\omega, T), not\_\omega\text{-}acc((b^*a)^\omega, T)$ (unfold 20)
23. $g_1 \leftarrow g_1$ (fold 22)
24. def. of g_1 (clause 23) removed (replacement 23)
25. clause 19 removed (unfold 19)

This means that $M(P_f \cup \{12, 13, 14, 15\}) \not\models g$, namely $\mathcal{L}(f_1) \subseteq \mathcal{L}(f_2)$.

The above example is originally due to Pettorossi, Proietti and Senni [16], where (i) the given problem is encoded in an *ω-program* P, a locally stratified program on infinite lists, then (ii) their transformation rules in [16] are applied to P, deriving a *monadic* ω-program T, and finally (iii) the decision procedure in [15] is applied to T to check whether or not $PERF(T) \models \exists X\, prop(X)$.

On the other hand, our approach uses co-LP, which allows us to make the representation more succinct, about the half the lines of the ω-program in this particular example. □

5 Related Work and Concluding Remarks

We have shown that the negation technique (NT) can be used for co-logic programs, and its application is correct under the alternating fixpoint semantics of co-logic programs. In co-LP, NT allows us to derive a complementary program under a weaker condition than in the original NT for definite programs. We have explained in Sect. 4 how NT, when incorporated into the previous framework for unfold/fold transformation, allows us to represent and reason about a wider class of co-logic programs which our previous framework [20] cannot deal with.

Simon et al. [22] have proposed an operational semantics, co-SLD resolution, for co-LP, which has been further extended by Min and Gupta [12] to co-SLDNF resolution. On the other hand, as explained in Example 3, NT derives a program which simulates failed computations of a given program. When NT is applicable, our approach is simpler than co-SLDNF resolution in that we can do without an extra controlling mechanism such as a positive/negative context in co-SLDNF resolution.

Pettorossi, Proietti and Senni [16] have proposed another framework for transformation-based verification based on ω-programs, which can also represent infinite computations. As explained in Example 4, our approach can prove the given property in a simpler and more succinct manner, as far as this particular example is concerned. However, the detailed comparison between our approach based on co-logic programs and their approach based on ω-programs is left for future work.

One direction for future work is to extend the current framework to allow a more general class of co-logic programs. Gupta et al. [6], for example, have discussed such an extension, where they consider co-LP without the stratification restriction.

Acknowledgement The author would like to thank anonymous reviewers for their constructive and useful comments on the previous version of the paper.

References

1. Burstall, R. M., and Darlington, J., A Transformation System for Developing Recursive Programs, *J. ACM*, 24, 1, pp. 144-67, 1977.
2. Clarke, E. M., Grumberg, O. and Peled, D. A., *Model Checking*, MIT Press, 1999.
3. Colmerauer, A., Prolog and Infinite Trees, Logic Programming, Academic Press, (1982), pp. 231-251.
4. Etalle, S., Gabbrielli, M., Transformations of CLP Modules. *Theor. Comput. Sci.* pp. 101-146, 1996.
5. Gupta, G., Bansal, A., Min, R., Simon, L., and Mallya, A., Coinductive logic programming and its applications, ICLP'07, LNCS 4670, pp. 27-44, 2007.
6. Gupta, G., Saeedloei, N. et al., Infinite computation, co-induction and computational logic, CALCO'11, pp. 40–54, Springer-Verlag, 2011.
7. Jaffar, J., Stuckey, P., Semantics of infinite tree logic programming, *Theoretical Computer Science*, 46, pp. 141-158, 1986.
8. Jaffar, J., and Maher, M, J., Constraint Logic Programming: A Survey, *J. Log. Program.*, 19/20, pp. 503-581, 1994.
9. Kott, L., Unfold/fold program transformations, *Algebraic Methods in Semantics*, pp. 411-434. Cambridge University Press, 1985.
10. Lloyd, J. W., *Foundations of Logic Programming*, Springer, 1987, Second edition.
11. Lloyd, J. W. and Shepherdson, J. C., Partial Evaluation in Logic Programming, *J. Logic Programming*, 11:217-242, 1991.
12. Min, R. and Gupta, G., Coinductive Logic Programming with Negation, *Proc. LOPSTR'09*, LNCS 6037, pp. 97-112, 2010.
13. Pettorossi, A. and Proietti, M., Transformation of Logic Programs: Foundations and Techniques, *J. Logic Programming*, 19/20:261-320, 1994.
14. Pettorossi, A. and Proietti, M., Perfect Model Checking via Unfold/Fold Transformations, *Proc. CL2000*, LNAI 1861, pp. 613-628, 2000.
15. Pettorossi, A., Proietti, M. and Senni, V., Deciding Full Branching Time Logic by Program Transformations, *Proc. LOPSTR'09*, LNCS 6037, pp. 5-21, 2010.
16. Pettorossi, A., Proietti, M. and Senni, V., Transformations of logic programs on infinite lists, *Theory and Practice of Logic Programming*, 10, pp. 383-399, 2010.
17. Przymusinski, T.C., On the Declarative and Procedural Semantics of Logic Programs. *J. Automated Reasoning*, 5(2):167–205, 1989.
18. Sato, T., and Tamaki, H., Transformational Logic Program Synthesis, Proc. FGCS'84, Tokyo 1984, pp. 195-201.
19. Seki, H., On Inductive and Coinductive Proofs via Unfold/fold Transformations, *Proc. LOPSTR'09*, LNCS 6037, pp. 82-96, 2010.
20. Seki, H., Proving Properties of Co-logic Programs by Unfold/Fold Transformations, *Proc. LOPSTR'11*, LNCS 7225, pp. 205-220, 2012.
21. Seki, H., Unfold/Fold Transformation of Stratified Programs, *Theoretical Computer Science* 86, 107-139, 1991.
22. Simon, L., Mallya, A., Bansal, A., Gupta, G., Coinductive Logic Programming, *Proc. ICLP'06*, LNCS 4079, pp. 330-344, 2006.
23. Simon, L. E., Extending Logic Programming with Coinduction, Ph.D. Dissertation, University of Texas at Dallas, 2006.
24. Tamaki, H. and Sato, T., Unfold/Fold Transformation of Logic Programs, *Proc. 2nd Int. Conf. on Logic Programming*, 127-138, 1984.

Programming in Logic without Prolog

M.H. van Emden

Department of Computer Science
University of Victoria, Canada
vanemden@cs.uvic.ca

Abstract. Logic can be made useful for programming and for databases independently of logic programming. To be useful in this way, logic has to provide a mechanism for the definition of new functions and new relations on the basis of those given in the interpretation of a logical theory. We provide this mechanism by creating a compositional semantics on top of the classical semantics. In this approach, verification of computational results relies on a correspondence between logic interpretations and a class definition in languages like Java or `C++` . The advantage of this approach is the combination of an expressive medium for the programmer with, in the case of `C++` , optimal use of computer resources.

Keywords First-order predicate logic, compositional semantics, relations, recursive definitions, object-oriented programming languages.

1 Introduction

"What can logic do for programming?" This question could have been asked as early as 1950 when it was first noticed that valuable time on one of the few computers had been wasted due to a programming error. The question has not been used as a starting point in the development of programming languages. When logic was connected for the first time with programming it was in the form of logic programming, arising as a special form of automatic theorem-proving with the resolution inference rule. Given that many alternatives need to be excluded before arriving at logic programming, it is likely that it is not the whole answer to the question of what logic can do for programming. To make sure that we don't prematurely exclude interesting possibilities, let us see how far we can get without any inference rule. On the other hand, in this paper we do not try to do everything, so we restrict the scope of "logic" to mean first-order predicate logic.

A good starting point for the use of logic for programming is that logic formulas have symbols that are interpreted as functions or as relations. On the programming side we note that code is organized into subroutines, which can take the form of procedures or function subroutines. A desirable property of subroutines is that they are free from side-effects: that function subroutines only interact with their environment by delivering a result and procedures only do so by modifying one or more parameters. Our approach to programming in logic relies on a correspondence between such side effect-free subroutines on the one hand and functions or relations in interpretations of logic theories on the other.

Most of a programmer's work consists of writing new subroutines in terms of existing ones. To be useful for programming, logic has to provide a mechanism for defining new functions and relations in terms of existing ones. It may seem that the semantics of logic does not provide such a mechanism. However, this problem is solved by reformulating the classical semantics as *compositional semantics*. In compositional semantics the meaning of a composite syntactic construct is defined as a combination of the meanings of the constituent parts of the construct. In this way we obtain a meaning assigned to a term with variables. This meaning is a function. We also obtain a meaning assigned to formula with free variables. This meaning is a relation. Compositional semantics makes any term and formula with (free) variables available for the definition of a new function or relation.

What do we do with this definition mechanism? To go beyond a theoretical exercise there needs to be a way to get a computer program verified in some way by a specification in logic.

Let us consider interpretations for a theory of logic. Classical examples are theories for commonly used structures of algebra, such as monoids, groups, rings, and fields. For each of these, the axioms are formalized as a theory of logic. A given algebraic structure is then by definition a monoid, group, ring, or field according to whether it satisfies, in the sense of model theory, the defining logical theory. Although these theories and this methodology were established before the birth of computers, it is a topic of current research to establish a link between them and efficient implementations of algorithms [3].

An interpretation for a theory T consists of a universe of discourse D, a function over D as interpretation for every function symbol in T, and a relation over D for every relation symbol in T. A properly structured program consists almost entirely of definitions of function subroutines and of procedures. In a language like Java or `C++` *classes* are used to organize function subroutines and procedures into meaningful groups. This suggests to us that the desired connection between logic and the operation of a computer can be made by writing a class in such a way that it is sufficiently similar to an interpretation of a logic theory that the behaviour of the (side-effect free) subroutines of the class is decribed by the corresponding function or relation symbol of the theory. Although classes are used, this is not object-oriented programming, where side-effects of subroutines are not merely regrettable lapses, but constitute the essence.

To illustrate how this correspondence between a logical theory and a class can be used consider the following example. Let the theory consist of the axioms for Euclidean domains in the sense of abstract algebra. The constants are 0 and 1 for the additive and multiplicative identities. The function symbols are those for addition, multiplication, and subtraction. The integers are the most familiar example. Especially interesting are the ones leading to a finite universe of discourse, such as the modular numbers. To be specific, consider a `C++` class that emulates an interpretation for the axioms for Euclidean domains by having instances named `zero` and `one` and subroutines for addition, multiplication, and subtraction. Let us add to the theory for Euclidean domains a relation symbol

gcd such that $gcd(a, b, c)$ holds iff c is the greatest common denominator of a and b. The definition can be made to correspond to a `C++` procedure using an implementation of the `C++` class for Euclidean domains. Such a procedure is a program for a computer on which `C++` is implemented. Because the procedure relies on the `C++` class, the correctness of the implementations of addition, multiplication, and subtraction of the Euclidean domain class imply correctness of the implementation of *gcd*.

Plan of the paper As preparation for compositional semantics of logic, and to establish terminology and notation, we begin with two reviews: material on relations in Section 2; on model-theoretic semantics in Section 3. Much of this material is standard, but some concepts, such as the quotient of one function by another, are rarely found in the literature. This particular one is crucial to the paper. The basic observation underlying our definitions of relations is that any formula defines a relation as the set of tuples that, when assigned to the tuple of free variables, makes the formula true, given a fixed interpretation of function and relation symbols. In this way formulas denote relations; we call this the *compositional semantics of logic*. It is the topic of Section 3.2. In Section 4 we apply the compositional semantics to the introduction of new function and relation symbols and define their interpretations. This section does not treat the recursive case, which is the topic of Section 5. In Section 6 we discuss the design of a `C++` class so that it can be regarded as a specification of an interpretation for a given theory.

2 Notation and terminology for functions and relations

Not all authoritative texts agree on the notations of set theory needed in this paper. We also need some concepts that are usually not covered in introductions like the present one. Therefore we collect in this section the necessary material, terminology and notation.

Functions The set of functions that take arguments in a set S and have values in a set T is denoted $S \to T$. This set is said to be the *type* of a function $f \in (S \to T)$. We write $f(a)$ for the element of T that is the value of f for argument $a \in S$; $f(a)$ may also be written as f_a. If an expression E with a single free variable x is used to define the values of $f \in (S \to T)$, then the mapping of f is $(a \in S) \mapsto E[a]$.

If $S' \subseteq S$, then the *projection* (or *restriction*) $f{\downarrow}S'$ of $f \in S \to T$ on S' is the function in $S' \to T$ such that $(f{\downarrow}S')(a) = f(a)$ for all $a \in S'$.

Suppose we have $f \in S \to T$ and $g \in T \to U$. Then the composition $g \circ f$ of f and g is the function $h \in S \to U$ defined by $x \mapsto g(f(x))$. Suppose now that we are given $f \in S \to T$ and $h \in S \to U$, is there a $g \in T \to U$ such that $h = g \circ f$? The answer, depending on f and h, may be that there is no such g, or one, or more than one. We therefore define h/f, the quotient of h by f, to be $\{g \in T \to U \mid f \circ g = h\}$.

Tuples Often an n-tuple over a set D is thought of as an object $(d_0, \ldots, d_{n-1})$ in which an element of D is associated with each of the indexes $0, \ldots, n-1$. It is convenient to view such a d as a function of type $\{0, \ldots, n-1\} \to D$. This formulation allows us to consider tuples of which the index set is a set other than $\{0, \ldots, n-1\}$. Hence we define a tuple as an element of the function set $I \to T$, where I is an arbitrary countable set to serve as index set. $I \to T$ is the *type* of the tuple.

Example If t is tuple in $\{x, y, z\} \to R$, then we may have $t_x = 1.1$, $t_y = 1.21$, and $t_z = 1.331$. A more compact notation would be welcome; we use $t = \begin{array}{c|c|c} x & y & z \\ \hline 1.1 & 1.21 & 1.331 \end{array}$, where the order of columns is immaterial.

Example $t \in \{0, 1, 2\} \to \{a, b, c\}$, where $t = \begin{array}{c|c|c} 2 & 1 & 0 \\ \hline c & c & b \end{array}$. In cases like this, where the index set is of the form $\{0, \ldots, n-1\}$, we use the compact notation $t = (b, c, c)$, using the conventional order of the index set.

Relations A relation is a set of tuples with the same type. This type is the *type* of the relation.

Example
$sum = \{(x, y, z) \in (\{0, 1, 2\} \to \mathcal{R}) \mid x + y = z\}$ is a relation of type $\{0, 1, 2\} \to \mathcal{R}$. Compare this relation to the relation
$\sigma = \{s \in (\{x, y, z\} \to \mathcal{R}) \mid s_x + s_y = s_z\}$. As their types are different, they are different relations; $(2, 2, 4) \in sum$ is not the same tuple as $s \in \sigma$ where $s = \begin{array}{c|c|c} x & y & z \\ \hline 2 & 2 & 4 \end{array}$.

Definition 1. *If r is a relation with type $I \to T$, then the* projection $\pi_{I'}(r)$ *of r on $I' \subseteq I$ is*

$$\{f' \in I' \to T \mid \exists f \in r.\ (f{\downarrow}I') = f'\}.$$

If r_0 and r_1 are relations with types $I_0 \to T$ and $I_1 \to T$, respectively, then the join $r_0 \bowtie r_1$ *of r_0 and r_1 is*

$$\{f \in (I_0 \cup I_1) \to T \mid (f{\downarrow}I_0) \in r_0 \text{ and } (f{\downarrow}I_1) \in r_1\}.$$

Definition 2. *Let $H \subseteq (S \to U)$ be a relation and let $f \in S \to T$ be a tuple. Then the quotient H/f of H by f is defined as the relation $\cup\{h/f \mid h \in H\}$ of type $T \to U$.*

Example With $S = \{0, 1, 2\}$, $U = \mathcal{R}$, $T = \{x, y\}$, $sum = \{h \in \{0, 1, 2\} \to \mathcal{R} \mid h_0 + h_1 = h_2\}$, and $f = (x, x, y)$ we have

$$sum/(x, x, y) = \{(s \in \{x, y\}) \to \mathcal{R} \mid s_y = 2s_x\}.$$

Here quotient on relations is used to define on the basis of the *sum* relation the relation indexed by $\{x, y\}$ in which the argument indexed by y is double the one indexed by x.

3 Semantics for first-order predicate logic

Conventional semantics is primarily concerned with the justification of inference systems. The use of predicate logic for the definition of functions or relations is secondary, if considered at all. As a result, conventional semantics centres around the concept of *satisfaction*: under what conditions is a formula satisfied by a given interpretation of the relation symbols and constants under a given assignment of individuals to the variables. Because of the emphasis on satisfaction, we refer to this kind of semantics as *satisfaction semantics*. Compositional semantics, in contrast with satisfaction semantics, defines the meaning of a complex term or formula as a composition of the meanings of the constituent terms or formulas.

3.1 Satisfaction semantics

Our language of logical formulas is determined by a set V of *variables*, a set F of *function symbols*, and a set R of *relation symbols*. The role constant symbols is played by 0-ary function symbols.

To avoid lexical details we give the syntax in an abstract form.

A *term* is a variable, a constant symbol, or an expression consisting of a k-ary function symbol and a tuple of k terms.

An *atom* (or *atomic formula*) is an expression consisting of a k-ary relation symbol and a tuple of k terms.

A *conjunction* is a formula consisting of a set of formulas.

An *existential quantification* is a formula consisting of a variable and a formula.

A *negation* is a pair consisting of a formula and an indication that the pair is a negation.

An *interpretation* M for the language consists of a set D called the *universe of discourse* (with elements called *individuals*) of the interpretation, a function that maps every n-ary function symbol in F to a function of type $D^n \to D$, and a function that maps every n-ary relation symbol in R to a subset $M(p)$ of D^n.

The interpretation M is extended to assign the meaning $M(t)$ to every variable-free term t and extended to determine whether a variable-free formula is true.

We first define the meaning of variable-free atoms.

- $M(f(t_0, \ldots, t_{n-1})) = (M(f))(M(t_0), \ldots, M(t_{n-1})))$.
- A variable-free atom $p(t_0, \ldots, t_{k-1})$ is satisfied by an interpretation iff $(M(t_0), \ldots, M(t_{k-1})) \in M(p)$.
- A *conjunction* $\{F_0, \ldots, F_{n-1}\}$ of variable-free formulas is satisfied by M iff F_i is satisfied by M, for all $i \in \{0, \ldots, n-1\}$.
- A variable-free formula that is the negation of F is satisfied by M iff F is not satisfied by M.

We now consider meanings of formulas that contain variables. Let A be an *assignment*, which is a function in $V \to D$, assigning an individual in D to every variable. In other words, A is a tuple of elements of D indexed by V. As meanings of terms with variables depend on A, we write M_A for the function mapping a term to a domain element. M_A is defined as follows.

- $M_A(t) = A(t)$ if t is a variable
- $M_A(c) = M(c)$ if c is a constant
- $M_A(f(t_0, \ldots, t_{n-1})) = (M(f))(M_A(t_0), \ldots, M_A(t_{n-1}))$.
- $p(t_0, \ldots, t_{k-1})$ is satisfied by M with A iff $(M_A(t_0), \ldots, M_A(t_{k-1})) \in M(p)$

Now that satisfaction of atoms is defined, we can continue with:

- $\{F_0, \cdots, F_{n-1}\}$ is satisfied by M with A iff the formulas F_i are satisfied by M with A, for all $i = 0, \ldots, n-1$.
- If F is a formula, then $\exists x.F$ is satisfied by M and A iff there is a $d \in D$ such that F is satisfied with M with $A_{x|d}$ where $A_{x|d}$ is an assignment that maps x to d and maps the other variables according to A.
- $\neg F$ is satisfied by M with A iff formula F is not thus satisfied.

The meaning of formulas contain disjunction, implication, or existential quantification is obtained by eliminating these according to the usual rules.

3.2 Compositional semantics for first-order predicate logic

The conventional semantics for first-order predicate logic focuses on the conditions under which a sentence, that is, a variable-free formula, is satisfied by an interpretation for constants, function symbols, and relation symbols. Yet terms with variables and formulas with free variables are potentially definitions of new functions and relations defined in terms of existing ones.

Definition 3. *Let t be a term with set V of variables, and let M be an interpretation for the theory in which the term occurs. $M(t)$ is a function of type $(V \to D) \to D$ that maps $A \in (V \to D)$ to $M_A(t)$.*

Example $t = x^2 + 2y^2 + 3z^2$, $V = \{x, y, z\}$, $D = \mathcal{R}$, $M(t)$ is the function with map

$$(A \in (V \to \{x, y, z\})) \mapsto (M_A(t) \in \mathcal{R}).$$

e.g. $M(t)(\frac{x|y|z}{3|2|1}) = 3^2 + 2 \cdot 2^2 + 3 \cdot 1^2 = 20.$

Following Cartwright [2]:

Definition 4. *Let V be the set of the free variables in the formula $p(t_0, \ldots, t_{n-1})$. We extend M to atomic formulas containing variables by defining $M(p(t_0, \ldots, t_{n-1}))$ to be*

$$\{A \in V \to D \mid (M_A(t_0), \ldots, M_A(t_{n-1})) \in M(p)\}.$$

According to this definition, a closed formula denotes a relation consisting of tuples of length 0. As there is only one such tuple, there are only two such relations, each of which is identified with one of the two truth values. With this understanding, the following definition generalizes the conventional one for logical implication.

Definition 5. *Let formulas φ_0 and φ_1 have the same set of free variables and admit of the same interpretations. We define $\varphi_0 \models \varphi_1$ to mean that $M(\varphi_0) \subseteq M(\varphi_1)$ for all interpretations M.*

In the previous section we used the conventional semantics, which determines under what conditions a sentence is satisfied by an interpretation for the relation symbols and constants, to define a semantics that extends the meaning function M from from relation symbols to atomic formulas with free variables. According to this extended semantics every atomic formula with set V of free variables denotes a relation of type $V \to D$.

For a semantics to be compositional it is necessary that the meaning of a composite formula is a composition of the relations that are the meanings of its constituent formulas. Accordingly we define in this section the compositional semantics of conjunctions, negations and existentially quantified formulas in terms of the relations denoted by their constituent formulas. And although we have already given, in Definition 4, a relational semantics for an atomic formula that may have free variables, this semantics is not compositional. For this it is necessary that we specify what operation on $M(p)$ gives the $M(p(t_0, \ldots, t_{n-1}))$ of Definition 4.

Theorem 1. *If $t_0, \ldots, t_{n-1}$ are variables, then we have $M(p(t_0, \ldots, t_{n-1})) = M(p)/(t_0, \ldots, t_{n-1})$.*

Proof. Let A be such that $a = A{\downarrow}V$.
a in the left-hand side $\Leftrightarrow$ (Definition 4)
$p(t_0, \ldots, t_{k-1})$ is satisfied by M with $A \Leftrightarrow$ (use satisfaction)
$(a(t_0), \ldots, a(t_{k-1})) \in M(p) \Leftrightarrow$ (use $f = a \circ t$)
$(f_0, \ldots, f_{k-1}) \in M(p) \Leftrightarrow$ (use definition of / (quotient))
$a \in M(p)/(t_0, \ldots, t_{k-1})$.

Example

$$\begin{aligned} M(sum(x, x, y)) &= \{s \in x, y \to \mathcal{R} \mid 2s_x = s_y\} \\ &= \{t \in \{0, 1, 2\} \to \mathcal{R} \mid t_0 + t_1 = t_2\}/(x, x, y) \\ &= M(sum)/(x, x, y) \end{aligned}$$

The first equality arises by Definition 4. The second equality arises by Definition 2. The third equality arises by the meaning of *sum* in the assumed interpretation of the relation symbol.

Theorem 2. *For any formulas $\varphi_0, \ldots, \varphi_{k-1}$ we have*

$$M(\varphi_0 \wedge \cdots \wedge \varphi_{k-1}) = M(\varphi_0) \bowtie \cdots \bowtie M(\varphi_{k-1})$$

Theorem 3. *Let φ be a formula with V as its set of free variables, and $W = \{w_0, \ldots, w_{k-1}\}$ a be subset of V. Then we have*

$$M(\exists w_0 \ldots w_{k-1}.\varphi) = \pi_{V \setminus W}(M(\varphi))$$

Theorem 4. *Let φ be a formula with V as set of free variables. Then $M(\neg\varphi)$ is the complement in $V \to D$ of $M(\varphi)$.*

4 Extensions of theories

Most subroutines call subroutines. This means that much of a programmer's activity consists of defining a new subroutine in terms of existing ones. What a programmer should look for in logic is the possibility of defining new functions and relations in terms of existing ones. In this section we use the compositional semantics developed earlier as the basis of such a definition mechanism.

Definition of functions Let t be a term with V as set of variables. It can be a complex term, deeply nested, with many occurrences of function symbols. Its subterms can share variables in intricate patterns. This richness of expression makes it attractive for a programmer to encapsulate such a complex term by making its denotation the interpretation of a new function symbol f. A candidate for the interpretation of such an f is $M(t)$.

But suppose that we interpret f by $M(t)$, how do we then interpret the term $f(t_0, \ldots, t_{n-1})$ when the interpretation gives $a_0, \ldots, a_{n-1} \in D$ as values for $t_0, \ldots, t_{n-1}$? How do the n individuals $a_0, \ldots, a_{n-1}$ find their way to the corresponding n variables in $t_0, \ldots, t_{n-1}$? The difficulty here is that $M(t)$ is a function of type $(V \to D) \to D$, whereas f needs to be interpreted by a function of type $(\{0, \ldots, n-1\} \to D) \to D$. The difficulty is resolved in the following definition.

Definition 6. *Let T be a theory of first-order predicate logic not containing a function symbol f. Let M be an interpretation for T and let t with set V of variables be a term of T. The* extension of T by f *is a theory T' with function symbol f and otherwise identical to T. The* extension of M by f and t *is an interpretation M' that is identical to M except that M' assigns to f the function of type $(\{0, \ldots, n-1\} \to D) \to D$ with map $(a_0, \ldots, a_{n-1}) \mapsto M_A(t)$ (footnote[1]) where*

$$A = (a_0, \ldots, a_{n-1}) \circ (x_0, \ldots, x_{n-1})^{-1} = \frac{x_0 | \ldots | x_{n-1}}{a_0 | \ldots | a_{n-1}}$$

and $(x_0, \ldots, x_{n-1})$ (see footnote[2]) is some enumeration of V.

[1] See Definition 3 for the meaning of $M_A(t)$.

[2] This tuple is a function of type $\{0, \ldots, n-1\} \to V$. The inverse $(x_0, \ldots, x_{n-1})^{-1}$ exists because all variables in the tuple are different.

Example $t = x^2 + 2y^2 + 3z^2$, $V = \{x, y, z\}$, $D = \mathcal{R}$.

With $A = (3,2,1) \circ (x,y,z)^{-1} = \dfrac{x|y|z}{3|2|1} \in \{x,y,z\} \to \mathcal{R}$ we get $f(3,2,1) = M'_A(t) = 3^2 + 2 \cdot 2^2 + 3 \cdot 1^2 = 20$.

With a different enumeration of the variables we get a different function. E.g. $A = (3,2,1) \circ (y,z,x)^{-1} = \dfrac{x|y|z}{1|3|2} \in \{x,y,z\} \to \mathcal{R}$ we get $f(3,2,1) = M'_A(t) = 1^2 + 2 \cdot 3^2 + 3 \cdot 2^2 = 31$.

Definition 6 is a semantic one, so has no commitment to any particular syntax. Syntax for the definition is only of secondary concern in this paper. However, we do want to remark here that specifying the extensions to T and M by writing

$$f \text{ def } \lambda(x_0, \ldots, x_{n-1}).\ t \tag{1}$$

carries the necessary information. The use of lambda suggests the intent of the definition that to evaluate $f(t_0, \ldots, t_{n-1})$ one has to pair the x_i in t with the values of the t_i. However, this has no formal connection to lambda calculus: the meaning of (1) is determined by the interpretation for f specified in Definition 6.

Iterated extensions of interpretations for new function symbols When an interpretation has been extended by new function symbols, one can repeat a similar process where new function symbols are defined in terms of the function symbols that have already been introduced. This is natural from a programming point of view: a function subroutine often contains calls to subroutines defined in the same program.

We therefore define the *iterated extension of order* n for $n = 0, 1, 2, \ldots$

- The iterated extension of order 0 is the extension according to Definition 6.
- The iterated extension of order $n > 0$ is the extension according to Definition 6 when the interpretation M has incorporated all iterated extensions of orders $0, \ldots, n-1$.

Definition of relations Let F be a formula with V as set of free variables. Then $M(F)$ is a relation of type $V \to D$. F can be a formula with quantifications nested arbitrarily deeply, with many relation and function symbols. Such complexity, together with intricate patterns of shared variables within the same scope give the programmer a powerfully expressive tool for the definition of new relations out of existing ones.

However, a new relation symbol p needs to be interpreted by a relation of type $\{0, \ldots, n-1\} \to D$, where n is the number of free variables in F. Theorem 1 suggests how to resolve the type mismatch.

Definition 7. *Let T be a theory of first-order predicate logic not containing the relation symbol p. Let M be an interpretation for T and let F be a formula of T. The* extension of T by p *is a theory T' with relation symbol p and otherwise identical to T. The* extension of M by p and F *is an interpretation M' that*

is identical to M except that M' assigns to p the relation $M(F)/(x_0, \ldots, x_{n-1})$ where $(x_0, \ldots, x_{n-1})$ is some enumeration of the variables in F.

Definition 7 is a semantic one, so has no commitment to any particular syntax. Syntax for the definition is only of secondary concern in this paper. However, we do want to remark here that specifying the extensions to T and M by writing

$$p \text{ def } \lambda(x_0, \ldots, x_{n-1}).\ F \tag{2}$$

carries the necessary information. The use of lambda suggests the intent of the definition that to determine the truth value of $p(y_0, \ldots, y_{n-1})$ one has to pair the x_i with the $M(y_i)$. However, this has no formal connection to lambda calculus: the meaning of (2) is determined by $M(F)/(x_0, \ldots, x_{n-1})$ being the interpretation for p.

Example Let $M(\text{sum}) = \{h \in \{0,1,2\} \to \mathcal{R} \mid h_0 + h_1 = h_2\}$, $V = \{x, y\}$, and $F = \text{sum}(x, x, y)$. With p def $\lambda(x, y).\ F$ and q def $\lambda(y, x).\ F$ we get e.g.
$p(6,3)$ iff $M(sum(x,x,y)/(x,y))(6,3)$ iff $M(sum(6,6,3))$ iff false
$q(6,3)$ iff $M(sum(x,x,y)/(y,x))(6,3)$ iff $M(sum(3,3,6))$ iff true

5 Recursively defined extensions

So far we have assumed that the introduced function or relation symbol does not occur in the defining term or formula. If we allow the introduced function symbol to occur in the defining term we allow the possibility of the resulting function to be partial. As we stay within classical first-order predicate logic, where functions are total, we impose the restriction that the definiendum cannot occur in the definiens.

However, in first-order logic, relation symbols are interpreted by relations. As the interpretation of p with n arguments can be any subset of $\{0, \ldots, n-1\} \to D$, including the empty subset, no such obstacle exists for the definition of new relation symbols. In this section we consider the case where the definition is recursive in the sense of the defining formulas containing new relation symbols.

Because of the absence of recursive definitions of new functions we can suppose all new function symbols introduced by extensions of all orders to have been replaced by their definition before considering the semantics of the recursive definitions of the relations. This is only necessary for theoretical purposes; in practice one leaves the function definitions in place to have the advantage of a compact theory.

Definition 8. *Among interpretation extensions with fixed function interpretations, an interpretation M is included in M' iff $M(p) \subseteq M'(p)$ for all new relation symbols p. A mapping $\mathcal{T}$ from the set of interpretations to itself is said to be* monotonic *iff M is included in M' implies that $\mathcal{T}(M)$ is included in $\mathcal{T}(M')$, where M and M' have the same interpretation for their function symbols.*

Thus we find that the desire to stay within first-order predicate logic suggests unrestricted definitions of new relations based on a fixed repertoire of given

relations and given functions. The use of logic proposed here has not proposed any inference system. Yet we find something in common with logic programming, where only relations are defined by the program and the function symbols have fixed interpretations. A difference is that logic programming also fixes the universe of discourse to be the Herbrand universe. Here the fixed interpretations for function symbols are freely chosen functions over arbitrary universes of discourse.

We consider simultaneous definitions p_i def $\lambda(x_{i,0}, \ldots, x_{i,m_i-1}).F_i$. How to extend a given theory with these relation symbols? According to Definition 7 the extended interpretation assigns to the new relation symbols $p_0, \ldots, p_{n-1}$ the relations $M(p_0), \ldots, M(p_{n-1})$ that satisfy the equations

$$\begin{aligned} M(p_0) &= M(F_0)/(x_{0,0}, \ldots, x_{0,m_0-1}) \\ \cdots &= \cdots \\ M(p_{n-1}) &= M(F_{n-1})/(x_{n-1,0}, \ldots, x_{n-1,m_{n-1-1}}) \end{aligned} \tag{3}$$

The variables are local to each of the right-hand sides separately. One can see this by observing that a systematic renaming of the variables in a right-hand side does not change the meaning of that expression.

In general we cannot say anything about existence and uniqueness of solutions. Let us consider one example of a condition on $F_0, \ldots, F_{n-1}$ that ensures a unique solution: that these formulas are existentially quantified conjunctions of atomic formulas with the right sets of free variables. When such formulas are translated to clausal form they are right-hand sides of Horn clauses. Let us call such formulas "Horn formulas", even though they do not represent Horn clauses in their most general form.

Theorem 5. *The equations in (3) have a unique least solution if $F_0, \ldots, F_{n-1}$ are Horn formulas.*

Proof. Because the formulas in Equation 3 are Horn formulas, the right-hand sides in (3) constitute a monotonic mapping on the set of interpretations with fixed function interpretations. The monotonicity implies that (3) has a unique least solution.

Example: Euclid's algorithm in a Euclidean domain
The logical theory has constants *zero* and *unit*, binary function symbols $+$ and $*$, and binary relation symbol $<$. We extend the theory by the definition *gcd* def $\lambda(x, y, z).F_0$ where F_0 is the Horn formula

$$\begin{aligned} &gcd(x, y, z) \leftarrow x < y \wedge gcd(x, y - x, z) \wedge \\ &gcd(x, y, z) \leftarrow y < x \wedge gcd(x - y, y, z) \wedge \\ &gcd(x, y, z) \leftarrow y = x \wedge z = x \end{aligned} \tag{4}$$

where we write $A \leftarrow B$ for $A \vee \neg B$. The meaning of *gcd* is given as the least solution of (3) where $n = 1$, p_0 is *gcd* and $\{x_{0,0}, \ldots, x_{0,m_0-1}\}$ is $\{x, y, z\}$.

6 Implementation of interpretations

So far all we have done is to evaluate logic on its merits as a programming language. This would be futile without a way to use a computer to obtain results that are verified by a logic theory. In this section we describe a method for this purpose.

Our starting point is the way a logical theory is used to define an abstract mathematical concept. Take for example the concept of group. Whether a structure is a group is determined by the group axioms, which are formalized as a theory of logic. The criterion is whether the structure, regarded as an interpretation, makes the theory a true sentence. Many different structures are groups according to this criterion. Many computer applications can be analyzed in terms of mathematical structures: numbers of various kinds, strings, n-ary relations, vectors, matrices, graphs, partially-ordered sets, ... Axiomatizations of these structures have been expressed as logical theories or are candidates for such treatment. The values to be computed appear as values of functions or as arguments to relations. These functions and relations occur in a logical theory or, more likely, as extensions defined in the way described in this paper.

One way of combining logic and a computer application is to arrange the operations of the computer in such a way that they can be interpreted as inferences from an axiomatic theory. This is what is done in logic programming, where resolution is the inference rule. In this paper we propose a different way. We do not use an inference system. We use the fact that a given theory can be satisfied by two different interpretations, say, A and B. A is the familiar mathematical structure. B a program in a conventional programming language that is compiled and executed in the conventional way. If B is also sufficiently similar to an interpretation of the theory and if this interpretation satisfies the theory, then we can say that A is a specification of B and that B is verified with respect to A. If B is written in a language like `C++` , then there is the possibility that it makes optimal use of the computer's hardware.

Consider the mathematical concept of a Euclidean domain. The structure is axiomatized as having as functions commutative addition, subtraction and commutative multiplication with multiplication distributing over addition. It contains 0 as the neutral element for addition and 1 as the neutral element for multiplication. The integers are an example of a structure that satisfies the Euclidean domain axioms. As another example of such a structure consider the set of bit patterns stored in computer memory and operations on them implemented by hardware instructions or software programs.

> *In so far as the resulting structure satisfies the Euclidean domain axioms, these instructions and programs are verified as being a correct implementation of a Euclidean domain.*

This fact is the basis of the method of using logic for programming that we propose in this paper. It remains to find a convenient way of tying together subroutines and a type that can be regarded as an interpretation that can be examined whether it satisfies the intended theory. The class mechanism of `C++` offers a

reasonably convenient way of assembling types and subroutines to be regarded as an interpretation for a theory of logic.

We present the class listed below as an interpretation for the axioms for a Euclidean domain extended with the definition of the three-argument relation `gcd`. Ideally one should be able to translate the Horn formula (4) to the code

```
bool gcd(x, y, z){
  if (x<y && gcd(x, y-x, z)) return true;
  if (y<x && gcd(x-y, y, z)) return true;
  z = x; return true;
}
```

In actual fact we needed to clutter up this definition as shown in the listing below. The listing implements one specific Euclidean domain: that of the natural numbers modulo 65521. It has the property that the class is an exact interpretation of the axioms: no approximations are made, nor is the correctness vitiated by the possibility of overflow. The bit patterns in the computer (little-endian two's complement integers) are one of the many universes of discourse for interpretations satisfying the axioms for Euclidean domains.

```
class ED{ // ED: Euclidean Domain
int val; const static int mod = 65521;
//class invariant: 0 <= val < mod
public:
  ED(): val(0) {}
  ED(int val): val(val) {
   if (val < 0) this -> val = mod - (-val)%mod;
   else this -> val %= mod; }
  static ED zero() { return ED(0); }
  static ED unit() { return ED(1); }
  friend ED operator+(const ED& x, const ED& y)
    { return ED(x.val + y.val); }
  friend ED operator-(const ED& x, const ED& y)
    { return ED(x.val - y.val); }
  friend ED operator*(const ED& x, const ED& y)
    { return ED(x.val * y.val); }
  friend bool operator<(const ED& x, const ED& y)
    { return x.val < y.val; }
  static bool gcd(const ED& x, const ED& y, ED& z){
    if (x<y && gcd(x, y - x, z)) return true;
    if (y<x && gcd(x - y, y, z)) return true;
    z = x; return true;
  }
};
int main() { ED c; ED::gcd(ED(48), ED(36), c); }
```

7 Conclusions

In this paper we addressed the question of what first-order predicate logic, in its pristine form before there were computers, can do for programming. On the positive side we see terms that range over a universe of discourse (corresponding to the values that program variables assume), function symbols that have functions as interpretation (corresponding to side-effect free function subroutines), and relation symbols that have relations as interpretation (corresponding to side-effect free procedures). On the negative side are (1) no mechanism for defining new functions and relations on the basis of existing ones and (2) it is not clear how to get a computer to evaluate a term or determine the truth value of a formula. Both of these shortcomings have been met in this paper.

As for problem (1), our analysis is that it is caused by the absence of compositional semantics for logic. We corrected this deficiency by introducing a mechanism for extending an existing theory with new function and relation symbols and its interpretation with the corresponding functions and relations.

Our function definitions are not allowed to be recursive. This restriction is forced by the fact that function symbols are interpreted by total functions. Lifting this restriction has been the subject of much research, a sample of which is found in [2]. For us this is not a high priority, as the restriction is no obstacle to making definitions of new *relation* symbols recursive.

This leaves us with a language in which the programmer can define new relations in mutual recursion on the basis of existing relations and a repertoire of total functions (some coming from the axiomatic theory, some programmer-defined according to the mechanism described in this paper) that is fixed in the context of the relational definitions. Logic programming is more restricted: the universe of discourse is the Herbrand universe, therefore entirely determined by the function symbols of the theory, and the same holds for their interpretations. In our approach the universe of discourse can be any data types that are representable in a computer memory; the functions can be any total functions definable as first-order terms.

The meaning of our recursive definitions of relation symbols is determined by a set of equations. We restrict ourselves to a simple special case in which these equations are known to have a unique least solution. We call the formulas of this special case "Horn formulas" as they correspond to a subset of the Horn clauses if translated to clausal form.

Let us now consider problem (2), how to connect the logical theory and its extensions to a computer in a way that optimally uses its hardware, including its arithmetic. In logic programming, relations are defined by Horn clauses. The computer is used to carry out resolution inference to obtain a logical consequence of the definitions. In our method we use no inference. Instead we use the fact that a theory of logic is agnostic about its interpretations. According to our method we write a program that can be regarded as an interpretation for this same theory. Our counterpart of the elements of the Euclidean domain are two's complement little-endian bit patterns that behave according to some Intel manual. The fact that the program is also an interpretation of a theory that is

satisfied by Euclidean domains verifies the program as a correct implementation of this abstract algebraic structure.

To our knowledge no programming language exists that allows one to specify an interpretation for a theory in first-order predicate logic. Our method is interesting because one can approach this ideal by writing in `C++` a class with functions and relations that correspond closely enough to those of a Euclidean domain. We extend our theory by a three-place relation for *gcd*.

The correspondence between the `C++` definition of the algorithm and the logical formula extending the theory is far from perfect, but significant. That anything like this is possible at all is a marvel, considering that `C++` started out as "C With Classes" [4] and has remained constrained by compatibility with C during its formative years. We rejected Java as a language for interpretations because of its reliance on heap storage allocation. Our hunch is that there are plenty of interesting algorithms that only need stack storage allocation, which is what we see exclusively in the listing in Section 6.

The results in this paper suggest research both in logic and in programming languages. First-order predicate logic, as we have inherited it from the early twentieth century, is only suited for the formalization of small axiom systems. It works fine for a group. But it fails already for something as mundane as a Euclidean domain ("a ring with cancelation law and a valuation, where a ring is a commutative additive group as well as a multiplicative monoid" [1]). The mechanisms developed for programming languages may be helpful here. On the programming-language side `C++` is an encouraging example. In spite of its having evolved under the constraint of compatibility with C, it seems the best existing vehicle for implementing interpretations of logic theories that run efficiently. A language for implementing interpretations of logic theories that is similar to `C++` , but released from the constraint of compatibility with a primitive language, may be an advance in programming languages not seen since the main paradigms, exemplified by Fortran, Lisp, Algol, Simula, Prolog, Smalltalk, and ML, were all in place.

Acknowledgments

Thanks to Paul McJones and the referees for help in many ways. This research benefited from facilities provided by the University of Victoria and by the Natural Science and Engineering Research Council of Canada.

References

1. Garrett Birkhoff and Thomas Bartee. *Applied Modern Algebra.* McGraw-Hill, 1970.
2. Robert Cartwright. Recursive programs as definitions in first order logic. *SIAM Journal of Computing*, vol. 13 (1984), pages 374 – 408.
3. Alexander Stepanov and Paul McJones. *Elements of Programming.* Addison-Wesley, 2009.
4. Bjarne Stroustrup. *The Design and Evolution of* `C++` . Addison-Wesley, 1994.

Program Analysis and Manipulation to Reproduce Learners' Erroneous Reasoning

Claus Zinn

Department of Computer Science
University of Konstanz
Box D188, 78457 Konstanz, Germany,
claus.zinn@uni-konstanz.de

Abstract. Pedagogical research shows that learner errors are seldom random, but result from correctly executing an erroneous procedure. Effective teaching, thus, depends on deep cognitive analyses to diagnose, and subsequently repair those incorrect parts. We report a method for the reconstruction of such erroneous procedures based on the analysis and manipulation of logic programs. The method relies on an iterative application of two algorithms: an innovative use of algorithmic debugging to identify learner errors by the analysis of (initially) correct *(sic)* Prolog-based procedures, and a subsequent program manipulation phase where errors are introduced into (initially) correct procedures. The iteration terminates with the derivation of an erroneous procedure that was followed by the learner. The procedure, and its step-wise reconstruction, can then be used to inform remedial feedback.

1 Introduction

The diagnosis of learner input is central for the provision of effective scaffolding and remedial feedback in intelligent tutoring systems. A main insight is that errors are rarely random, but result from correctly executing erroneous procedures. Effective teaching thus depends on deep cognitive analyses to diagnose and then repair those incorrect parts. State-of-the-art intelligent tutoring systems, however, fail to give a full account of learners' erroneous skills. In *model tracing* tutors (*e.g.*, the Lisp tutor [2]; the Algebra Tutor [6]), appropriately designed user interfaces and tutor questions invite learners to provide their answers in a piecemeal fashion. It is no longer necessary to reproduce a student's line of reasoning from question to (final) answer; only the student's next step towards a solution is analyzed, and immediate feedback is given. Model tracing tutors thus keep learners close to ideal problem solving paths, hence preventing learners to fully exhibit erroneous behaviour. *Constraint-based tutors* (*e.g.*, the SQL tutor [8]) perform student modelling based on constraints [9]. Here, diagnostic information is not derived from an analysis of learner actions but of problem states the student arrived at. With no representation of actions, the constraint-based approach makes it hard to identify and distinguish between the various (potentially erroneous) procedures learners follow to tackle a given problem.

None of the two approaches attempt to explain buggy knowledge or skills. There is no explicit and machine-readable representation to mark deviations of an expert rule from the buggy skill; and also, there is no mechanism for automatically deriving buggy skills from correct ones. In this paper, we report a method capable of reconstructing erroneous procedures from expert ones. The method is based on an iterative analysis and manipulation of logic programs. It relies on an innovative use of algorithmic debugging to identify learner error by the analysis of (initially) correct *(sic)* Prolog-based procedures (modelling expert skills), and a subsequent program manipulation to introduce errors into the correct procedure to finally produce the erroneous procedure followed by the learner. The method extends our previous work [12] by having algorithmic debugging now qualify the irreducible disagreement with its cause, *i.e.*, by specifying those elements in the learner's solution that are missing, incorrect, or superfluous. Moreover, we have defined and implemented a perturbation algorithm that can use the new information to transform Prolog programs into ones that can reproduce the observed error causes.

The remainder of the paper is structured as follows. Sect. 2 introduces the domain of instruction (multi-column subtraction), typical errors and how they manifest themselves in this domain, and our previous work on adapting Shapiro's algorithmic debugging to support diagnosis in intelligent tutoring systems. Sect. 3 first describes our new extension to Shapiro's technique; it then explains how we interleave the extended form of algorithmic debugging with automatic code perturbation, and how this method can be used iteratively to track multiple errors. In Sect. 4, we give examples to illustrate the method, showing the strengths and current limits of our approach. In Sect. 5, we discuss related work, and Sect. 6 lists future work and concludes.

2 Background

Our approach to cognitive diagnosis of learner input is applicable for any kind of domain that can be encoded as logic program. For illustration purposes, we focus on the example domain of multi-column subtraction.

2.1 Multi-Column Subtraction

Fig. 1 gives an implementation of multi-column subtraction in Prolog. Sums are processed column by column, from right to left. The predicate `subtract/2` determines the length of the sum, and passes the arguments to `mc_subtract/3`, which implements the recursion.[1] The predicate `process_column/3` gets a partial sum, processes its right-most column and takes care of borrowing (`add_ten_to_minuend/3`) and pay-back (`increment/3`) actions. Sums are encoded as Prolog lists of columns, where a column is represented as 5-element term `(B, M, S, P, R)` representing

[1] Note that the variable `CurrentColumn` is not required to specify our algorithm for multi-column subtraction; it is rather necessary for the mechanisation of the Oracle (see below), and thus passed on as argument to most of the predicates.

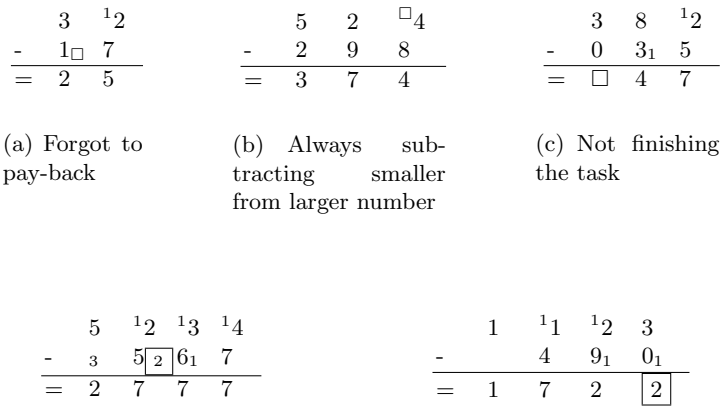
```
subtract(PartialSum, Sum) :-
        length(PartialSum, LSum),
        mc_subtract(LSum, PartialSum, Sum).

mc_subtract(_, Sum, Sum ) :-
        finished(Sum).
mc_subtract(CurrentColumn, Sum, NewSum) :-
        process_column(CurrentColumn, Sum, Sum1),
        shift_left(CurrentColumn, Sum1, Sum2, ProcessedColumn),
        CurrentColumn1 is CurrentColumn - 1,
        mc_subtract(CurrentColumn1, Sum2, SumFinal),
        append(SumFinal, [ProcessedColumn], NewSum).

process_column(CurrentColumn, Sum, NewSum) :-
        last(Sum, LastColumn),       allbutlast(Sum,RestSum),
        subtrahend(LastColumn, Sub), minuend(LastColumn, Min),
        Sub > Min,
        add_ten_to_minuend(CurrentColumn, LastColumn, LastColumn1),
        last(RestSum, LastColumnRestSum), allbutlast(RestSum, RestSum1),
        CurrentColumn1 is CurrentColumn - 1,
        increment(CurrentColumn1, LastColumnRestSum, LastColumnRestSum1),
        take_difference(CurrentColumn, LastColumn1, LastColumn2),
        append(RestSum1,[LastColumnRestSum1,LastColumn2],NewSum).

process_column(CurrentColumn, Sum, NewSum) :-
        last(Sum, LastColumn),       allbutlast(Sum,RestSum),
        subtrahend(LastColumn, Sub), minuend(LastColumn, Min),
        Sub =< Min,
        take_difference(CurrentColumn, LastColumn, LastColumn1),
        append(RestSum,[LastColumn1], NewSum).

finished( [] ).

irreducible.

shift_left( _CurrentColumn, SumList, RestSumList, Item ) :-
        allbutlast(SumList, RestSumList),
        last(SumList, Item).

add_ten_to_minuend(_CC, (_B, M, S, P, R), (10, M, S, P, R)) :- irreducible.
increment(         _CC, (B, M, S, _P, R),  (B, M, S, 1, R)) :- irreducible.
take_difference(   _CC, (B, M, S, P, _R),  (B, M, S, P, R)) :- irreducible,
        plus_safe(M, B, MB),
        R1 is MB - S,
        minus_safe( R1, P, R).

minuend( (B, M, _S, _P, _R),  MB)    :- plus_safe(M, B, MB).
subtrahend( (_B, _M, S, P, _R),  SP) :- plus_safe(S, P, SP).

plus_safe(  A1, A2, A) :- var(A2) -> A = A1 ; A is A1+A2.
minus_safe( A1, A2, A) :- var(A2) -> A = A1 ; A is A1-A2.
```


Fig. 1. Multi-column subtraction in Prolog

borrow, minuend, subtrahend, payback and result cell.[2] The program code implements the *equal additions (aka Austrian) method*. If the sum of the subtrahend `S` and the payback `P` is larger than the minuend `M`, then `M` is increased by 10 (marked in the borrowing cell `B`) before the difference between `M+B` and `S+P` is taken. To compensate, the `P` in the column left to the current one is then increased by one (pay-back).

2.2 Error Analysis in Multi-column Subtraction

Clearly, not all errors are equal. Some may be caused by a simple oversight (usually, pupils are able to correct such errors as soon as they see them), and others are *systematic errors* (those keep re-occurring again and again). It is the systematic errors that we aim at diagnosing as they indicate a pupil's wrong understanding about some subject matter (see Fig. 2 for a small sample of examples; the use of squares is explained below).



(a) Forgot to pay-back

(b) Always subtracting smaller from larger number

(c) Not finishing the task

(d) Accumulating all pay-backs to highest place value

(e) Performing algorithm from left to right

Fig. 2. A small selection of errors in subtraction

Errors can be classified according to the following scheme: (i) Errors of omission – forget to do something; (ii) Errors of commission – doing the task incorrectly; and (iii) Sequence errors – doing the task not in the right order. The application of this simple scheme would classify as errors of omission the examples given in Fig. 2(a, b, c); as errors of commission the examples in Fig. 2(b, d), and as sequence error the example given in Fig. 2(e).

[2] The predicates `append/3` (append two lists), `last/2` (get last list element), and `allbutlast/2` (get all list elements but the last) are defined as usual.

For each error, we aim to reconstruct their underlying erroneous procedures automatically using logic programming techniques, relying only on expert programs and learner answers.

2.3 Shapiro's Algorithmic Debugging

Shapiro's algorithmic debugging technique for logic programming prescribes a systematic manner to identify bugs in programs [10]. In the top-down variant, the program is traversed from the goal clause downwards. At each step during the traversal of the program's AND/OR tree, the programmer is taking the role of the *oracle*, and answers whether the currently processed goal holds or not. If the oracle and the buggy program agree on a goal, then algorithmic debugging passes to the next goal on the goal stack. If the oracle and the buggy program disagree on the result of a goal, then this goal is inspected further. Eventually an *irreducible disagreement* will be encountered, hence locating the program's clause where the buggy behavior is originating from. Shapiro's algorithmic debugging method extends, thus, a simple meta-interpreter for logic programs.

Shapiro devised algorithmic debugging to systematically identify bugs in incorrect programs. Our Prolog code for multi-column subtraction in Fig. 1, however, presents the expert model, that is, a presumably correct program. Given that cognitive modelling seeks to reconstruct students' erroneous procedure by an analysis of their problem-solving behavior, it is hard to see – at least at first sight – how algorithmic debugging might be applicable in this context. There is, however, a simple, almost magical trick, first reported in [12]. We can turn Shapiro's algorithm on its head: instead of having the Oracle specifying how the assumed incorrect program should behave, we take the expert program to take the role of the buggy program, and the role of the Oracle is filled by students' potentially erroneous answers. An irreducible disagreement between program behavior and student answer then pinpoints students' potential misconception(s).

An adapted version of algorithmic debugging for the tutoring context is given in [12]. Students can "debug" the expert program in a top-down fashion. Only clauses declared as being *on the discussion table* are subjected to Oracle questioning so that rather technical steps (such as `last/2` and `allbutlast/2`) do not need to be discussed with learners. Moreover, a Prolog clause whose body starts with the subgoal `irreducible/2` is subjected to Oracle questioning; but when program and Oracle disagree on such a clause, this disagreement is irreducible so that the clause's (remaining) body is not traversed. Most importantly, we have implemented the Oracle for subtraction tasks [12]. The answers to all questions posed by algorithmic debugging are automatically reconstructed from pupils' exercise sheets, given their solution to a subtraction problem.

3 Combining Algorithmic Debugging with Program Manipulation

While the method reported in [12] returns the first irreducible disagreement between expert and learner behaviour, additional information about the nature

of the disagreement is needed to better inform the transformation of the correct program into an incorrect program to better model learner behaviour.

3.1 Analysing the Causes of Disagreements

We further refine our adaptation of algorithmic debugging – and the Oracle is relies on – as follows: whenever expert and learner disagree on a goal, one or more of the following cases hold:

- the learner solution *misses* parts that are present in the expert solution, *e.g.*, a result cell in the multi-column subtraction table has not been filled out;
- the learner solution has *incorrect* parts with regard to the expert solution, *e.g.*, a result cell is given a value, albeit an incorrect one; and
- the learner solution has *superfluous* parts not present in the expert solution, *e.g.*, the learner performed a borrowing operation that was not necessary.

When subjecting each of the incorrect solutions given in Fig. 2 to our extended version of algorithmic debugging, we thus obtain irreducible disagreements *together* with their nature (*missing*, *incorrect*, or *superfluous*).

3.2 Main Algorithm

We take the enriched output of algorithmic debugging to inform the transformation of the correct program into a buggy program that better models student behavior. Note that multiple bugs can be tracked by iterative applications of algorithmic debugging and code perturbation.

```
1: function ReconstructErroneousProcedure(Program, Problem, Solution)
2:     (Disagr, Cause) ← AlgorithmicDebugging(Program, Problem, Solution)
3:     if Disagr = nil then
4:         return Program
5:     else
6:         NewProgram ← Perturbation(Program, Disagr, Cause)
7:         ReconstructErroneousProcedure(NewProgram, Problem, Solution)
8:     end if
9: end function

10: function Perturbation(Program, Clause, Cause)
11:     return chooseOneOf(Cause)
12:         DeleteCallToClause(Program, Clause)
13:         DeleteSubgoalOfClause(Program, Clause)
14:         ShadowClause(Program, Clause)
15:         SwapClauseArguments(Program, Clause)
16:         ChangeSequence(Program, Clause)
17: end function
```

Fig. 3. Pseudo-code: compute variant of *Program* to reproduce a learner's *Solution*.

Fig. 3 gives a high-level view of the algorithm for the reconstruction of learners' erroneous procedure. The irreducible disagreement resulting from the algorithmic debugging phase locates the place in the program where the code transformation must take place; its cause influences the preference over the type of program perturbation.

The function `Perturbation/3` implements various kinds of transformations: the deletion of a call to the clause in question, or the deletion of one of its subgoals (both reproducing errors of omission), or the shadowing of the clause in question by a more specialised instance, or the swapping of the clause' arguments (both reproducing errors of commission), or the consistent change of recursion operators (reproducing sequence errors). Future transformations may involve constructive elements such as the creation and insertion of new subgoals to extend the body of the clause in question.

We have implemented a number of generic program transformations to account for typical error types. For this, the clauses of the expert program receive *mode* annotations, marking their arguments as input and output arguments. Our algorithm for clause call deletion, for instance, traverses a given program until it identifies a clause whose body contains the clause in question; once identified, it removes the clause from the body and replaces all occurrences of its output argument by its input argument in the adjacent subgoals as well as in the clause's head, if present. Then, `DeleteCallToClause/2` returns the modified program.[3]

4 Example

We give a detailed discussion of our approach with regard to all of our five examples of typical errors in whole number computation.

4.1 Single Error Tracking (Error Omission)

In the single error example shown in Fig. 2(a), the learner forgets to honor the pay-back operation, following the borrowing that happened in the first (rightmost) column. The execution of the adapted version of algorithmic debugging produces the following dialogue (with all questions automatically answered by the mechanised Oracle):

```
algorithmic_debugging(subtract([(B1, 3, 1, P1, S1), (B2, 2, 7, P2, S2)],
                               [(B1, 3, 1, P1, 2),  (10, 2, 7, P2,  5)],
                      IrreducibleDisagreement).

do you agree that the following goal holds:
   subtract( [(B1, 3, 1, P1, R1), (B2, 2, 7, P2, R2)],
             [(B1, 3, 1, 1,  1),  (10, 2, 7, P2,  5)])
|: no.
```

[3] Prolog code implementing our approach is available from the author's website, see `http://www.inf.uni-konstanz.de/~zinn/lmlp_links.html`.

```
do you agree that the following goal holds:
   mc_subtract(2, [(B1, 3, 1, P1, R1), (B2, 2, 7, P2, R2)],
                  [(B1, 3, 1,  1, 1),  (10, 2, 7, P2, 5)])
|: no.

   process_column(2, [(B1, 3,1, P1, R1), (B2, 2, 7, P2, R2)],
                     [(B1, 3,1, 1,  R1), (10, 2, 7, P2, 5)])
|: no.

   add_ten_to_minuend(2, (B2, 2, 7, P2, R2), (10, 2, 7, P2, R2))
|: yes.

   increment(1, ( B1, 3, 1, P1, R1), ( B1, 3, 1, 1, R1))
|: no.
=> IrreducibleDisagreement=( increment(1,(B1, 3,1, P1, R1),
                                         (B1, 3,1, 1,  R1)),
                                missing )
```

With the indication of error (the location is marked by □ in Fig. 2a), program transformation now attempts various manipulations to modify the expert program. Given the cause "missing", the perturbation heuristics chooses to omit a call to the indicated program clause, which succeeds: in the body of the first clause of `process_column/3`, we eliminate its subgoal `increment(CurrentColumn, LastColumnRestSum, LastColumnRestSum1)` and subsequently replace its output variable `LastColumnRestSum1` with its input variable `LastColumnRestSum`.[4]. With this program manipulation, we achieve the intended effect; the resulting buggy program reproduces the learner's answer; both program and learner agree on the top `subtract/2` goal.

4.2 Tracking Multiple Errors (Error Omission and Commission)

An iterative execution of algorithmic debugging and program manipulation to the problem in Fig. 2(b) shows how multiple errors are attacked one by one.

First Run. Running algorithmic debugging on the expert program and the learner's solution concludes the dialogue (now omitted) with the irreducible disagreement

```
add_ten_to_minuend( 3, (B, 4, 8, P, R), (10, 4, 8, P, R))
```

and the cause "missing". The code perturbation algorithm deletes the subgoal `add_ten_to_minuend/3` from the first program clause of `process_column/3`; the occurrences of its output variable `LastColumn1` are replaced by its input variable `LastColumn` in the subsequent call to `take_difference/2`.

[4] The deletion of `increment/3` renders the two adjacent subgoals `last/2` and `allbutlast/2` obsolete so that they can be removed as well. The same holds for `is/2` adding 1 to `CurrentColumn`.

Second Run. A new run of algorithmic debugging on the modified program yields the disagreement

```
increment(2, (B, 2, 9, P, R), (B, 2, 9, 1, R))
```

with cause "missing". Again, after `DeleteCallToClause/2` deleted the subgoal `increment/3` from the first clause of `process_column/3`, we obtain a new program that is closer in modelling the learner's erroneous behaviour.

Third Run. Re-entering algorithmic debugging with the modified expert program now yields an irreducible agreement at

```
take_difference(3, (B, 4, 8, P, R), (B, 4, 8, P, -4))
```

with cause "incorrect" (ones column). Here, a mere deletion of a call to the clause in question is a bad heuristics as the result cell must obtain a value; the mode annotation for this clause is `(+,+,-)`. We must thus perturbate the clause's body, or shadow the clause with an additional, more specialised clause. A simple manipulation to any of the clause's subgoals fails to achieve the intended effect. To accommodate the result provided by the learner, we shadow the existing clause with:

```
take_difference( _CC, (B, 4, 8, P, _R),  (B, 4, 8, P, 4)) :- irreducible.
```

Note that this new clause can be immediately derived from the Oracle's analysis: the expert program and the learner disagreed on the value of the result cell (-4 vs. 4). While the new clause covers the learner's input, it is rather specific.

Fourth Run. We now obtain an irreducible disagreement in the tens column:

```
take_difference(2, (B, 2, 9, P, R), (B, 2, 9, P, -7))
```

with cause "incorrect". Similar to the previous run, we add another clause for `take_difference/3` to capture the learner's input:

```
take_difference( _CC, (B, 2, 9, P, _R),  (B, 2, 9, P, 7)) :- irreducible.
```

With these changes to the expert program, we now obtain a buggy program that entirely reproduces the learner's solution in Fig. 2(b).

Generalisation. The resulting program could be generalised over the last two additions

```
take_difference( _CC, (B, 4, 8, P, _R),  (B, 4, 8, P, 4)) :- irreducible.
take_difference( _CC, (B, 2, 9, P, _R),  (B, 2, 9, P, 7)) :- irreducible.
```

into a more general

```
take_difference( _CC, (B, M, S, P, _R),  (B, M, S, P, R)) :- irreducible,
        S > M,
        R is S - M.
```

See our section on future work.

4.3 Partially Executing The Correct Algorithm

In Fig. 2(c), we have the learner following the correct algorithm, but not executing it in its entirety. Algorithmic debugging will yield the irreducible disagreement on `take_difference/3` (left-most column). While the learner's result cell is empty, its mode annotation indicates that the learner's bug is an error of omission, hence the mechanised Oracle returns the cause "missing". The deletion of a call to the clause in question is not yielding the intended effect as `take_difference/3` has been successfully applied in the first two columns. The deletion of the respective call would thus yield a program that is unable to reproduce the learner's behaviour as observed earlier; rather then deleting a call to the clause in question, it must be complemented by a more specific clause that "overwrites" or "shadows" it. In the given case, as the subtrahend `S` is zero, we have `take_difference/3` do nothing to compute a value for the result cell:

```
take_difference( CC, (B, M, 0, P, R), (B, M, 0, P, R) ).
```

The situation would have been a little different if the learner had written 0 rather than nothing into the left-most result cell (exhibiting the error "subtracting zero from a number yields zero"). In this case, algorithmic debugging would return the same irreducible disagreement, but with the cause "incorrect". Here, we need to complement the existing definition of `take_difference/3` with:

```
take_difference( CC, (B, M, 0, P, R), (B, M, 0, P, 0) ).
```

Given the open question "did the learner write nothing into the result cell to mean it having a zero value?" we can have algorithmic debugging play to its advantage. Once the irreducible disagreement has been identified, the system could take this question to the learner, enticing him to provide input for the column in question.

4.4 Error Analysis: Accumulating Paybacks

The solution given in Fig. 2(d) indicates that the learner is accumulating the result of all `increment/3` operations to the highest place value.

First Run. A first run of algorithmic debugging returns the irreducible disagreement `increment/3` in the hundreds column with the cause "incorrect" (see the $\boxed{2}$, which must be 1). Similar to the previous example, the `increment/3` operation has been successfully applied in the ones column, so that the deletion of its respective call is not progressing the program into a state closer to learner behaviour. If we complement the existing definition of the predicate `increment/3` with the clause

```
increment( 2, (B, 2, 5, P, R), (B, 2, 5, 2, R)) :- irreducible.
```

we can "fix" the expert program.

Second Run. With the perturbated program, the next irreducible disagreement appears in the tens column, where `take_difference/3` finds 7 rather than 6 in the result cell (cause is "incorrect"). While `take_difference` succeeded in the past (ones column), it did because the payback cell had no entry. The deletion of its last subgoal `minus_safe(R1, P, R)` makes the irreducible disagreement disappear for the tens column without affecting it to work properly in the ones column.

Third run. With this change, the next irreducible disagreement is `increment/3` in the thousands column, where the learner recorded 3 instead of 1 in the payback cell. Similar to the first run, we complement `increment/3` with

```
increment(1, (B, 5, 0, P, R), (B, 5, 0, 3, R))  :- irreducible.
```

Fourth run. With this change, the next irreducible disagreement appears in the thousands column (clause `take_difference/3` with cause "incorrect"), where the accumulated paybacks must be deducted from the minuend. In our last modification to its definition, we had it ignore the value of the payback cell (deletion of its last subgoal). This must be qualified now; when the first argument to `take_difference/3` reaches 1 (indicating that we arrived at the top-left column), then the accumulated value of the `P` cell is considered. With the following change, the new program reproduces the answer given in Fig. 2(d):

```
take_difference( CC, (B, M, S, P, _R),  (B, M, S, P, R)) :- irreducible,
        plus_safe(M, B, MB),
        R1 is MB - S,
        (
           CC == 1 -> minus_safe( R1, P, R) ; R is R1
        ).
```

Generalisation. A program abstraction phase should take the clauses

```
increment( 1,   (B, 5, 0, P, R),  (B, 5, 0, 3, R)) :- irreducible.
increment( 2,   (B, 2, 5, P, R),  (B, 2, 5, 2, R)) :- irreducible.
increment( _CC, (B, M, S, _P, R), (B, M, S, 1, R)) :- irreducible.
```

to generalise them into a single clause. If the payback cell of the previous column is a variable, we instantiate it to 1, otherwise we increment the existing value by one (see our section on future work):

```
increment( CC, (B, M, S, _P, R), (B, M, S, NP, R)) :-
        irreducible,
        payback_previous_column(CC, PPC),
        var(PPC) -> NP is 1 ; NP is PPC + 1.
```

4.5 Changing the Sequence

The learner's solution given in Fig. 2(e), exhibiting an error of sequence, is an interesting and complex case; it highlights the current limits of our approach. In

this example, the output of algorithmic debugging gives little information to hint at the changes required. The first irreducible agreement identified is found when calling `take_difference/3` in the ones column. Since the minuend is larger than the subtrahend, the algorithm executes the second clause of `process_column/3`, and is thus ignorant of the value of the payback cell in this column. A perturbation of `take_difference/3` to align expert and learner behaviour fails.

If we were to observe the learner's problem solving from start to end, rather than its end product, we could easily diagnose that the learner processes the columns from left to right rather than right to left.

To capture the learner's erroneous sequencing behaviour, we need to give the code in Fig. 1 a rather substantial rewrite to change the order in which columns are being processed: all occurrences of `last/2` must be replaced by `first/2` (*i.e.*, list head), and all occurrences of `allbutlast/2` must be replaced by `allbutfirst` (*i.e.*, list tail). Moreover, in each call of `append`, we need to swap its first two arguments (also, the elements of the list given as second argument to `append/3` in the first clause of `process_column/3` need to be swapped).

4.6 Summary

With the discussion of the examples, we now summarise the heuristics employed to choose among the many possible program perturbations. In all but the last example, algorithmic debugging correctly indicated the clause that required manipulation. If learners committed errors of omission, deleting the respective call to the clause in question proved successful in the examples 4.1 and 4.2 (first and second run). However, a call to the clause should not been deleted whenever it ran successfully at an earlier stage (*e.g.*, example 4.3 where `take_difference/3` only failed for the hundreds column). In this case, the clause in question should be shadowed with a specialised instance that we immediately derived from the output of algorithmic debugging. The same holds when irreducible disagreements have cause "incorrect", see the shadowing performed in examples 4.2 (third and fourth run), 4.3 (second case discussed), and 4.4 (first and third run).

Example 4.4 shows that the manipulation of a clause' body has to be taken with caution. In its second run, we deleted a subgoal from the original `take_difference/3`, only to render its deletion conditional at a later stage (fourth run).

The necessary perturbation illustrated in example 4.5 can only be mechanised when the system is "made aware" that `head/2`, `tail/2`, and `rev_append/3` are the reciprocal predicates for `last/2`, `allbutlast/2` and `append/3`, respectively.

5 Related Work

There is only little research in the intelligent tutoring systems community that builds upon logic programming and meta-level reasoning techniques. In [1], Beller & Hoppe use a fail-safe meta-interpreter to identify student error. A Prolog program, modelling the expert knowledge for doing subtraction, is executed by instantiating its output parameter with the student answer. While standard

Prolog interpretation would fail, a fail-safe meta-interpreter can recover from execution failure, and can also return an execution trace. Beller & Hoppe then formulate *error patterns* which they then match against the execution trace; each successful match is indicating a plausible student bug.

In Looi's "Prolog Intelligent Tutoring System" [7], Prolog programs written by students are debugged with the help of the automatic derivation of mode specifications, dataflow and type analysis, and heuristic code matching between expert and student code. Looi also makes use of algorithmic debugging techniques borrowed from Shapiro [10] to test student code with regard to termination, correctness and completeness. The Oracle is mechanised by running expert code that most likely corresponds to given learner code, and simple code perturbations are carried out to correct erroneous program parts.

Most closely related to our research is the work of Kawai et al. [5]. Expert knowledge is represented as a set of Prolog clauses, and Shapiro's Model Inference System (MIS) [10], following an inductive logic programming (ILP) approach, is used to synthesize learners' (potentially erroneous) procedure from expert knowledge and student answers. Once the procedure to fully capture learner behaviour is constructed, Shapiro's Program Diagnosis System (PDS), based upon *standard* algorithmic debugging, is used to identify students' misconceptions, that is, the bugs in the MIS-constructed Prolog program.

While Kawai et al. use similar logic programming techniques, there are substantial differences to our approach. By turning Shapiro's algorithm on its head, we are able to identify the first learner error with no effort – for this, Kawai et al. require an erroneous procedure, which they need to construct first using ILP. To analyse a learner's second error, we overcome the deviation between expert behaviour and learner behaviour by introducing the learner's error into the expert program. This "correction" is performed by small perturbations within existing code, or by the insertion of elements of little code size (usally derived by building variants of clause heads with some of its arguments instantiated). We believe these changes to the expert program to be less complex than the synthesis of entire erroneous procedures by inductive methods alone.

6 Future Work and Conclusion

In our approach, an irreducible disagreement corresponds to an erroneous aspect of a learner answer not covered by the expert program (or a perturbated version thereof). The transformations performed by our system are (i) deleting calls to the clause in question, (ii) deleting one of the goals from the clause's body, or (iii) shadowing the clause in question with more specialised instances. At the time of writing, we are unable to fully mechanise the *generalisations* discussed in examples 4.2 and 4.4. Future work will reconsider Shapiro's MIS and follow-up work to construct the aforementioned generalisations, but also elements that are not mentioned in the expert program (or other kinds of background knowledge). We also aim at investigating the generation of system-learner interactions to obtain a small but sufficient set of learner input to serve as additional *positive or*

negative examples to the generalisation or induction process. These interactions must not distract students from achieving their learning goals; ideally, they can be integrated into a tutorial dialogue that is beneficial to student learning.

Expert models with representational requirements or algorithmic structure different to the subtraction algorithm given in Fig. 1 might have been beneficial to our analysis. At the time of writing, we have implemented four different subtraction methods: the Austrian method, its trade-first variant, the decomposition method, and an algorithm that performs subtraction from left to right. We have also created variants to these four methods (with differences in sequencing subgoals, or the structure of clause arguments). When a learner follows a subtraction method other than the Austrian method, we are likely to give a wrong diagnosis of learner action whenever the Austrian method is the only point of reference to expert behaviour. To improve the quality of diagnosis, it will thus be necessary to compare learner performance to a multitude of difference models. In a related strand of research, we have developed a method to identify the algorithm the learner was most likely following. For this, we have further adapted algorithmic debugging as follows: we count the number of *agreements* before the first irreducible disagreement is identified, and the number of *agreements* after the first and any other irreducible disagreements, thus covering the case where learners committed multiple errors (for details, see [11]). When counting agreements, we also take into account the "code size" that is being agreed upon.

Once an erroneous procedure has been constructed by an iterative application of algorithmic debugging and automatic code perturbation, it will be necessary to use the procedure as well as its construction history to inform the generation of remedial feedback. Reconsider our example Fig. 2(b) for the tracking of multiple errors. Its construction history shows that the following transformations to the correct subtraction procedure were necessary to reproduce the learners' erroneous procedure: (i) deletion of the goal `add_ten_to_minuend/3`, (ii) deletion of the goal `increment/3`, and (iii; iv) shadowing `take_difference/3` with more specialised instances. Some amount of reasoning about the four elements of the construction history is necessary to derive the compound diagnosis "Always subtracting smaller from larger number". Getting the diagnoser to acknowledge this learner behaviour is not trivial as it requires additional knowledge such as "lack of borrowing is usually followed by lack of payback", or the necessary reasoning involved in generalising the two `take_difference/3` instances. With this additional reasoning, a remediation strategy could start by asserting the observed compound erroneous learner behaviour before pointing out the need for the borrowing and payback operations.

In our approach, the analysis of learner input is linked to dialogue induced by algorithmic debugging. For multi-column subtraction, we were able to mechanise the Oracle, thus relieving the learner from answering any questions at all. From a pedagogical point of view, however, well-posed questions can have a positive effect on student learning. In the future, we would like to make a more informed use of the Oracle. Following Dershowitz and Lee [3], we would like to formulate specifications that can be executed to answer Oracle questions. We

will enrich such specifications with domain-specific tutorial expertise to decide whether questions should be forwarded to the learner. Following Fritzson et al. [4], we would like to complement test cases (generated from specifications, or from a test case database) with program splicing. Once algorithmic debugging has located an irreducible disgreement between expert performance and learner performance, the respective clause could serve as a *slicing criterion* for the computation of a subprogram – the *program slice* – whose execution may have an effect on the values of the clause's arguments. The program slice, only containing the relevant skill set with regard to the error in question, could then better focus the remediation activity of the intelligent tutoring system.

There is good evidence that a sound methodology of cognitive diagnosis in intelligent tutoring can be realized in the framework of declarative programming languages such as Prolog. In this paper, we reported on our work to combine an innovative variant of algorithmic debugging with program transformation to advance cognitive diagnosis in this direction.

Acknowledgments. Thanks to the reviewers whose comments helped improve the paper. Our work is funded by the German Research Foundation (ZI 1322/2/1).

References

1. S. Beller and U. Hoppe. Deductive error reconstruction and classification in a logic programming framework. In P. Brna, S. Ohlsson, and H. Pain, editors, *Proc. of the World Conference on Artificial Intelligence in Education*, pages 433–440, 1993.
2. A. T. Corbett, J. R. Anderson, and E. J. Patterson. Problem compilation and tutoring flexibility in the lisp tutor. In *Intell. Tutoring Systems*, Montreal, 1988.
3. N Dershowitz and Y.-J. Lee. Logical debugging. *J. Symbolic Computation*, 15(5-6):745–773, May 1993.
4. P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM Lett. Program. Lang. Syst.*, 1(4):303–322, 1992.
5. K. Kawai, R. Mizoguchi, O. Kakusho, and J. Toyoda. A framework for ICAI systems based on inductive inference and logic programming. *New Generation Computing*, 5:115–129, 1987.
6. K. R. Koedinger, J. R. Anderson, W. H. Hadley, and M. A. Mark. Intelligent tutoring goes to school in the big city. *Journal of Artificial Intelligence in Education*, 8(1):30–43, 1997.
7. C.-K. Looi. Automatic debugging of Prolog programs in a Prolog Intelligent Tutoring System. *Instructional Science*, 20:215–263, 1991.
8. A. Mitrović. Experiences in implementing constraint-based modeling in SQL-tutor. In *Intell. Tutoring Systems*, volume 1452 of *LNCS*, pages 414–423. Springer, 1998.
9. S. Ohlsson. Constraint-based student modeling. *Journal of Artificial Intelligence in Education*, 3(4):429–447, 1992.
10. E. Y. Shapiro. *Algorithmic Program Debugging*. ACM Distinguished Dissertations. MIT Press, 1983. Thesis (Ph.D.) – Yale University, 1982.
11. C. Zinn. Identifying the closest match between program and user behaviour. Unpublished manuscript, see `http://www.inf.uni-konstanz.de/~zinn`.
12. C. Zinn. Algorithmic debugging to support cognitive diagnosis in tutoring systems. In J. Bach and S. Edelkamp, editors, *KI 2011: Advances in Artificial Intelligence*, volume 7006 of *LNAI*, pages 357–368, Berlin, 2011. Springer.

Author Index