18th Workshop on Logic-based methods in Programming Environments

WLPE’08

Udine, Italy

December 12, 2008

Editors:

Puri Arenas and Damiano Zanardini
18th Workshop on Logic-based methods in Programming Environments

WLPE’08

Udine, Italy

December 12, 2008

Editors:

Puri Arenas and Damiano Zanardini
Preface

This volume contains the papers presented at the 18th Workshop on Logic-based methods in Programming Environments (WLPE’08), which was held in Udine, Italy, on December 12th, 2008. This workshop continues the series of successful international workshops on logic programming environments held in Ohio, USA (1989), Eilat, Israel (1990), Paris, France (1991), Washington D.C., USA (1992), Vancouver, Canada (1993), Santa Margherita Ligure, Italy (1994), Portland, USA (1995), Leuven, Belgium (1997), Las Cruces, USA (1999), Paphos, Cyprus (2001), Copenhagen, Denmark (2002), Mumbai, India (2003), Saint Malo, France (2004), Sitges (Barcelona), Spain (2005), Seattle, USA (2006) and Porto, Portugal (2007).

WLPE aims at providing an informal meeting for researchers working on logic-based methods and tools which support program development and analysis. This year, we have continued and consolidated the shift in focus from environmental tools for logic programming to logic-based environmental tools for programming in general, so that this workshop can be possibly interesting for a wider scientific community.

All the papers submitted to WLPE’08 have gone through a careful process of peer reviewing, with at least three reviews for each paper and a subsequent in-depth discussion in the Program Committee. In addition to regular papers, the Program Committee of WLPE’08 also included an invited talk by Andy King (Portcullis Computer Security Limited).

We would like to thank all the people who contributed to WLPE’08: the contributing authors; the PC members and the additional reviewers for their great effort in the review process; the invited speaker for his willingness to attend WLPE’08 and to prepare an extended abstract of his talk; the Organizing Committee for their continuous help; and, last but not least, the sponsoring institutions for their financial and logistic support.

Udine, December 12, 2008

Puri Arenas and Damiano Zanardini
Workshop Organization

Programme Chairs

Puri Arenas
Damiano Zanardini

Programme Committee

Michael Codish  Roberta Gori
Arnaud Gotlieb  Patricia Hill
Jacob Howe     Sabina Rossi
Tom Schrijvers  Alexander Serebrenik
Wim Vanhoof    Germán Vidal

Local Organization

Alberto Casagrande
Elisabetta De Maria
Luca Di Gaspero
Carla Piazza

External Reviewers

Stefano Bistarelli
Rafael Caballero
Florence Charreteur
Marco Comini
François Degrave
Andy King
Edison Mera
Salvador Tamarit
Sponsoring Institutions

Ministerio de Educación y Ciencia
Grants TIN2005-09207-C03-03 ‘MERIT-FORMS’
TIN2008-06622-C03-01 ‘FAST-STAMP’

Comunidad de Madrid
Grant S-0505/TIC/0407 ‘PROMESAS-CAM’

European Union, 6th Framework program
Grant 015905, ‘MOBIUS’

All the supporting
– Universitas Studiorum Utinensis
– NM State University
– Fondazione CRUP
– Agemont
– Istituto Nazionale di Alta Matematica Francesco Severi

sponsoring
– NICTA
– Kodak
– Vodafone
– SICS

and patronating
– Comune di Udine
– Association for Logic Programming
– Gruppo Ricercatori e Utenti Logic Programming

institutions of ICLP’08.
# Table of Contents

Relations, Constraints and Abstractions: Using the Tools of Logic Programming in the Security Industry *(invited talk)* .......... 1  
*Andy King*

Prolog Visualization System Using Logichart Diagrams ............... 8  
*Yoshihiro Adachi*

A Semantics-Aware Editing Environment for Prolog in Eclipse ........ 18  
*Jens Bendisposto, Ian Endrijautzki, Michael Leuschel, David Schneider*

On the Generation of Test Data for Prolog by Partial Evaluation ...... 26  
*Miguel Gómez-Zamalloa, Elvira Albert, Germán Puebla*

Improving Size-Change Analysis in Offline Partial Evaluation .......... 44  
*Michael Leuschel, Salvador Tamarit, Germán Vidal*

A Lightweight Combination of Semantics for Non-deterministic Functions 52  
*Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, Jaime Sánchez-Hernández*

Rfuzzy framework .......................................................... 62  
*Victor Pablos Ceruelo, Susana Muñoz Hernández, Hannes Strass*

Constraint solving for high-level WCET analysis ....................... 77  
*Adrian Prantl, Jens Knoop, Markus Schordan, Markus Triska*

Better Termination for Prolog with Constraints .......................... 90  
*Markus Triska, Ulrich Neumerkel, Jan Wielemaker*
Relations, Constraints and Abstractions: Using the Tools of Logic Programming in the Security Industry

Andy King
Portcullis Computer Security Limited, Pinner, HA5 2EX, UK

Abstract. Logic programming is sometimes described as relational programming: a paradigm in which the programmer specifies and composes n-ary relations using systems of constraints. An advanced logic programming environment will provide tools that abstract these relations to transform, optimise, or even verify the correctness of a logic program. This talk will show that these concepts, namely relations, constraints and abstractions, turn out to also be important in the reverse engineer process that underpins the discovery of bugs within the security industry.

1 Introduction

Logic programming is a wonderful paradigm: it is wonderfully expressive and yet also comes equipped with some wonderfully elegant semantics. One legacy of the foundational work on semantics by pioneers such as Kowalski, Levi and van Emden, are suites of tools that we build and deploy within the field of research that we refer to as logic programming environments. Partial evaluators, program specialisation tools, and various program analyses are all formulated in terms of the base semantics proposed by these pioneers. These base semantics provide a way to judge the correctness of a program manipulation technique, and by applying abstraction methods, we can even synthesise program analyses from these base semantics in a systematic and principled way [5]. Abstraction is a powerful idea in program manipulation, but when coupled with the pantheon of semantics that exist in logic programming, the concept becomes doubly powerful: we just need to select a suitably expressive semantics and then abstract it in an appropriate way. These ideas and these tools are so much a part of our heritage that we give this rich corpus of work a second thought.

The richness of the tooling that is available in logic programming becomes more evident when it is compared against the tooling that is available in reverse engineering. Reverse engineering is the discipline of extracting information from a program when the source is unavailable. Reversing engineering (abbreviated to reversing in the security sector) is routinely applied when performing a security audit on a commercial product that relies on software developed by a third-party such as a library. Security engineers also reverse to reason about the latest malicious programs and devise antivirus software. Reversing is also
necessary when auditing programs for vulnerabilities that are introduced by the compilation process itself, or are most evident at the level of the executable.

The most popular tool that is used for reversing within the security community is the IDA Pro dissembler [11]. This dissembler divides an executable into (more or less) its basic blocks, presenting them visually to the engineer in a flow diagram. Needless to say, the major impediment to reversing is the enormous effort required to understand an executable even when it is presented as a flow diagram. As researchers in programming environments, we are conscious that tool support can underpin the development of a new program, and aid the understanding of an existing program. The problem of extracting information from a program — which is the very essence of reversing — is not new to us. We recognise it as the problem of discovering invariants in a program. The problem is more about how to migrate techniques from higher-level paradigms to the level of an executable. In this short paper, we shall show how the familiar ideas from logic programming — relations, constraints, and abstractions of constraints — can be reinterpreted and reapplied in the setting of reverse engineering.

2 Where are the relations?

The place to start has to be the base semantics. In assembler, the problem is not that the semantics is ambiguous (like some languages); the problem is more one of granularity. Instructions perform bit-wise operations on words rather than merely arithmetical and logical operations on variables. The foci of computation are words, bit sequences and control-flags. Moreover, these objects are referenced through pointers and pointer offsets rather than as local variables and as, say, elements of an array. These semantics can be modelled, at least partially, by using relations. The idea is to exploit the finite nature of machine words and model each word or register as a vector of bits. The before and after states of each instruction can then be represented as a relation between the bits of the input and output vectors. Such a relation can be described propositionally as a Boolean formula over the propositional variables in the input and output vectors. This approach of modelling is colloquially referred to as “bit-blasting” within the model-checking community, presumably because of the explosive nature of the technique. Bit-blasting was famously used within the CBMC tool [4] in which the loops of C programs are unwound to a fixed depth so as to search for violations against prescribed correctness properties. Although initially treated with some skepticism, bit-blasting has gained acceptance as SAT solvers have emerged that can check the satisfiability of very large formulae [12] and SMT solvers have been developed that include bit vector theories [3] that directly support word-level instructions.

Bounded model checking has been successfully applied to check invariants, and find circumstances in which invariants are violated, but it cannot extract a hitherto unknown invariant from a program. Nevertheless, the relational nature of bit-blasting does provide a base semantics that is compositional. To see this, consider a sequence of just two instructions that both add the constant one to the
same 32-bit register. The input and output relation for this increment operation could be expressed as a Boolean formula \( f \) over the bits in the input and output vectors \( \langle r_0, \ldots , r_{31} \rangle \) and \( \langle r'_0, \ldots , r'_{31} \rangle \) where \( r_i \) and \( r'_i \) are the variables that express the state of bit \( i \) in the register before and after the increment. To compose two increments, two formulae \( f_1 \) and \( f_2 \) are obtained from \( f \) by, respectively, systematically renaming the \( r'_i \) variables to \( r''_i \), and renaming the \( r_i \) variables to \( r'_i \). The conjunction \( f_1 \land f_2 \) then asserts a double increment on the vectors \( \langle r_0, \ldots , r_{31} \rangle \) and \( \langle r'_0, \ldots , r'_{31} \rangle \), albeit using a vector of temporary variables \( \langle r''_0, \ldots , r''_{31} \rangle \). The \( \langle r''_0, \ldots , r''_{31} \rangle \) variables can be removed from the formula \( f_1 \land f_2 \) without loss of information by applying existential quantifier elimination. This can give a denser representation of the composed semantics though it is not strictly necessary.) By iterating this composition technique, it is possible to derive the relational semantics for a sequence of instructions of arbitrary length.

3 Where are the constraints?

One important idea in the analysis of logic programs is to use systems of constraints to describe systems of constraints [6]: systems of arbitrary Herbrand equations might be described by equations that are limited to depth-\( k \); systems of finite domain constraints might be described by conjunctions of Horn formulae that express definiteness dependencies [1]. We can reinterpret this idea for Boolean formulae and use formulae in one class to describe those in a more expressive class. Alternatively formulae could be described by systems of linear constraints. Using linear constraints as descriptions for formulae is more natural than one would initially think for reversing. When formulae are derived by bit-blasting and composition, the relationship between input and output vectors often resemble systems of simple linear constraints. For instance, in the case of the double increment, the formula \( f_1 \land f_2 \) could be described by the constraint \( 2 + \sum_{i=0}^{31} 2^i r_i = \sum_{i=0}^{31} 2^i r'_i \mod 2^{32} \). The constraint is not actually linear but is a congruent constraint with a modulo of \( 2^{32} \) which reflects the bounded nature of arithmetic that is expressed by the formula \( f_1 \land f_2 \). Describing the function with the linear relationship \( 2 + \sum_{i=0}^{31} 2^i r_i = \sum_{i=0}^{31} 2^i r'_i \) would actually misrepresent \( f_1 \land f_2 \). This is because, if the register initially stored the value \( 2^{32} - 1 \), then after the double increment, the register will contain 1 and not \( 2^{32} + 1 \). Thus the relationship is only linear on a sub-range of the input data values. Congruence constraints are natural abstractions for reversing because they are already familiar to the reverse engineer. This is because a number of security vulnerabilities relate to moduli; such vulnerabilities typically arise because the programmer has overlooked the wrap-around nature of arithmetic. Moreover, a security engineer will pay close attention to the size of an operand when reconstructing an algorithm from an executable.

An astute reader (and certainly a reverse engineer) will recall that a word can either be interpreted as a signed or an unsigned value. The congruence \( 2 + \sum_{i=0}^{31} 2^i r_i = \sum_{i=0}^{31} 2^i r'_i \mod 2^{32} \) stems from an unsigned treatment, otherwise congruence would be \( 2 - 2^{31} r_{31} + \sum_{i=0}^{30} 2^i r_i = -2^{31} r'_{31} + \sum_{i=0}^{30} 2^i r'_i \mod 2^{32} \).
where \( r_{31} \) and \( r'_{31} \) are the signs. However, observe that by adding \( 2^{32} r_{31} + 2^{32} r'_{31} \) to both sides, the congruence reduces to \( 2 + \sum_{i=0}^{31} 2^i r_i = \sum_{i=0}^{31} 2^i r'_i \mod 2^{32} \). Thus the same congruence conveniently describes both the signed and unsigned interpretation of words.

Congruences reflect the bounded nature of computer arithmetic, but an equation such as \( 2 + \sum_{i=0}^{31} 2^i r_i = \sum_{i=0}^{31} 2^i r'_i \mod 2^{32} \) possesses solutions for the variables \( \langle r_0, \ldots, r_{31} \rangle \) and \( \langle r'_0, \ldots, r'_{31} \rangle \) that are not 0-1 (truth) values. For instance, the congruence is satisfied by the assignment \( \{ r_0 \mapsto 2, r_1 \mapsto 0, \ldots, r_{31} \mapsto 0, r'_0 \mapsto 0, r'_1 \mapsto 2, r'_2 \mapsto 0, \ldots, r'_{31} \mapsto 0 \} \). Such an assignment has no clear relationship with a Boolean function: a Boolean function is characterised by its set of assignments to 0-1 values. It is therefore necessary to be clear as to how a Boolean function can be described by a congruence. Formally, this is role of the concretisation map: the concretisation for system of congruences is the Boolean function whose satisfying assignments constitute the 0-1 solutions of the system (any solution that assigns a value other than 0 or 1 is simply ignored in this interpretation of a congruence).

### 4 Where are the abstractions?

Stating the concretisation map (or dually an abstraction map) is much like providing a specification of a problem. Realising an algorithm that satisfies the specification and thus solves the problem is another thing entirely. Superficially it would seem that Boolean formulae and congruences are not closely related, and therefore it is not obvious how to find a system of congruences that best describe a given Boolean function. However, this problem can be recently solved using an iterative algorithm [9]. The force of this result is that it gives a way to describe the relational semantics of an instruction, or even a sequence of instructions, with a system of congruences: bit-blasting is first used to derive a formula for the sequence and then this formula is described by congruences. Then the invariants on the basic blocks can be derived by fixpoint techniques [10] that have been proposed for imperative programs. To illustrate these ideas, we return to reasoning about a double increment. For expositional purposes, we will suppose that a word is merely 4 bits wide. Then bit-blasting could derive the following system of (implicitly conjoined) formulae:

\[
f = \begin{cases} 
  r'_0 & \iff \neg r_0 \\
  r'_1 & \iff r_1 \oplus r_0 \\
  r'_2 & \iff r_2 \oplus (r_0 \land r_1) \\
  r'_3 & \iff r_3 \oplus (r_0 \land r_1 \land r_2) \\
\end{cases} \quad \begin{cases} 
  r''_0 & \iff \neg r'_0 \\
  r''_1 & \iff r'_1 \oplus r'_0 \\
  r''_2 & \iff r'_2 \oplus (r'_0 \land r'_1) \\
  r''_3 & \iff r'_3 \oplus (r'_0 \land r'_1 \land r'_2) \\
\end{cases}
\]

Note that the formula contains the intermediate variables \( \langle r'_0, r'_1, r'_2, r'_3 \rangle \) which could be eliminated to derive a (possibly smaller) formula that still relates the input and output vectors \( \langle r_0, r_1, r_2, r_3 \rangle \) and \( \langle r''_0, r''_1, r''_2, r''_3 \rangle \).

A congruent description is derived for \( f \) by first searching for a satisfying assignment (model) of \( f \). This can be readily accomplished with a SAT solver. One
such assignment is $M_1$ that is given below as 0-1 vector, where the propositional variables are ordered as follows $\langle r_0, r_1, r_2, r_3, r'_0, r'_1, r'_2, r'_3, r''_0, r''_1, r''_2, r''_3 \rangle$.

$$
M_1 = \langle 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0 \rangle \\
M_2 = \langle 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1 \rangle \\
M_3 = \langle 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0 \rangle \\
\vdots \\
M_{10} = \langle 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0 \rangle
$$

The truth assignment $M_1$ can be reinterpreted as the system of congruences $S_1$. For instance, the single assignment $r_0 \mapsto 1$ gives rise to the single congruence $r_0 = 1 \mod 2^4$. Henceforth, for brevity, we omit the modulo, which in this circumstance is chosen to be $2^4 = 16$ since words are 4 bits wide.

$$
S_1 = \begin{cases}
  r_0 = 1, r_1 = 0, r_2 = 0, r_3 = 0, r'_0 = 0, r'_1 = 1, \\
  r'_2 = 0, r'_3 = 0, r''_0 = 1, r''_1 = 1, r''_2 = 0, r''_3 = 0
\end{cases}
$$

$$
S_2 = \begin{cases}
  r_0 = 1, r_1 = 0, r_2 = 0, r_3 = r'_3, r'_0 = 0, r'_1 = 1, \\
  r'_2 = 0, r'_3 = r''_3, r''_0 = 1, r''_1 = 1, r''_2 = 0
\end{cases}
$$

$$
S_3 = \begin{cases}
  r_0 = 1, r_1 = 0, r_2 = r'_2, r_3 = r'_3, r'_0 = 0, r'_1 = 1, \\
  r'_2 = r''_2, r'_3 = r''_3, r''_0 = 1, r''_1 = 1
\end{cases}
$$

$$
\vdots
$$

$$
S_{10} = \begin{cases}
  r_0 + r'_0 = 1, r_1 + r''_0 = 1, 4r_2 + 4r''_2 + 4 = 8r_3 + 4r'_0 + 4r'_1 + 8r''_0 + 8r''_1, \\
  r'_0 + r''_0 = 1, 2r'_1 + 4r'_2 + 2 = 8r''_3 + 2r''_0 + 2r''_1 + 4r''_2 + 8r''_3
\end{cases}
$$

The algorithm proceeds by searching for an assignment of $f$ that is not described by the system $S_1$. This gives the model $M_2$ which can be translated into another system of simple congruences $S'_2$. The system $S_2$ is then derived from $S_1$ and $S'_2$ by computing the merge of $S_1$ and $S'_2$. This is the unique system that contains all the solutions of $S_1$ and $S'_2$. This operation is not dissimilar to the affine hull that is used to merge systems of linear equations [7]. With $S_2$ in place, the algorithm continues by searching for a model $M_3$ of $f$ that does not satisfy $S_2$. Translating $M_3$ as a system of congruences gives $S'_3$ which is then merged with $S_2$ to give $S_3$ that is also given in the table. This iterative scheme continues until $S_{10}$ is derived. All the models of $f$ are contained in $S_{10}$ and thus the algorithm stops at this point.

The system $S_{10}$ contains relational information pertaining to the intermediate bits as well as the input and output bits. The intermediate bits can be eliminated by applying a triangular form [7] which makes explicit any hidden relationships between the input and output bits:

$$
r_0 = r''_0, \quad r_1 + r''_1 = 1, \quad 2r_1 + 4r_2 + 8r_3 + 2 = 2r''_1 + 4r''_2 + 8r''_3
$$

Interestingly, the relationships derived are richer than one would expect. We have inferred that the states of the low bits are not changed by the double increment; that the states of the bits in position one always change; and that upper bits differ by two.
5 Related work

It has recently been pointed out that even recovering the control-flow graph is more complicated than one would initially expect [8] and, in fact, that IDA Pro often fails to reconstruct the complete control-flow graph. The problem stems in part from indirect calls, that is, when the address of a function is stored to a memory location pointed to by a register. The technical problem it is necessary to solve is to reason how intermediate instructions can possibly alter the value stored in the register and thereby infer that the address remained unchanged when the indirect call is resolved [8].

One notable body of work that also aims to support the reversing is the thesis work of Balakrishnan [2]. Balakrishnan, under the direction of Reps, has developed a so-called value set analysis that attempts to uniformly track addresses and numeric values. They intelligently chose a simple form of modulo constraint to represent a non-continuous range of values. For example, in their notation $4[0, 12]$ denotes the set $\{0, 4, 8, 12\}$ that describes the sets $\{0, 8\}$ and $\{8, 12\}$ among others. The rationale for this approach is that it enables sets of addresses on some word alignment to be accurately represented. We consider this approach to be a major advance in the analysis of binaries, since it attempts to seamlessly support addresses and numeric values.

Acknowledgments

This work was funded by EPSRC projects EP/C015517, EP/E033105 and EP/F012896 and a Royal Society Industrial Fellowship that has enabled King to be seconded to Portcullis Computer Security Limited. We thank Harald Søndergaard who has contributed to much of this work.

References


Prolog Visualization System Using Logichart Diagrams

Yoshihiro Adachi
Toyo University, 2100 Kujirai Kawagoe Saitama 3508585, Japan, adachi@toyonet.toyo.ac.jp

Abstract. We have developed a Prolog visualization system that is intended to support Prolog programming education. The system uses Logichart diagrams to visualize Prolog programs. The Logichart diagram is designed to visualize the Prolog execution flow intelligibly and to enable users to easily correlate the Prolog clauses with its parts. The system has the following functions. (1) It visually traces Prolog execution (goal calling, success, and failure) on the Logichart diagram. (2) Dynamic change in a Prolog program by calling extra-logical predicates, such as ‘assertz’ and ‘retract’, is visualized in real time. (3) Variable substitution processes are displayed in a text widget in real time.

1 Introduction

Prolog is a representative programming language for introductory education in AI programming. It has several characteristic mechanisms, including powerful pattern matching (unification, in Prolog terminology), automatic backtracking, and meta-programming. However, the implementation of these mechanisms is unique, so it is difficult for beginners to learn Prolog, especially if they have experience in procedural programming languages like C and BASIC. Extra-logical predicates, such as ‘assertz’ and ‘retract’, enable knowledge data to be altered dynamically and meta-programs to be created but they can also make Prolog programs difficult to understand and debug.

Visualization using program diagrams can effectively facilitate the understanding and debugging of programs. The Transparent Prolog Machine, a well-known Prolog visualization system[1, 2], displays the structure of a pure Prolog program as a tree with AND/OR branches (an AND/OR tree) and depicts the states of the various goals as symbols at its nodes. Other visualization and debugging systems for Prolog and (constraint) logic programming languages (e.g. [3–5]) also use AND/OR trees. However, it is not easy to correlate the content of a Prolog program with that of its corresponding AND/OR tree because the structure of the clauses of the Prolog program and their representations in the AND/OR tree are different.

This paper is the first report that describes a Prolog visualization system, which we have implemented in SICStus Prolog [6] to support Prolog programming education. The system uses Logichart diagrams to visualize Prolog programs. Logichart is a program diagram description language that we developed.
Prolog Visualization System Using Logichart Diagrams

to help visualize the execution flow of Prolog programs [7–9]. A Logichart diagram has a tree-like structure, as shown in Fig. 1, with the following two features. (1) The head and body goals that compose each clause are aligned horizontally, and (2) a calling goal and the heads of the clauses that it calls are aligned vertically. Feature (1) gives clauses in a Prolog program and their representations in the corresponding Logichart diagram a similar structure so that it is easier to see the correspondences between them. Feature (2) makes it easier to understand the relationships between related clauses, because they are vertically adjacent.

The system has three functions: (1) it animates Prolog execution (goal calling, success, and failure) on the Logichart diagram, (2) it visualizes a dynamic change in a Prolog program in real time by calling extra-logical predicates, such as ‘assertz’ and ‘retract’, and (3) it displays variable substitution processes in a text widget in real time.

Fig. 1. Logichart diagram

2 Logichart diagrams and Logichart-AGG

Logichart diagrams have been developed to represent computation, which is the response of a Prolog program to a query, as an intelligible diagram [7–9]. A Prolog program is visualized by using Logichart diagrams as follows. For goal sequences ‘G₁, G₂, · · ·, Gₙ’ of a user’s query, the system adds the clause ‘prolog_program :- G₁, G₂, · · ·, Gₙ.’ to the program. The node labeled ‘prolog_program’ corresponding to the head of this clause is the root node of the Logichart diagram. The head and body goals composing a clause are horizontally aligned from left to right according to the Prolog syntax. However, a goal and clauses, which have heads that can be unified with it, are vertically aligned from top to bottom in the
same order as the clauses in the Prolog program. In this manner, the Logichart is
defined based on the Prolog syntax and the evaluation rule of Prolog interpreters
(leftmost derivation, depth-first search). As a result, a Logichart diagram is rel-
atively easy to understand, and correspondence with the source Prolog program
is clearly presented. It must be noted that the Logichart diagram includes an
execution tree (whose root node is labeled ‘prolog_program’) for the user’s query
as a subtree.

Figure 1 shows a Logichart diagram that corresponds to the Prolog program
shown below and the query ‘?- test(X,Y,Z).’.

test(X,Y,Z) :- appendList(X,Y,Z),
             write((X,Y,Z)),nl.
test(_,_,_) :- write(end),nl.
appendList([],X,X).
appendList([X|L1],L2,[X|List]) :-
                appendList(L1,L2,List).

Node labels used in Logichart diagrams are shown in Fig. 2. The heads, body
goals, built-in predicates, and user-defined predicates are clearly and distinctively
depicted using these node labels. The label ‘recursive clause’ enables depicting a
Prolog program including recursive goals within a finite-area Logichart diagram.

```
[ ]  head
[ ]  goal
[ ]  negated goal
[ ]  recursive clause
\   \    unit clause
\   \    built-in predicate
(   )  goal sequence
\   \    if goal
  \   \    THEN label
  \   \    ELSE label
```

**Fig. 2.** Node labels of Logichart diagrams

We formalized Logichart-AGG [7–9], an attribute graph grammar defined
for specifying the syntax and layout rules of Logichart diagrams. It consists
of a context-free graph grammar whose productions are formalized to specify the graph-syntax rules of Logichart diagrams. It also uses semantic rules, which are defined so that they extract the layout information needed to display a Logichart diagram as the attributes attached to node labels. The semantic rules are formalized so as to obtain Logichart diagrams for a minimum-area layout under a specific layout constraint. The Logichart-AGG specifications are very concise and consist of 13 productions associated with 88 semantic rules. Some of the productions and their associated semantic rules in the Logichart-AGG are illustrated in Figs. 3 to 5.

**Production**

\[
[\text{prolog\_program}]_0 := \begin{array}{c}
\text{prolog\_program} \\
\text{goal}
\end{array}^{e,h} \quad C = \phi
\]

**Semantic Rules**

- \(\pi_x(1) = \text{RootX}\)
- \(\pi_y(1) = \text{RootY}\)
- \(\pi_x(2) = \text{RootX} + \text{get\_width}(1) + \text{GapX}\)
- \(\pi_y(2) = \text{RootY}\)
- \(\text{subtree\_width}(0) = \text{get\_width}(1) + \text{GapX} + \text{subtree\_width}(2)\)
- \(\text{subtree\_depth}(0) = \max(\text{get\_depth}(1), \text{subtree\_depth}(2))\)

**Fig. 3.** Rules used to rewrite initial node ‘[prolog\_program]’

Figure 3 shows the production and semantic rules to rewrite the initial node ‘[prolog\_program]’. These rules are formalized to represent queries given in the Prolog syntax, and the nonterminal node ‘goal’ in the right-hand-side graph corresponds to the query. A graph that is isomorphic to the right-hand-side graph of Production 1 is derived by applying this production to the initial node. Semantic rules \(\pi_x(1) = \text{RootX}\) and \(\pi_y(1) = \text{RootY}\) mean that the x-coordinate of node ‘1’ is ‘RootX’ and that the y-coordinate of node ‘1’ is ‘RootY’. Semantic rule \(\pi_x(2) = \text{RootX} + \text{get\_width}(1) + \text{GapX}\) means that the x-coordinate of node ‘2’ is equal to ‘RootX’ plus the width of the node labeled “prolog\_program” plus the horizontal gap ‘GapX’. Semantic rule \(\pi_y(2) = \text{RootY}\) means that the y-coordinate of node ‘2’ is ‘RootY’. The root node “prolog\_program” and the subdiagram derived from the nonterminal node ‘2’ labeled ‘goal’ are aligned with a horizontal separation of ‘GapX’ by these semantic rules.

The production and semantic rules shown in Fig. 4 are as formalized for the ‘and’ operation on Prolog goals. A node labeled ‘goal’ is replaced with a graph that is isomorphic to the right-hand-side of Production 2 by applying this production. Semantic rule \(\pi_x(2) = \pi_x(1) + \text{subtree\_width}(1) + \text{GapX}\) means that the x-coordinate of node ‘2’ is equal to the x-coordinate of node ‘1’ plus the width of the subdiagram derived from node ‘1’ plus the horizontal gap ‘GapX’.

```prolog
[prolog_program]_0 := \begin{array}{c}
\text{prolog\_program} \\
\text{goal}
\end{array}^{e,h} \quad C = \phi
```
Production
\[ [\text{goal}]_0 \xrightarrow{\ V \ H \ L \ \text{subtree}} [\text{goal}]_2, \]
\[ C = \{(#,e,e,1,in), (#,e,e,2,out), (#,h,h,1,in), \]
\[ (#,h,h,2,out), (#,v,v,1,in), (#,v,v,1,out)\}\n
Semantic Rules
\[ \pi_x(1) = \pi_x(0), \quad \pi_y(1) = \pi_y(0), \]
\[ \pi_x(2) = \pi_x(1) + \text{subtree\_width}(1) + \text{GapX}, \quad \pi_y(2) = \pi_y(1), \]
\[ \text{subtree\_width}(0) = \text{max}(\text{subtree\_width}(1), \text{subtree\_width}(2)), \]
\[ \text{subtree\_depth}(0) = \text{max}(\text{subtree\_depth}(1), \text{subtree\_depth}(2)).\]

Fig. 4. Rules formalized for ‘and’ operation on Prolog goals

Therefore, goals connected by the operator ‘and’ are aligned with a separation of ‘GapX’ horizontally.

Production
\[ [\text{goal}]_0 \xrightarrow{\ V \ H \ L \ \text{subtree}} [\text{call\_goal}]_1, \]
\[ C = \{(#,e,e,1,in), (#,e,e,1,out), \]
\[ (#,h,h,1,in), (#,h,h,1,out), \]
\[ (#,v,v,2,out), (#,v,v,2,out)\}\n
Semantic Rules
\[ \pi_x(1) = \pi_x(0), \quad \pi_y(1) = \pi_y(0), \]
\[ \pi_x(2) = \pi_x(1) + \text{get\_depth}(1) + \text{GapY}, \quad \pi_y(2) = \pi_y(1) + \text{get\_depth}(1) + \text{GapY}, \]
\[ \text{subtree\_width}(0) = \text{max}(\text{get\_width}(1), \text{subtree\_width}(2)), \]
\[ \text{subtree\_depth}(0) = \text{get\_depth}(1) + \text{GapY} + \text{subtree\_depth}(2).\]

Fig. 5. Rules formalized for call of goal

The production and semantic rules shown in Fig. 5 are formalized for the call of a goal. The semantic rule \[\pi_y(2) = \pi_y(1) + \text{get\_depth}(1) + \text{GapY}\] means that the y-coordinate of node ‘2’ is equal to the y-coordinate of node ‘1’ plus the depth of node ‘1’ plus the vertical gap ‘GapY’. Therefore, a calling goal and the clause heads of the goals called by it are aligned with a separation of ‘GapY’ vertically.

Implementing the Prolog visualization system in complete accord with Logichart-AGG guarantees that, for any correct Prolog program, the corresponding Logichart
The system has some features specifically designed to support Prolog programming learning.

### 3.1 Visual trace of Prolog execution

The system visually traces Prolog execution on a Logichart diagram in real time. A tracer implemented using a meta-interpreter technique displays the goal-execution process on the Logichart diagram; the color of each node changes depending on the goal state, i.e., the node corresponding to a goal that has been called becomes green, a goal that has succeeded becomes blue, and a goal that has failed becomes red.

The tracer has two modes, i.e., one-step and automatic. In the one-step mode, each step of goal calling is executed and visualized in the Logichart diagram whenever the ‘go’ button is pressed, while in the automatic mode the query is
executed completely and the final state is visualized in the Logichart diagram. The effect of cut (‘!’) on a program execution flow is clearly and intelligibly visualized in the one-step mode on a Logichart diagram. Figure 8 shows a screen shot of a visual trace of the Prolog program shown below and the query ‘?- f.’.

```prolog
f :- g, !, h, fail.
f.
g :- write(a), nl.
g :- write(b), nl.
h.
```

![Logichart Diagram](image)

**Fig. 8.** Visual trace of Prolog execution

If the goals specified in a query are satisfied, and if variables are included in the query, then the system displays a MessageBox window that asks whether the system executes backtracking or not. If the user clicks the ‘yes’ button, the system executes backtracking and the backtracking process is visualized on the Logichart diagram.

### 3.2 Dynamic change in the Logichart diagram

Dynamic changes in a Prolog program by calling extra-logical predicates, such as ‘asserta’, ‘assertz’, and ‘retract’, are visualized in real time using a Logichart diagram. Figures 9 and 10 show a dynamic change in the Prolog program shown below and the query ‘?- f.’ by calling ‘assertz((g(Y) :- k(Y))).

```prolog
f :- g(X), h(X), g(X).
g(a).
h(Y) :- assertz((g(Y) :- k(Y))).
k(X) :- write(X).
```

Clauses retracted by calling ‘retract’ are not eliminated but depicted with crosses in a Logichart diagram. For the Prolog program shown below and the query ‘?- f.’, a screen shot of a Logichart diagram after calling ‘retract((g :- write(a)))’ is shown in Fig. 11.
f :- g, h, g, fail.
g :- write(a).
g :- write(b).
h :- retract((g :- write(X))).

If the nodes corresponding to those of the retracted clauses are eliminated from the Logichart diagram, the backtracking process in the Logichart diagram becomes impossible to visualize. Depicting retracted clauses with crosses enables visualizing the backtracking process in the Logichart diagram.

3.3 Display of variable substitutions

The variable-substitution process is displayed in a text widget in real time. Figure 12 shows a text widget displaying variable substitution for each node. We are now implementing a function to display variable-substitution information within the Logichart diagram.
4 Conclusions

We presented a Prolog visualization system that is implemented in complete accord with Logichart-AGG. Logichart diagrams make it easy to understand the Prolog execution flow. A remarkable feature of the system is that it visualizes the dynamic alteration of a Logichart diagram by calling extra-logical predicates, such as ‘assertz’ and ‘retract’.

We are currently developing fine- and coarse-grained (Logichart) diagrams as proposed in [1] to visualize and navigate in large execution trees. These will help to develop the Prolog visualization system into a practical Prolog developing environment. The usefulness of our Prolog visualization system for Prolog programming education needs to be empirically investigated.

References


A Semantics-Aware Editing Environment
for Prolog in Eclipse

Jens Bendisposto, Ian Endrijautzki, Michael Leuschel, and David Schneider

Softwaretechnik und Programmiersprachen
Institut für Informatik
Universität Düsseldorf
{bendisposto,leuschel}@cs.uni-duesseldorf.de

1 Introduction

The open-source Eclipse platform has become hugely popular as an integrated development environment (IDE) for Java, mainly because of its editing features. The Java IDE comes with code highlighting, hover information, code completion, quick fixes and many more. It supports the development process by highlighting problematic aspects of the code and providing a convenient and robust way to improve it using refactorings. All those features have a common goal: let the user stay focused on his task, writing code.

It would be very useful for the logic programming community to obtain an IDE with similar features for the development of Prolog programs.

However, writing an industrial strength integrated development environment like Eclipse for another language from scratch is a very difficult and complex task. Fortunately, in previous work we have developed BE4 [3], a framework for building semantic aware editors built on top of Eclipse and the Rigorous Open Development Environment for Complex Systems (RODIN). BE4 contains a reliable, multi phase build process that can be used for arbitrary languages. The build process was designed with the purpose of integrating different tools, even proprietary compilers into BE4. In particular, it is not restricted to Java. BE4 also contains a toolset to implement state-of-the-art editors, providing features such as syntactic and semantic highlighting, code completion, outline view, hover information, quick fixes and semantic checks. Finally BE4 already comes with language plug-ins for classical B, CSP, Promela and SableCC, and a such its applicability has already been tested on a variety of source languages.

In this paper we present a Prolog plugin for Eclipse based upon BE4, and providing many features such as semantic-aware syntax highlighting, outline view, error marking, content assist, hover information, documentation generation, and quick fixes. The plugin makes use of a Java parser for full Prolog with an integrated Prolog engine, and can be extended with further semantic analyses, e.g., based on abstract interpretation.

2 Features

Beside the obligatory syntax highlighting, our tool ProClipse offers a wide variety of features helping the user develop and adapt his or her Prolog programs. Our
features are based on a full syntactical and semantical analysis (i.e., they are not derived on approximate solutions using regular expressions). Thus far, the following features have been implemented:

Outline View: ProClipse creates an outline view which represents an overview of the Prolog file or module, containing exported or non-exported predicates, defined DCG predicates and import directives. Each item in the outline view can be used to quickly access the respective lines of the Prolog code.

Error presentation: Syntax and semantic errors are highlighted in the editor view. A wiggly line underlines the erroneous part of the source code and each line containing an error is tagged with an error marker. The problems view lists all errors of all Prolog files in an Eclipse Prolog project and can also be used to directly recall an erroneous source code line.

Content Assist: To improve faster coding a content assist has been implemented which offers content sensitive proposals to automatically complete the word the developer types. As can be seen in Figure 1, content assist can also be used to retrieve information about predicates, like their synopsis and usage, and present an overview of the available predicates, modules, DCGs and more.

Quick Fixes: Quick fixes offer the possibility to directly auto-correct erroneous source code via a mouse click. Based on the type of error, a suitable set of fixes
is offered\(^1\). For instance, if the developer calls an unknown predicate (which is presented as an error in the editor view), ProClipse will look for Prolog modules and databases exporting this predicate and offer to import one of these. This is illustrated in Figure 2.

\[\text{Fig. 2. ProClipse Screenshot - Error presentation and Quick Fix}\]

**PrologDoc:** Inspired by the documentation generator JavaDoc from Sun Microsystems for Java, we created PrologDoc. To attach a documentation to a predicate, a comment field containing PrologDoc entries\(^2\) must be written above the first appearance of a clause defining this predicate. Documentation can also be attached to entire modules, in which case the PrologDoc comment field must be written above the module definition. Our implementation can also extract a PrologDoc summary for a complete project. This generated summary is presented in HTML format (see Figure 3), with the ability to directly navigate into

\(^1\) Quick fixes can be invoked by right clicking the error in the problems view and selecting the "Quick Fix" entry

\(^2\) Default PrologDoc entries are: 'Author:', 'Arguments:', 'Description:'.

the summaries of imported Prolog modules or files. It shows an outline of all defined predicates, including their synopsis.

Fig. 3. ProClipse Screenshot - PrologDoc Summary

Text Hover: ProClipse offers two types of text hover which can deliver quick information of a lexical token in our editor view. For instance, if a developer wants to know how a certain predicate has been defined, he simply has to point the mouse at the predicate. The text hover can also present the synopsis and the arguments of a Prolog built-in, the exported predicates of imported Prolog modules or files, and the definition of an user-defined operator. By additionally pressing the shift key the PrologDoc hover is shown, which displays the PrologDoc entries of a predicate or module.

3 Architecture and Implementation

3.1 BE4 Phases

Our development is based on BE4 [3], which contains a multi-phase parsing and analysis framework. The architecture was designed for extensibility by new, as of yet unknown, plug-ins.

As shown in Figure 4, our tool is decomposed into four phases.
The first phase is the parsing phase which constructs an abstract syntax tree (AST). To that end we have written a Parser and Engine for full Prolog in Java (see Subsection 3.2 below).

Phases II and III decorate this AST and then combine various ASTs from multiple files to perform global analysis.

The last phase (IV) generates the relevant information for the semantics aware editor to work and decorates the source files, e.g., generating Eclipse markers\(^3\).

![Fig. 4. Phases of the building framework](image)

### 3.2 Parser

The centerpiece of a semantic-aware editor for programming languages is a parser that generates a model from source-code. In order to provide seamless integration into Eclipse and BE4, we have written a Prolog parser in Java.

Our parser generates a fully typed\(^4\) parse tree of the processed Prolog code. Note that in order to parse full Prolog with directives and operator declarations, a Prolog engine is required. We have hence also developed a simple Prolog engine in Java, with support for a basic set of built-ins. Our parser and engine fully support the dynamic operator definitions and the dynamic grammar associated to them. Our Prolog parsing framework provides a mechanism to attach custom post-processing steps to the results produced by the parser, so they can be used in a flexible way in different contexts.

\(^3\) [http://www.eclipse.org/articles/Article-Mark My Words/mark-my-words.html](http://www.eclipse.org/articles/Article-Mark My Words/mark-my-words.html)

\(^4\) Note that we do type the individual nodes in the abstract syntax tree, but we do not try to infer Prolog types for arguments of predicates.
The Prolog parser is built of a set of different components which also represent the different steps of the evaluation process. These are the lexical analysis, the parser and a series of post-processing steps called engines.

**Lexical analysis** The lexer implemented for Prolog was created using JFLex and is based on the definition of the language as provided in the SICStus Prolog Users Manual [5]. Every token type is represented by an instance of a corresponding Java class and holds information about the source text it matched and where on the input stream it appeared.

Prolog tokens can have different meanings depending on their position in the input stream and the state of the program (itself depending on operations performed in previous steps). The lexer takes this information into account when generating lexical tokens, performing a state aware token generation, so that the same text can be represented by different tokens depending on the context. E.g. the '+' atom would be represented as an atom or an operator, depending of the current operator definitions stored in the internal database.

**Syntactic Analysis** The parser processes the token stream on a sentence by sentence basis. Every time the abstract syntax tree for a sentence is constructed, it is dispatched to the post-processing steps for evaluation, before resuming the parsing process. This way the parser can take into account possible modifications to the environment done by the evaluation of the preceding sentence, such as an operator definition.

The algorithm used in the parser follows the shift reduce parsing principle, with one token lookahead [1]. Instead of using an approach based on DFA as described in [2], this parser takes advantage of the simple structure of the Prolog grammar, which makes it unnecessary to track multiple possible productions for a given input at the same time. Our algorithm is based on the current token and as necessary on the previous and next tokens on the input stream.

This approach allows to provide a fine-grained and fully typed AST, providing as much information as possible about the language tokens in the context they appeared.

**Post-Processing, The Prolog Runtime** After a sentence has been parsed, the AST representing it is dispatched to the post-processor. The post-processor is built of a chain of objects called engines. These engines are responsible of performing different analysis and execution steps on the AST and to pass the AST to next engine registered in the chain. There are different engines with different responsibilities, such as executing the code, storing it in the internal term database or providing an interactive read eval loop as a command line interface. There is a special engine responsible for evaluation directives, every time a directive passes this engine the directive is executed, so additional files get loaded and modifications to internal settings are executed.
Error Recovery To provide as much information as possible in case of an error, the parser supports a simple form error recovery by dropping tokens until it can resume the parsing process normally. All errors are collected and made available to the user, so that they can be used - for instance - to highlight the errors in an IDE. This enables our plugin to detect multiple errors in a source file (and not just the first error).

4 Related Work and Conclusion

Various Prolog systems come with some support for convenient editing and development. For example, SICStus Prolog and Ciao Prolog are distributed with an Emacs-mode for Prolog. SWI Prolog has an Emacs clone based on its XPCE package as well as an editor for Windows. Visual Prolog and LPA Prolog both have a custom editor for Windows.

Other related work is [4] which present a refactoring tool for Prolog based on the editor VIM. In future we plan to add refactoring capabilities to our plugin. While existing Prolog documentation tools such as PrologDoc provide a wide range of documentation options, our tool currently only provides a subset of these options, but has the advantage of being fully integrated with eclipse. In future we plan to integrate PrologDoc with our documentation tools to allow more detailed documentation.

On the Eclipse side we are aware of the following three plugins.

- Prolog Plugin developed in 2003/2004. It is unclear whether this Plugin is still maintained and it only seems to provide syntax highlighting and consulting a Prolog interpreter.
- PDT This plugin supports more features and links up with SWI Prolog. Quick fixes, document generation and hover information do not seem to be supported yet.
- ProDevTools only supports SWI Prolog and does not yet seem to support quick fixes and document generation.

All three plugins definitely have potential. One distinguishing aspect of our plugin is the language-independent BE4 framework, which will be maintained in the foreseeable future. Improvements made to BE4 for other languages, such as B or CSP, will feed back into the Prolog plugin as well.

We also plan to keep our editor largely independent from any particular Prolog system and plan to integrate more analysis information based on abstract interpretation. While it is easy to create analysis tools written in Java, in future we will investigate ways of integrating analysis tools written in prolog without having to dismiss the independence from any Prolog system.

We also plan to integrate termination analysis and partial evaluation tools into our plugin.

5 http://prologdoc.sourceforge.net/
7 http://roots.iai.uni-bonn.de/research/pdt/
8 http://prodevtools.sourceforge.net/
References

On the Generation of Test Data for Prolog by Partial Evaluation

Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla

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

Abstract. In recent work, we have proposed an approach to Test Data Generation (TDG) of imperative bytecode by partial evaluation (PE) of CLP which consists in two phases: (1) the bytecode program is first transformed into an equivalent CLP program by means of interpretive compilation by PE, (2) a second PE is performed in order to supervise the generation of test-cases by execution of the CLP decompiled program. The main advantages of TDG by PE include flexibility to handle new coverage criteria, the possibility to obtain test-case generators and its simplicity to be implemented. The approach in principle can be directly applied for TDG of any imperative language. However, when one tries to apply it to a declarative language like Prolog, we have found as a main difficulty the generation of test-cases which cover the more complex control flow of Prolog. Essentially, the problem is that an intrinsic feature of PE is that it only computes non-failing derivations while in TDG for Prolog it is essential to generate test-cases associated to failing computations. Basically, we propose to transform the original Prolog program into an equivalent Prolog program with explicit failure by partially evaluating a Prolog interpreter which captures failing derivations w.r.t. the input program. Another issue that we discuss in the paper is that, while in the case of bytecode the underlying constraint domain only manipulates integers, in Prolog it should properly handle the symbolic data manipulated by the program. The resulting scheme is of interest for bringing the advantages which are inherent in TDG by PE to the field of logic programming.

1 Introduction

Test data generation (TDG) aims at automatically generating test-cases for interesting test coverage criteria. The coverage criteria measure how well the program is exercised by a test suite. Examples of coverage criteria are: statement coverage which requires that each line of the code is executed; path coverage which requires that every possible trace through a given part of the code is executed; etc. There are a wide variety of approaches to TDG (see [22] for a survey). Our work focuses on glass-box testing, where test-cases are obtained from the concrete program in contrast to black-box testing, where they are deduced from a specification of the program. Also, our focus is on static testing, where we assume no knowledge about the input data, in contrast to dynamic approaches [6] which execute the program to be tested for concrete input values.
The standard approach to generating test-cases statically is to perform a symbolic execution of the program \([18,14,11]\), where the contents of variables are expressions rather than concrete values. The symbolic execution produces a system of constraints consisting of the conditions to execute the different paths. This happens, for instance, in branching instructions, like if-then-else, where we might want to generate test-cases for the two alternative branches and hence accumulate the conditions for each path as constraints. The symbolic execution approach is usually combined with the use of constraint solvers in order to handle the constraints systems by solving the feasibility of paths and, afterwards, to instantiate the input variables.

TDG for declarative languages has received comparatively less attention than for imperative languages. In general, declarative languages pose different problems to testing related to their own execution models, like laziness in functional programming (FP) and failing derivations in constraint logic programming (CLP). The majority of existing tools for FP are based on black-box testing (see e.g. \([4]\)). An exception is \([7]\) where a glass-box testing approach is proposed to generate test-cases for Curry. In the case of CLP, test-cases are obtained for Prolog in \([16,3,21]\); and very recently for Mercury in \([5]\). Basically the test-cases are obtained by first computing constraints on the input arguments that correspond to execution paths of logic programs and then solving these constraints to obtain test inputs for such paths.

In recent work \([2]\), we have proposed to employ existing partial evaluation (PE) techniques developed for CLP in order to automatically generate test-case generators for glass-box testing of bytecode. PE \([13]\) is an automatic program transformation technique which has been traditionally used to specialise programs w.r.t. a known part of its input data and, as Futamura predicted, can also be used to compile programs in a (source) language to another (object) language (see \([8]\)). The approach to TDG by PE of \([2]\) consists of two independent CLP PE phases. (1) First, the bytecode is transformed into an equivalent (decompiled) CLP program by specialising a bytecode interpreter by means of existing PE techniques. (2) A second PE is performed in order to supervise the generation of test-cases by execution of the CLP decompiled program. Interestingly, it is possible to employ control strategies previously defined in the context of CLP PE in order to capture coverage criteria for glass-box testing of bytecode. A unique feature of this approach is that, this second PE phase allows generating not only test-cases but also test-case generators. Another important advantage is that, in contrast to previous work to TDG of bytecode, it does not require devising a dedicated symbolic virtual machine.

In this work, we study the application of the above approach to TDG by means of PE to the Prolog language. Compared to TDG of an imperative language \([2]\), dealing with Prolog brings in as the main difficulty to generate test-cases associated to failing computations. This happens because an intrinsic feature of PE is that it only produces results associated to the non-failing derivations. While this is what we need for TDG of an imperative language (like bytecode above), we now want to capture non-failing derivations in Prolog and
still rely on a standard partial evaluator. Our proposal is to transform the original Prolog program into an equivalent Prolog program with explicit failure by partially evaluating a Prolog interpreter which captures failing derivations w.r.t. the input program. This transformation is done in the phase (1) above. As another difference, in the case of bytecode, the underlying constraint domain only manipulates integers. However, the above phase (2) should properly handle the data manipulated by the program in the case of Prolog. Compared to existing approaches to TDG of Prolog [3,16], our approach basically is of interest for bringing the advantages which are inherent in TDG by PE to the field of Prolog:

(i) It is more powerful in that we can produce test-case generators which are CLP programs whose execution in CLP returns further test-cases on demand without the need to start the TDG process from scratch;

(ii) It is more flexible, as different coverage criteria can be easily incorporated to our framework just by adding the appropriate local control to the partial evaluator.

(iii) It is simpler to implement compared to the development of a dedicated test-case generator, as long as a CLP partial evaluator is available.

The rest of the paper is organized as follows. In the next section, we give some basics on PE of logic programs and describe in detail the approach to TDG by PE proposed in [2]. Sect. 3 discusses some fundamental issues like the Prolog control-flow and the notion of computation path. Then, Sect. 4 describes the program transformation to make failure explicit, Sect. 5 outlines existing methods to properly handle symbolic data during the TDG phase, and finally Sect. 6 concludes and discusses some ideas for future work.

2 Basics of TDG by Partial Evaluation

In this section we recall the basics of partial evaluation of logic programming and summarize the general approach of relying on partial evaluation of CLP for TDG of an imperative language, as proposed in [2].
2.1 Partial Evaluation and its Application to Compilation

We assume familiarity with basic notions of logic programming and partial evaluation (see e.g. [9]). Partial evaluation is a semantics-based program transformation technique which specialises a program w.r.t. given input data, hence, it is often called \textit{program specialisation}. Essentially, partial evaluators are non-standard interpreters which evaluate goals as long as termination is guaranteed and specialisation is considered profitable. In logic programming, the underlying technique is to construct (possibly) \textit{incomplete} SLD trees for the set of atoms to be specialised. In an incomplete tree, it is possible to choose \textit{not} to further unfold a goal. Therefore, the tree may contain three kinds of leaves: failure nodes, success nodes (which contain the empty goal), and non-empty goals which are not further unfolded. The latter are required in order to guarantee termination of the partial evaluation process, since the SLD being built may be infinite. Even if the SLD trees for fully instantiated initial atoms (as regards the input arguments) are finite, the SLD trees produced for partially instantiated initial atoms may be infinite. This is because the SLD for partially instantiated atoms can have (infinitely many) more branches than the actual SLD tree at run-time.

The role of the \textit{local control} is to determine how to construct the (incomplete) SLD trees. In particular, the \textit{unfolding rule} decides, for each resolvent, whether to stop unfolding or to continue unfolding it and, if so, which atom to select from the resolvent. On the other hand, partial evaluators need to compute SLD-trees for a number of atoms in order to ensure that all atoms which appear in non-failing leaves of incomplete SLD trees are “covered” by the root of some tree (this is known as the closedness condition of partial evaluation [9]). The role of the \textit{global control} is to ensure that we do not try to compute SLD trees for an infinite number of atoms. The usual way of achieving this is by applying an \textit{abstraction operator} which performs “generalizations” on the atoms for which SLD trees are to be built. The global control returns a set of atoms $T$. Finally, the partial evaluation can then be systematically extracted from the set $T$ (see [9] for details).

Traditionally, there have been two different approaches regarding the way in which control decisions are taken, \textit{on-line} and \textit{off-line} approaches. In \textit{online} PE, all control decisions are dynamically taken during the specialisation phase. In \textit{offline} PE, a set of previously computed annotations (often manually provided) gives information to the control operators to decide, 1) when to stop unfolding (\textit{memoise}) in the local control, and 2) how to perform generalizations in the global control.

The development of PE techniques has allowed the so-called “interpretative approach” to compilation which consists in specialising an interpreter w.r.t. a fixed object code. Interpretive compilation was proposed in Futamura’s seminal work [8], whereby compilation of a program $P$ written in a (source) programming language $L_S$ into another (object) programming language $L_O$ is achieved by partially evaluating an interpreter for $L_S$ written in $L_O$ w.r.t. $P$. The advantages of interpretive (de-)compilation w.r.t. dedicated (de-)compilers are well-known and discussed in the PE literature (see, e.g., [1]). Very briefly, they include: \textit{flexibility,
it is easier to modify the interpreter in order to tune the decompilation (e.g.,
observe new properties of interest); easier to trust, it is more difficult to prove
that ad-hoc decompilers preserve the program semantics; easier to maintain,
new changes in the language semantics can be easily reflected in the interpreter.

2.2 A General Scheme to TDG of Imperative Languages by PE

In recent work, we have proposed an approach to Test Data Generation (TDG)
by PE of CLP [2] and used it for TDG of bytecode. The approach is generic
in that the same techniques can be applied to TDG other both low and high-
level imperative languages. In Figure 1 we overview the main two phases of
this technique. In Phase I, the input program written in some (imperative)
language $L$ is compiled into an equivalent CLP program $P_{CLP}$. This compilation
can be achieved by means of an ad-hoc decompiler (e.g., an ad-hoc decompiler
of bytecode to Prolog [17]) or, more interestingly, can be achieved automatically
by relying on the first Futamura projection by means of PE for logic programs
as explained above (e.g., [12,1,10]).

Now, the aim of Phase II is to generate test-cases which traverse as many
different execution paths of $P_L$ as possible, according to a given coverage criteria.
From this perspective, different test data will correspond to different execution
paths. With this aim, rather than executing the program starting from different
input values, the standard approach consists in performing symbolic execution
such that a single symbolic run captures the behavior of (infinitely) many input
values. The central idea in symbolic execution is to use constraint variables
instead of actual input values and to capture the effects of computation using
constraints. Hence, the compilation from $L$ to CLP allows us to use the standard
CLP execution mechanism to carry out this phase. In particular, by running the
$P_{CLP}$ program without input values, each successful execution corresponds to a
different computation path in $P_L$.

Rather than relying on the standard execution mechanism, we have proposed
in [2] to use PE of CLP to carry out Phase II. Essentially, we can rely on a CLP
partial evaluator which is able to solve the constraint system, in much the same
way as a symbolic abstract machine would do. Note that performing symbolic
execution for TDG consists in building a finite (possibly unfinished) evaluation
tree by using a non-standard execution strategy which ensures both a certain
coverage criterion and termination. This is exactly the problem that unfolding
rules, used in partial evaluators of (C)LP, solve. In essence, partial evaluators
are non-standard interpreters which receive a set of partially instantiated atoms
and evaluate them as determined by the so-called unfolding rule. Thus, the
role of the unfolding rule is to supervise the process of building finite (possibly
unfinished) SLD trees for the atoms. This view of TDG as a PE problem has
important advantages. First, we can directly apply existing, powerful, unfolding
rules developed in the context of PE. Second, it is possible to explore additional
abilities of partial evaluators in the context of TDG. In particular, the generation
of a residual program from the evaluation tree returns a program which can be
used as a test-case generator, i.e., a CLP program whose execution in CLP
returns further test-cases on demand without the need to start the TDG process from scratch. In the rest of the paper, we study the application of this general approach to TDG of Prolog programs.

3 Computation Paths for Test Data Generation of Prolog

As we have already mentioned, test data generation is about producing test-cases which traverse as many different execution paths as possible. From this perspective, different test data should correspond to different execution paths. Thus, a main concern is to specify the computation paths for which we will produce test-cases. This requires first to determine the control flow of the considered language. In this section, we aim at defining the control flow of Prolog programs that we will use for TDG. Test data will be generated for the computation paths in the control flow.

3.1 The Control Flow of Prolog

As usual a Prolog program consists of a set of predicates, where each predicate is defined as a sequence of clauses of the form \( H \leftarrow B_1, \ldots, B_m \) with \( m \geq 0 \). A predicate is univocally determined by its predicate signature \( p/n \), being \( p \) the name of the predicate and \( n \) its arity. Throughout the rest of the paper we will consider Prolog programs with the following features:

- Rules are normalized, i.e., arguments in the head of the rule are distinct variables. The corresponding bindings will appear explicitly in the body as unifications.
- Atoms appearing in the bodies of rules can be: unifications (considered as builtins), calls to defined predicates, term checking builtins (==/2, \\==/2, etc), and arithmetic builtins (is/2, <=/2, =</2, etc). Other typical Prolog builtins like fail/0, !/0, if/3, etc, have been deliberately left out to simplify the presentation.
- All predicates must be moded and well-typed. We will assume the existence of a “:- pred” declaration associated with each predicate specifying the type expected for each argument (see as example the declarations in Fig. 2). Note that this assumption is sensible in the context of TDG (as the aim is the automatic generation of test input). Also, it should not be a limitation as analyses that can automatically infer this information exist.

The control flow in Prolog programs is significantly more complex than in traditional imperative languages. The declarative semantics of Prolog implies some additional features like: 1) several forms of backtracking, induced by the failure of a sub-goal, or by non-deterministic predicates; or 2) forced control flow change by the predicate “cut”. Traditionally, control-flow graphs (CFGs for short) are used to statically represent the control-flow of programs. Typically, in a CFG, nodes are blocks containing a set of sequential instructions, and edges represent the flows that the program can follow w.r.t. the semantics of the corresponding
programming language. In the literature, CFGs for Prolog (and Mercury) have been used for the aim of TDG in [16,21] ([5] for Mercury). In particular, CFGs determine the computation paths for which test-cases will be produced. Our framework relies on the CFGs of [16,21] which are known as p-flowgraph’s.\(^3\) As will be explained later, there are some differences between these CFGs and the ones in [5] which lead to different test-cases.

Figure 2 depicts the Prolog code together with the corresponding CFGs for predicates `foo/2` and `sorted/1`. Predicate `foo/2`, given a number in its first argument, returns, in the second one, the value `pos` if the number is positive and `zero` if it is zero. If the number is negative, it just fails. Predicate `sorted/1`, given a list of numbers, checks whether the list is strictly sorted, in that case it succeeds, otherwise it fails. The CFGs contain the following nodes:

- a non-terminal node associated to each atom in the body of each clause,
- a set of terminal nodes “\(T_i\)” representing the success of the \(i\)-th clause, and
- the terminal node “\(F\)” to represent failure.

As regards edges, in principle all non-terminal nodes have two output flows, corresponding to the cases where the builtin or predicate call succeeds or fails respectively. They are labeled as “yes” or “no” for builtins (including unifications), and as “\(rs\)” (return-after-success) or “\(rf\)” (return-after-failure) for predicate calls. There is an exception in the case of unifications where one of the arguments is a variable, in which case the unification cannot fail. This can be known statically by using the mode information. See for example nodes “\(Z=pos\)”\(^3\) The difference with the CFGs in [16,21] is that they consider one additional node per clause to explicitly represent the unification of the head of the rule. This is not needed in our case since predicates are normalised.
and “Z=zero” in the foo/2 CFG. Both “yes” and “rs” edges point to the node representing the next atom in the clause or to the corresponding “T_i” node if the atom is the last one. Finally, each “T_i” node has an output edge labeled as “redo” to represent the case in which the predicate is asked for more solutions. All “no”, “rf” and “redo” edges point either to the node corresponding to the first previous non-deterministic call in the same clause, or the first node of the following clause, or the “F” node if no node meets the above conditions. See as an example the “rs” and “rf” edges from the non-terminal node for sorted([Y|R]).

3.2 Generating Test Data for Computation Paths

In order to define the computation paths determined by the CFGs, every edge in every CFG is labeled with a unique natural number. An special edge labeled with “0” and p/n represents the entry of predicate p/n.

Definition 1 (Computation sub-path). Given the CFG for predicate P, a computation sub-path is a sequence of numeric labels (natural numbers) \langle l_1, \ldots, l_n \rangle s.t.:

- l_1 corresponds to either an entry, an “rs”, an “rf” or a “redo” edge,
- l_n leads to a terminal node or to a predicate call, and
- for all consecutive labels l_i, l_j, there exists a node corresponding to a builtin in the CFG of P, for which l_i is an input flow and l_j is an output flow.

Definition 2 (Computation path). Given the CFGs corresponding to the set of predicates defining a program, a computation path (CP for short) for predicate p is a concatenation sp_1 \cdots sp_m (m ≥ 1) of computation sub-paths such that:

- First label in sp_1 is either 0, in which case we say it is a full CP, or corresponds to a “redo” edge, in which case we say it is a partial CP (PCP for short).
- Last label in sp_m leads to a terminal node in the CFG of p. If it is a T node the CP is said to be successful otherwise it is called failing.
- For all sp_k whose last label leads to a node corresponding to a predicate call, cp = sp_{k+1} \cdots sp_{j}, j > k is a CP for the called predicate, and:
  • if cp is successful then the first label in sp_{j+1} corresponds to an “rs” edge,
  • otherwise (cp is failing), it corresponds to an rf edge.
- For all sp_k whose first label corresponds to a “redo” edge flowing from a “T_a” node in the CFG of predicate q, \exists sp_j, j < k, whose first label corresponds either to an entry edge or to a “redo” edge flowing from “T_b”, b < a, of the CFG of q.

If a CP contains at least one label corresponding to a “redo” flow, then the CP is said to be an after-retry CP. The rest of the CPs are first-try CPs.
For example in foo/2, \( p_1 = (0, 1, 2) \) and \( p_2 = (0, 3, 5, 6) \) are first-try successful CPs; \( p_3 = (0, 3, 7) \) is a first-try failing branch; \( p_4 = (0, 1, 2) \cdot (4, 5, 6) \) is an after-retry successful CP (although this one is unfeasible as \( X > 0 \) and \( X = 0 \) are disjoint conditions), and \( p_5 = (0, 1, 2) \cdot (4, 7) \) is an after-retry failing branch. In sorted/1, \( p_6 = (0, 2, 5, 7, 10) \cdot (0, 2, 4) \cdot (11) \) is a first-try successful CP and \( p_7 = (0, 2, 5, 7, 10) \cdot (0, 2, 5, 7, 9) \cdot (12) \) is a first-try failing CP. It is interesting to observe the correspondence between the CPs and the test data that make the program traverse them. In foo/2, \( p_1 \) is followed by goal foo(1, Z), \( p_2 \) by goal foo(0, Z), \( p_3 \) by foo(-1, Z), \( p_4 \) is an unfeasible path, and \( p_5 \) is followed by foo(0, Z) when we ask for more solutions. As regards sorted/1, \( p_6 \) is followed by the goal sorted([0,1]) and \( p_7 \) by sorted([0,1,0]). As we will see in Sect. 5, these will become part of the test-cases that we automatically infer.

A key feature of our CFGs is that they make explicit the fact that after failing with a clause the computation has to re-try with the following clause, unless a non-deterministic call is left behind. E.g., in foo/2 the CFG makes explicit that the only way to get a first-try failing branch is through the CP \( (0, 3, 7) \), hence traversing, and failing in, both conditions \( X > 0 \) and \( X = 0 \). Therefore, a test data to obtain such a behavior will be a negative number for argument \( X \). Other approaches, like the one in [5], do not handle flows after failure in the same way. In fact, in [5], edge “3” in foo/2 goes directly to node “F”. It is not clear if these approaches are able to obtain such a test data. As another difference with previous approaches to TDG of Prolog, we want to highlight that we use CFGs just to reason about the program transformation that will be presented in the following section and, in particular, to clarify which features we want to capture. However, in previous approaches, test-cases are deduced directly from the CFGs.

4 A Program Transformation to Make Failure Explicit

As we outlined in Sect. 1, an intrinsic feature of the second phase of our approach is that it can only produce results associated to non-failing derivations. This is the main reason why the general approach to TDG by PE sketched in Sect. 2 is directly applicable only to TDG of imperative languages. To enable its application to Prolog, we propose a program transformation which makes failure explicit in the Prolog program. The specialisation of meta-programs has been proved to have a large number of interesting applications[9]. Futamura projection’s to derive compiled code, compilers and compiler generators fall into this category. The specialization of meta-interpreters for non-standard computation rules has also been studied. Furthermore, language extensions and enhancements can be easily expressed as meta-interpreters which perform additional operations to the standard computation. In short, program specialisation offers a general compilation technique for the wide variety of procedural interpretations of logic programs. Among them, we propose to carry out our transformation which makes failure in logic programs explicit by partially evaluating a Prolog meta-interpreter which captures failing derivations w.r.t. the original program. First, in Sect. 4.1 we describe such a meta-interpreter emphasizing the Prolog control features which
we want to capture. Then, Sect. 4.2 describes the control strategies which have to be used in PE in order to produce an effective transformation.

4.1 A Prolog Meta-Interpreter to Capture Failure

Given a Prolog program and given a goal, our aim is to define an interpreter in which the computation of the program and goal produces the same results as the ones obtained by using the standard Prolog computation but with the difference that failure is never reported. Instead, an additional argument Answer will be bound to the value “yes”, if the computation corresponds to a successful derivation, and to “no” if it corresponds to a failing derivation. Predicate solve/4 is the main predicate of our meta-interpreter whose first and second arguments are the predicate signature and arguments of the goal to be executed; and its third argument is the answer; by now we ignore the last argument. For instance, the call solve(foo/2,[0,Z],Answer,_) succeeds with \( Z = \text{zero} \) and Answer = yes, and solve(foo/2,[-1,Z],Answer,_) also succeeds, but with Answer = no. The interpreter has to handle the following issues:

1. The Prolog backtracking mechanism has to be explicitly implemented. To this aim, a stack of choice points is carried along during the computation so that:
   - if the derivation fails: (1) when the stack is empty, it ends up with success and returns the value “no”, (2) otherwise, the computation is resumed from the last choice point, if any;
   - if it successfully ends: (1) when the stack is empty, the computation finishes with answer “yes”, (2) otherwise, the computation is resumed from the last choice point.

2. When backtracking occurs, all variable bindings, between the current point and the choice point to resume from, have to be undone.

3. The interpreter has to be implemented in a big-step fashion. This is a requirement for obtaining an effective decompilation. More details are given in Sect. 4.2.

Figure 3 shows an implementation of a meta-interpreter which handles the above issues. The fourth argument of the main predicate solve/4, named TNCPS, contains upon success the total number of choice points not yet considered, whose role will be explained later. The interpreter assumes that the program is represented as a set of pred/2 and clause/3 facts. There is a pred/2 fact per predicate providing its predicate signature, number of clauses and mode information; and a clause/3 fact per clause providing the actual code and clause identifier. Predicate solve/4 basically builds an initial state on S0, by calling build_s0/4, and then delegates on exec/3 to obtain the final state Sf of the computation. The output information, OutVs, is taken from Sf. The state carried along is of the form st(PP,G,CPs,OutVs,Ans,NCPs), where PP is the current program point, G the current goal, CPs is the stack of choice points (list of program points), OutVs the list of variables in G corresponding to the output
solve(P/Ar, Args, Answer, TNCPs) :-
  pred(P/Ar, _),
  build_s0(P/Ar, Args, S0, OutVs),
  exec(Args, S0, Sf),
  Sf = st(_, _, _, OutVs', Answer, TNCPs/_.),
  OutVs' = OutVs.

exec(_, S, Sf) :-
S = st(_, [], [], OutVs, yes, NCPs),
Sf = st(_, [], _, OutVs, yes, NCPs).

exec(_, S, Sf) :-
S = st(_, [], [_|_], OutVs, yes, NCPs),
Sf = st(_, [], _, OutVs, yes, NCPs).

exec(_, S, Sf) :-
S = st(_, [], [], OutVs, no, TNCPs/0),
Sf = st(_, [], _, OutVs, no, TNCPs/0).

exec(_, S, Sf) :-
S = st(_, [CP|CPs], yes, TNCPs/0),
build_retry_state(Args, CP, CPs, TNCPs, S'),
exec(Args, S, Sf).

exec(_, S, Sf) :-
S = st(_, [CP|CPs], no, TNCPs/0),
build_retry_state(Args, CP, CPs, TNCPs, S'),
exec(Args, S, Sf).

exec(Args, S, Sf) :-
S = st(PF, [A|As], CPs, OutVs, yes, TNCPs/ENCPs),
PP = pp(P/Ar, ClId, Pt),
internal(A),
functor(A, A_f, A_ar),
A = ..[A_f|A_args],
next(Pt, Pt'),
solve(A_f/A_ar, Args, Ans, ENCPs'),
TNCPs' is TNCPs + ENCPs',
ENCPs'' is ENCPs + ENCPs',
PP' = pp(P/Ar, ClId, Pt'),
S' = st(PF', As, CPs, OutVs, Ans, TNCPs'/ENCPs''),
exec(Args, S, Sf).

exec(Args, S, Sf) :-
S = st(PF, [A|As], CPs, OutVs, yes, NCPs),
PP = pp(P/Ar, ClId, Pt),
builtin(A),
next(Pt, Pt'),
runc_builtin(PF, A, Ans),
PP' = pp(P/Ar, ClId, Pt'),
S' = st(PF', As, CPs, OutVs, Ans, NCPs),
exec(Args, S, Sf).

Fig. 3. Code of Prolog meta-interpreter to capture failure

parameters of the original goal, Ans the current answer (“yes” or “no”) and
NCPs the number of choice points left behind. A program point is of the form
pp(P/Ar, ClId, Pt), where P/Ar, ClId and Pt are the predicate signature,
the clause identifier and the program point of the clause at hand. Predicate exec/3
implements the main loop of the interpreter. Given the current state in its second
argument it produces the final state of the computation in the third one. It is
deﬁned by the seven clauses which are applied in they following situations:

1st cl. The current goal is empty, the answer “yes” and there are no pending
choice points. Then, the computation ﬁnishes with answer “yes”. The current
answer is actually used as a flag to indicate whether the previous step in the
computation succeeded or failed (see the last two exec/3 clauses).

2nd cl. As 1st cl. but having at least one choice point. This clause represents the
solution in which the computation ends. The 4th clause takes the other
alternatives.

3rd cl. The previous step failed and there are no pending choice points. Then, the
computation ends with answer “no”.

4th cl. The current goal is empty, the answer “yes” and there is at least one
pending choice point. This is the same situation as in the 2nd clause, however
in this case the alternative of resuming from the last choice point is taken.
The corresponding state S’ is built by means of build_retry_state/5 and
the computation is resumed from S’ by recursively calling exec/3.

5th cl. The previous step failed and there is at least one pending choice point.
Then, the computation is resumed from the last choice point in the same
way as in the previous clause.
The first atom to be solved is user-defined. A call to `solve/4` handles the atom, and the computation proceeds with the next program point of the same clause which was the current one before calling `solve/4`. This way of solving a predicate call makes the interpreter **big-step** (issue (3) above).

The first atom to be solved is a builtin. Then, `run_builtin/3` produces the corresponding answer, and the computation proceeds with the following program point. An interesting observation (also applicable for the previous clause) is that the answer obtained from `run_builtin/3` (or `solve/4`) is now set up as the answer of the next state. This will make the computation go through the 3rd or 5th clauses in the following step, if the obtained answer was "no".

The correspondence between these clauses and the flows in the CFGs is as follows: clauses 1st, 2nd and 4th represent the output edges from every “T” node. Clause 3rd represents the “no” edges to “F” nodes and 5th the “no” edges to non-terminal nodes. Finally clauses 6th and 7th represents the execution of builtins and predicate calls in non-terminal nodes and their corresponding “yes” edges.

Let us now explain how the interpreter handles the above three issues. To handle (1), a stack of choice points is carried along within the state, initialised to contain all initial program points of each clause defining the predicate to be solved, except for the first one. E.g., the initial stack of choice points for `sorted/1` is \([\text{pp(sorted/1,2,1)}, \text{pp(sorted/1,3,1)}]\). How this stack is used to perform the backtracking is already explained in the description of the 4th and 5th `exec/3` clauses above. As regards issue (2), a quite simple way to implement this in Prolog is to produce the necessary fresh variables every time the computation is resumed. This is done inside `build_retry_state/5`. The corresponding unification to link the fresh variables with the original goal variables is made at the end (see last line of `solve/4`). This is the reason why 1) the list of the actual variables used in the current goal needs to be carried along within the state; and 2) the original arguments are carried along as the first argument of `exec/3`, as the original ground arguments provided, have to be used when resuming from a choice point.

Finally, it is worth mentioning that `solve/4` does not return the actual stack of choice points but only the number of them. This means that during a computation the interpreter only considers choice points of the predicate being solved. The question is then, how can the interpreter backtrack to the last choice point, including those induced by other computations of `solve/4`? E.g., how can the interpreter follow edge “13” in the CFG of `sorted/1`? The interpreter performs the backtracking in the following way: 1) The total number of choice points left behind, `TNCPs`, is carried along within the state and finally returned in the last argument of `solve/4`. 2) The number of choice points corresponding to invoked predicates, `ENCPS`, is also carried along. It is updated right after the call to `solve/4` in the 6th clause of `exec/3`. Both numbers are stored in the last argument of the state as `TNCPs/ENCPS`. 3) Execution is resumed from choice points of the current predicate only if `ENCPS = 0`, as it can be seen in the 4th and 5th clauses. Otherwise, the computation just fails and Prolog’s backtracking
mechanism is used to ask the last invoked predicate for more solutions. This indeed means that the non-determinism of the program is still implicit.

4.2 Controlling Partial Evaluation

The specialisation of interpreters has been studied in many different contexts, see e.g. [9,10,19]. Very recently, [10] proposed control strategies to successfully specialise low-level code interpreters w.r.t. non-trivial programs. Here we demonstrate how such guidelines can be, and should be, used in the specialisation of non-trivial Prolog meta-interpreters. They include:

1. **Big-step** interpreter. This solves the problem of handling recursion (see [10]) and enables a compositional specialisation w.r.t. the program procedures (or predicates). Note that an effective treatment of recursion is specially important in Prolog programs where recursion is heavily used.

2. **Optimality** issues. Optimality must ensure that: a) the code to be transformed is traversed exactly once, and b) residual code is emitted once in the transformed program. To achieve optimality, during unfolding, all atoms corresponding with divergence or convergence points in the CFG of the program to be transformed, has to be memoised (see Sect. 2.1). A divergence (convergence) point is a program point from (to) which two or more flows originate (converge).

We already explained that the interpreter in Fig. 3 is big-step. As regards optimality, by looking at the CFGs of Fig. 2, we can observe: 1) all program points are divergence points except those corresponding with unifications in which one argument is a variable, and 2) the first program point of every clause, except for the one of the first clause, is a convergence point. We assume that \( \text{conv}_{\text{points}}(P) \) and \( \text{div}_{\text{points}}(P) \) denote, respectively, the set of convergence points and divergence points of a predicate \( P \). We follow the syntax of [10] for PE annotations. An annotation is of the form \([\text{Precond}] \Rightarrow \text{Ann Pred}\) where \( \text{Precond} \) is an optional precondition defined as a logic formula, \( \text{Ann} \) is the kind of annotation (only memo in this case), and \( \text{Pred} \) is a predicate descriptor, i.e., a predicate function and distinct free variables. Then, to achieve an effective transformation, we specialise the interpreter in Fig. 3 w.r.t. the program to be transformed by using the following annotation for each predicate \( P/\text{Ar} \) in the program:

\[
PP \in \text{div}_{\text{points}}(P/\text{Ar}) \cup \text{conv}_{\text{points}}(P/\text{Ar}) \Rightarrow \text{memo} \\text{exec}(_, \text{st}(PP, _, _, _, _), _)\]

Additionally \text{solve/4} and \text{run_builtin/3} are also annotated to be memoised always to avoid code duplications.

This already describes how the specialisation has to be steered in the local control. As regards the global control, the only predicate which can introduce non-termination is \text{exec/3}. Its first and third arguments contain a fixed structure with variables. The second one might be problematic as it ranges over the set of all computable states at specialisation time. Note that the number of computable states remains finite thanks to the big-step nature of the interpreter. Still, it can
solve(foo/2,[C,D],A,B) :-
  run_builtin_1(E,C),
  exec_1(C,E,F,A,B), F = [D].
  exec_3(no,[_],no,0).
  exec_3(yes,[zero],yes,0).

exec_1(_,yes,[pos],yes,1).
exec_2(A,G,H,I) :-
  run_builtin_2(K,A), exec_3(K,G,H,I).
  run_builtin_1(yes,A) :- A#>0.
  run_builtin_1(no,A) :- \+ A#>0.
  run_builtin_2(yes,A) :- A#=0.
  run_builtin_2(no,A) :- \+ A#=0.

Fig. 4. Transformed code with explicit failure for foo/2

happen that the same program point is reached with different values for the NCPs sub-term of the state. Therefore, if one wants to achieve the optimality criterion above, such argument has to be always generalised in global control.

Example 1. Figure 4 depicts the transformed code we obtain for predicate foo/2. It can be observed that there is a clear correspondence between the transformed code and the CFG in Fig. 2. Thus, predicate solve/4 represents the node “X>0”, exec_1/5 implements its continuation, whose three clauses correspond to the three sub-paths ⟨3⟩, ⟨1,2⟩ and ⟨1,2,4⟩ respectively. Predicate exec_2/4 represents the node “X=0” and exec_3/5 implements its continuation, whose two clauses correspond to the sub-paths ⟨7⟩ and ⟨5,6⟩. Note that edge “8” is not considered in the meta-interpreter (nor in the transformed program) as it is meaningless for TDG. It is worth mentioning that the transformed program captures the way in which variable bindings are undone. For instance in solve(foo/2,[C,D],...), if we keep track of variables C and D, it can be seen that D, which corresponds to variable Z in the original code, is only used for the final unification F=[D], while new fresh variables are used for the unifications with pos and zero. However, variable C, which corresponds to variable X in the original code, is actually used for the checks in run_builtin_1/2 and run_builtin_2/2. This turns out to be fundamental when trying to obtain test data associated to the first-try failing CP ⟨0,3,7⟩. It must be the same variable the one which, at the same time, is not “>0” and not “=0”. Otherwise we cannot obtain a negative number as test data for such CP. Finally, observe that the original Prolog arithmetic builtins have been (automatically) transformed into their clpfd counterparts ⁴.

5 Generating Test Cases by Partial Evaluation

Once the original Prolog program has been transformed into an equivalent Prolog program with explicit failure, we can use the approach of [2] to carry out phase

⁴ We are using the clpfd library of Sicstus Prolog. See [20] for details.
II (see Fig. 1) and generate test data both for successful and failing derivations. As we have explained in Sect. 2.2, the idea is to perform a second PE over the CLP transformed program where the unfolding rule plays the role of the coverage criterion. In [2] an unfolding rule implementing the block-count\( (k) \) coverage criterion was proposed. A set of computation paths satisfies the block-count\( (k) \) criterion if it includes all terminating computation paths which can be built in which the number of times each block is visited does not exceed the given \( k \). The blocks the criterion refers to are the blocks or nodes in the CFGs of the original Prolog program. As the only form of loops in Prolog are recursive calls, the “\( k \)” in the block-count\( (k) \) actually corresponds to the number of recursive calls which are allowed.

Unfortunately, the presence of Prolog’s negation in our transformed programs complicates this phase. The negation will appear in the transformed program for “no” branches originating from nodes corresponding to a (possibly) failing builtin. See for example predicates run\(_\text{builtin}\_1/3\) and run\(_\text{builtin}\_2/3\) in the transformed code of foo/2 in Fig. 4. While Prolog’s negation works well for ground arguments, it gives no information for free variables, as it is required in the evaluation performed during this TDG phase. In particular, in the foo/2 example, given the computation which traverses the calls “\(+ A\#>0\)” and “\(\neg A\#=0\)” (corresponding to the path \(\langle 0,3,7 \rangle\) in the CFG), we need to infer that “\(A<0\)”. In other words, we need somehow to turn the negative information into positive information. This transformation is straightforward for arithmetic builtins: we just have to replace “\(+ e_1\#=e_2\)” by “\(e_1\#\neq e_2\)” and “\(+ e_1\#>e_2\)” by “\(e_1\#<e_2\)”, etc.

Example 2. This transformation allows us to obtain the following set of test-cases for foo/2:

\[
\{ \langle [1],[\text{pos}],\text{yes/first-try} \rangle, \langle [1],[\_],\text{no/after-retry} \rangle, \\
\langle [0],[\text{zero}],\text{yes/first-try} \rangle, \langle [-100],[\_],\text{no/first-retry} \rangle \}
\]

They correspond respectively (reading by rows) to the CPs \(\langle 0,1,2,4,7 \rangle\), \(\langle 0,1,2,4,7 \rangle\cdot\langle 0,3,5,6 \rangle\) and \(\langle 0,3,7 \rangle\). Each test-case is represented as a 3-tuple \((\text{Ins}, \text{Outs}, \text{Ans})\) being \(\text{Ins}\) the list of input arguments, \(\text{Outs}\) the list of output arguments and \(\text{Ans}\) the answer. The answer takes the form \(A/B\) with \(A \in \{\text{yes}, \text{no}\}\) and \(B \in \{\text{first-try}, \text{after-retry}\}\)\(^5\), so that we obtain sufficient information about the kind of CP to which the test-case corresponds (see Sect. 3). As there are no recursive calls in foo/2 such test-cases are obtained using the block-count\( (k) \) criterion for any \(k\) (greater than 0). The domain used for the integer number is \([-100..100]\).

However, it can be the case that negation involves unifications with symbolic data. For example, the transformed code for sorted/1 includes the negations “\(+ L=[]\)” and “\(\neg L=[]\)”. As before, we might write transformations for the negated unifications involving lists, so that at the end it is inferred that “

\(^5\) To simplify the presentation in Sect. 4.1, we decided not include in the interpreter the support to calculate the first-try/after-retry value.
However this would be too an ad-hoc solution as many distinct term structures, different from lists, can appear on negated unifications. A solution for this problem has been recently proposed for Mercury in the same context [5]. It roughly consists in the following: 1) It is assumed that each predicate argument is well-typed. 2) A domain is initialised for each variable, containing the set of possible functors the variable can take. 3) When a negated unification involving an output variable is found (in their terminology a negated decomposition), the corresponding functor is removed from the variable domain. It is crucial at this point the assumption that complex unifications are broken down into simple ones. 4) Finally, a search algorithm is described to generate particular values from the type definition and final domain for the variable. The technique is implemented using CHR and can be directly used in principle for our purposes as well.

On the other hand, advanced declarative languages like TOY [15] make possible the co-existence of different constraint domains. In particular, the co-existence of boolean and numeric constraint domains enables the possibility of using disequalities involving both symbolic data and numbers. This allows for example expressing the negated unifications “\+ L=[ ]” and “\+ L=[[ ]]” as disequality constraints “L=/=[ ]” and “L=/=[ ]”. Additionally, by relying on the boolean constraint solver, the negated arithmetic builtins “\+ A#>0” and “\+ A#=0” can be encoded as “(A#>0) == false” and “(A#=0) == false”. This is in principle a more general solution that we want to explore, although a thorough experimental evaluation needs to be carried out to demonstrate its applicability to our particular context.

Example 3. Now, by using any of the techniques outlined above, we obtain the following set of test-cases for sorted/1, using block-count(2) as the coverage criterion:

```
\[
\{([],[]),\[\],yes/first-try),
\{([0],[]),\[\],yes/first-try),
\{([0,1],[]),\[\],yes/first-try),
\{([0,1,2,0],[]),\[\],no/first-try),
\{([0,0,0],[]),\[\],no/first-try)
\]
```

They correspond respectively (reading by rows) to the CPs “\((0,1)\)””, “\((0,2)\)””, “\((0,2,5,7,10)\) · \((0,2,4)\) · \((11)\)”, “\((0,2,5,7,10)\) · \((0,2,5,7,10)\) · \((0,2,4)\) · \((11)\) · \((11)\)””, “\((0,2,5,7,10)\) · \((0,2,5,7,10)\) · \((0,2,5,7,9)\) · \((12)\) · \((12)\)”, “\((0,2,5,7,10)\) · \((0,2,5,7,9)\) · \((12)\) · \((12)\)””, “\((0,2,5,7,9)\)””. They are indeed all the paths that can be followed with no more than 3 recursive calls. This time the domain has been set up to \{0..100\}.

6 Conclusions and Ongoing work

Very recently, we proposed in [2] a generic approach to TDG by PE which in principle can be used for any imperative language. However, applying this approach to TDG of a declarative language like Prolog introduces some difficulties like the handling of failing derivations and of symbolic data. In this work, we
have sketched solutions to overcome such difficulties. In particular, we have proposed a program transformation, based on PE, to make failure explicit in the Prolog programs. To handle Prolog's negation in the transformed programs, we have outlined existing solutions that make it possible to turn the negative information into positive information. Though our preliminary experiments already suggest that the approach can be very useful to generate test-cases for Prolog, we plan to carry out a thorough practical assessment. This requires to cover additional Prolog features like the module system, builtins like \texttt{cut/0}, \texttt{fail/0}, \texttt{if/3}, etc. and also to compare the results with other TDG systems. We also want to study the integration of other kinds of coverage criteria like data-flow based criteria. Finally, we would like to explore the use of static analyses in the context of TDG. For instance, the information inferred by a failure analysis can be very useful to prune some of the branches that our transformed programs have to consider.

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

References

11. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural
through logic programming. In *SCAM ’06: Proceedings of the Sixth IEEE Interna-
tional Workshop on Source Code Analysis and Manipulation*, pages 184–196. IEEE
394, 1976.
16. G. Luo, G. Bochmann, B. Sarikaya, and M. Boyer. Control-flow based testing
17. M. Méndez-Lojo, J. Navas, and M. Hermenegildo. A Flexible (C)LP-Based Ap-
proach to the Analysis of Object-Oriented Programs. In *17th International Sympo-
sium on Logic-based Program Synthesis and Transformation (LOPSTR’07)*, August
2007.
18. C. Meudec. Atgen: Automatic test data generation using constraint logic program-
19. J.C. Peralta, J. Gallagher, and H. Sağlam. Analysis of imperative programs
through analysis of constraint logic programs. In *Proc. of SAS’98*, volume 1503 of
http://www.sics.se/sicstus/.
22. Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage
Improving Size-Change Analysis in Offline Partial Evaluation

Michael Leuschel\textsuperscript{1}, Salvador Tamarit\textsuperscript{2}, and Germ\’an Vidal\textsuperscript{2}

\textsuperscript{1} Institut f"{u}r Informatik, Universit"{a}t D"{u}sseldorf, D-40225, D"{u}sseldorf, Germany
leuschel@cs.uni-duesseldorf.de
\textsuperscript{2} DSIC, Technical University of Valencia, E-46022, Valencia, Spain
\{stamarit, gvidal\}@dsic.upv.es

Abstract. Some recent approaches for scalable offline partial evaluation of logic programs include a size-change analysis for ensuring both so-called local and global termination. In this work—inspired by experimental evaluation—we introduce several improvements that may increase the accuracy of the analysis and, thus, the quality of the associated specialized programs. We aim to achieve this while maintaining the same complexity and scalability of the recent works.

1 Introduction

Partial evaluation [4] is a well-known technique for program specialization. In this work, we consider the so-called offline approach, which consists of two clearly separated phases: binding-time analysis and proper specialization. Basically, the binding-time analysis should annotate the source code in order to drive the specialization process. Roughly speaking,

\begin{itemize}
  \item every atom is annotated as either \textit{unfold} (the atom can be unfolded) or \textit{memo} (the atom should not be unfolded), and
  \item every predicate’s argument is classified as either \textit{static} (definitely known at specialization time) or \textit{dynamic} (possibly unknown at specialization time).
\end{itemize}

We say that the annotations are \textit{safe} if static arguments are actually ground at specialization time and the termination of the specialization is ensured. Termination issues are usually classified into local and global termination:

\begin{itemize}
  \item local termination ensures that no atom is infinitely unfolded;
  \item global termination guarantees that only finitely many atoms are specialized (i.e., that we do not create infinite specializations of the same predicate).
\end{itemize}

The main component of a binding-time analysis is a termination analysis that allows us to guarantee both local and global termination of the specialization

\textsuperscript{*} This work has been partially supported by the EU (FEDER) and the Spanish MEC/MICINN under grants TIN2005-09207-C03-02, TIN2008-06622-C03-02, and Acci\'{o}n Integrada HA2006-0008
process. In [8], a strong termination analysis—based on the so-called size-change termination principle [5]—for logic programs is introduced. Strong termination means termination w.r.t. all selection rules. Although this is a rather strong condition, it allows us to design much faster binding-time analysis (see [6]).

In this paper, we identify several weaknesses of the original size-change analysis of [8] and present different proposals that improve the accuracy of the specialization process.

2 Size-Change Termination Analysis

In this section, we informally present the basis of the quasi-termination analysis for logic programs of [8].

We say that a query $Q$ is strongly terminating w.r.t. a program $P$ if every SLD derivation for $Q$ with $P$ is finite. We denote by $calls^P_\mathcal{R}(Q_0)$ the set of calls in the computations of a goal $Q_0$ within a logic program $P$ and a computation rule $\mathcal{R}$. The query $Q$ is strongly quasi-terminating if, for every computation rule $\mathcal{R}$, the set $call^P_\mathcal{R}(Q)$ contains finitely many nonvariant atoms. A program $P$ is strongly (quasi-)terminating w.r.t. a set of queries $Q$ if every $Q \in Q$ is strongly (quasi-)terminating w.r.t. $P$. For conciseness, in the remainder of this paper, we write “(quasi-)termination” to refer to “strong (quasi-)termination.”

Size-change analysis is based on constructing graphs that represent the decrease of the arguments of a predicate from one call to another. For this purpose, some ordering on terms is required.

Definition 1 (reduction pair). We say that $(\succsim, \succ)$ is a reduction pair if $\succsim$ is a quasi-order and $\succ$ is a well-founded order where both $\succsim$ and $\succ$ are closed under substitutions and compatible (i.e., $\succsim \circ \succ \subseteq \succ$ and $\succ \circ \succsim \subseteq \succ$ but $\succsim \subseteq \succ$ is not necessary).

In logic programming, however, termination analyses usually rely on the use of norms which measure the size of terms. In [8], reduction orders $(\succsim, \succ)$ induced from symbolic norms $|| \cdot ||$ are used:

Definition 2 (symbolic norm [3, 7]). Given a term $t$,

$$||t|| = \begin{cases} m + \sum_{i=1}^{n} k_i ||t_i|| & \text{if } t = f(t_1, \ldots, t_n), \ n \geq 0 \\ t & \text{if } t \text{ is a variable} \end{cases}$$

where $m$ and $k_1, \ldots, k_n$ are non-negative integer constants depending only on $f/n$. Note that we associate a variable over integers with each logical variable (we use the same name for both since the meaning is clear from the context).

The introduction of variables in the range of the norm provides a simple mechanism to express dependencies between the sizes of terms.

The associated induced orders $(\succsim, \succ)$ are defined as follows: $t_1 \succ t_2$ (respec. $t_1 \succsim t_2$) if $||t_1\sigma|| > ||t_2\sigma||$ (respec. $||t_1\sigma|| \geq ||t_2\sigma||$) for all substitution $\sigma$ that makes $||t_1\sigma||$ and $||t_2\sigma||$ ground (e.g., an integer constant). Two popular instances
of symbolic norms are the symbolic term-size norm $||\cdot||_{ts}$ (which sums the arities of the term symbols) and the symbolic list-length norm $||\cdot||_{ll}$ (which counts the number of elements of a list), e.g.,

$$f(X,Y,a)\succ-ts f(X,a,b) \text{ since } ||f(X,Y,a)||_{ts} = X + Y + 3 > X + 3 = ||f(X,a,b)||_{ts}$$

$$[X|R] \succ/ts [s(X)|R] \text{ since } ||[X|R]||_{ll} = R + 1 \geq R + 1 = ||[s(x)|R]||_{ll}$$

Now, we produce a size-change graph $G$ for every pair $(H, B_i)$ of every clause $H \leftarrow B_1, \ldots, B_n$ of the program, with edges between the arguments of $H$ and $B_i$ when the size of the corresponding terms decrease w.r.t. a given reduction pair $(\succ, \succ)$.  

Example 1. Consider the following simple program:

\begin{align*}
(c_1) & \quad incList([\underline{1}, \underline{\ldots}, \underline{\ldots}], \\
(c_2) & \quad incList([X|R], I, L) \leftarrow iList(X, R, I, L), \\
(c_3) & \quad iList(X, I, [XI|R]) \leftarrow add(I, X, XI), incList(R, I, RI), \\
(c_4) & \quad add(0, Y, Y), \\
(c_5) & \quad add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z).
\end{align*}

Let $(\succ, \succ)$ be the reduction pair induced by the symbolic term-size norm $||\cdot||_{ts}$.  
Here, we have four size-change graphs, depicted in Fig. 1, which are associated to clauses $c_2$ (graph $G_1$), $c_3$ (graphs $G_2$ and $G_3$) and $c_5$ (graph $G_4$).

In order to identify the program loops, we should compute roughly a transitive closure of the size-change graphs by composing them in all possible ways.  
Basically, given two size-change graphs:

\begin{align*}
G = \{\{1_p, \ldots, n_p\}, \{1_q, \ldots, m_q\}, E_1\} & \quad G = \{\{1_p, \ldots, n_p\}, \{1_r, \ldots, l_r\}, E_2\}
\end{align*}

w.r.t. the same reduction pair $(\succ, \succ)$, their concatenation is defined by

$$G \cdot H = \{\{1_p, \ldots, n_p\}, \{1_r, \ldots, l_r\}, E\}$$

Fig. 1. Size-change graphs for incList
where $E$ contains an edge from $i_p$ to $k_r$ iff $E_1$ contains an edge from $i_p$ to some $j_q$ and $E_2$ contains an edge from $j_q$ to $k_r$. Furthermore, if some of the edges are labeled with $\succ$, then so is the edge in $E$; otherwise, it is labeled with $\succsim$.

In particular, according to [5], we only need to consider the idempotent size-change graphs $\mathcal{G}$ with $\mathcal{G} \cdot \mathcal{G} = \mathcal{G}$ for analyzing the termination of the program.

**Example 2.** For the program of Example 1, we compute the following idempotent size-change graphs:

\[
\begin{align*}
\text{incList} & \xrightarrow{\prec}\text{incList} \\
1 & \text{incList} \xrightarrow{\succ}\text{incList} \\
2 & \text{incList} \\
3 & \text{incList} \xrightarrow{\succ}\text{incList} \\
\text{iList} & \xrightarrow{\prec}\text{iList} \\
1 & \text{iList} \xrightarrow{\succ}\text{iList} \\
2 & \text{iList} \\
3 & \text{iList} \xrightarrow{\succ}\text{iList} \\
\text{add} & \xrightarrow{\prec}\text{add} \\
1 & \text{add} \xrightarrow{\succ}\text{add} \\
2 & \text{add} \\
3 & \text{add} \xrightarrow{\succ}\text{add} \\
4 & \text{add}
\end{align*}
\]

that represent how the size of the arguments of the three potentially looping predicates changes from one call to another.

Once the idempotent size-change graphs of a program have been computed, the following results hold:

**Termination:** An atom $A$ is (strongly) terminating if every idempotent size-change graph for $p/n$ contains at least one edge $i_p \xrightarrow{\succ} i_p$ such that, for every computation rule $R$ and atom $p(t_1, \ldots, t_n) \in \text{calls}_R^P(A)$, the argument $t_i$ is instantiated enough w.r.t. the considered symbolic norm. Clearly, the set $\text{calls}_R^P(A)$ is often infinite. Therefore, we usually consider an approximation based on a division that classifies every predicate’s argument as either static or dynamic and check that the $i$-th argument of $p$ is classified as static (rather than checking that $t_i$ is instantiated enough in all possible calls from $A$).

For instance, given a division that classifies the arguments of $\text{add}$ as follows:

\[
\text{add} \mapsto (\text{static, dynamic, dynamic})
\]

and according to the idempotent size-change graphs of Example 2, we have that all calls to $\text{add}$ terminate since there is an edge $1_{\text{add}} \xrightarrow{\succ} 1_{\text{add}}$ in the idempotent size-change graph and the first argument of $\text{add}$ is classified as static.

**Quasi-termination:** An atom $A$ is (strongly) quasi-terminating if it is either terminating or every idempotent size-change graph for $p/n$ contains, for all $i_p$ ($i = 1, \ldots, n$) an edge $j_p \xrightarrow{R} i_p$ for some $j_p$, with $R \in \{\succ, \succsim\}$ (i.e., all arguments are bounded by the value of some argument in a previous call). Furthermore, the considered norms must be bounded (see Definition 3 below).

---

3 A term $t$ is instantiated enough [3, 7] w.r.t. a symbolic norm $||\cdot||$ if $||t||$ is an integer constant.
For instance, according to the idempotent size-change graphs of Example 2, an atom \( add(X, Y, Z) \) is quasi-terminating since there is an input edge to every argument.

In [8], the termination condition is used for ensuring the local termination of partial evaluation, while the quasi-termination condition is used for ensuring its global termination. Basically,

- we reclassify as unfold those atoms which are terminating w.r.t. a given division (and with memo otherwise) and
- we mark with dynamic the argument of an atom if there is no input edge to this argument in some idempotent size-change graph, i.e., if the atom is not quasi-terminating.

Example 3. Given the idempotent size-change graphs of Example 2 and a division that classifies the predicates' arguments as follows:

\[
\begin{array}{c}
\text{incList} & \mapsto & \text{dynamic, static, dynamic} \\
\text{iList} & \mapsto & \text{dynamic, dynamic, static, dynamic} \\
\text{add} & \mapsto & \text{static, dynamic, dynamic}
\end{array}
\]

we have that

- \( \text{incList} \) and \( \text{iList} \) are marked with memo while \( \text{add} \) is marked with unfold, and
- no argument should be re-classified as dynamic.

3 Improving Size-Change Analysis

In this section, we introduce several extensions of the size-change analysis that may improve the accuracy of the specialization process by taking into account some basic properties of partial evaluation.

3.1 Non-Bounded Norms for Global Termination

Let us recall the notion of bounded norm required in [8] for ensuring quasi-termination:

**Definition 3 (bounded norm).** We say that a symbolic norm \( \|\cdot\| \) is bounded if the set \( \{ s \mid \|t\| \geq \|s\| \} \) contains a finite number of nonvariant terms for any term \( t \).

Roughly speaking, a symbolic norm is bounded if, for every term \( t \), there exist only finitely many nonvariant terms whose weights are lesser than or equal to that of \( t \) w.r.t. the symbolic norm \( \|\cdot\| \).

Unfortunately, many symbolic norms are not bounded; e.g., the symbolic list-length norm is not bounded since, given the term \( p([a]) \), we have an infinite
set \{p([a]), p([f(a)]), p([f(f(a))]), \ldots \} \text{ of non-variant terms such that } ||[a]||_H = ||[f(a)]||_H = ||[f(f(a))]||_H = \ldots = 1.

In the context of partial evaluation, however, symbolic norms need not be bounded if the \textit{problematic} parts of the terms are generalized at the global level. For instance, we can safely use the symbolic list-length norm as long as the list elements are replaced by fresh variables in the global level. This idea, already sketched in [6], is formalized by means of the \textit{most general generalization} operator:

\textbf{Definition 4 (mgg).} Let $||\cdot||$ be a symbolic norm. Given a term $t$, we denote by $\text{mgg}^{\parallel\parallel}(t)$ the most general generalization of $t$ such that $||t|| = ||\text{mgg}^{\parallel\parallel}(t)||$. We also let $\text{mgg}^{\parallel\parallel}(p(t_1, \ldots, t_n)) = p(\text{mgg}^{\parallel\parallel}(t_1), \ldots, \text{mgg}^{\parallel\parallel}(t_n))$.

For instance, given the term $t = [s(N), b]$, we have $\text{mgg}^{\parallel\parallel}(t) = [X, Y]$ but $\text{mgg}^{\parallel\parallel}(s(t)) = [s(N), b]$.

Moreover, the quasi-termination result in [8] also requires that all calls encountered during partial evaluation should be \textit{linear} w.r.t. the dynamic variables (i.e., no variable marked as dynamic could appear more than once in a call). However, this is not a real problem in the context of partial evaluation since all dynamic parts of terms are replaced by fresh variables in the global level anyway.

Therefore, one can ensure the global termination of partial evaluation when using arbitrary symbolic norms in the size-change analysis as long as

- dynamic parts of arguments are replaced by fresh variables in the global level (this is already done by current offline partial evaluators) and
- an atom $A$ is replaced by $\text{mgg}^{\parallel\parallel}(A)$ in the global level, where $||\cdot||$ is the symbolic norm used in the size-change analysis.

\section{Maximizing “Unfold” Annotations}

The original approach of [8] does not take into account that different idempotent size-change graphs may represent a single loop. For instance, the idempotent size-change graphs for both \textit{incList} and \textit{iList} actually represent the same program loop. Therefore, it would be safe to annotate only one of these predicates with “\textit{memo}” and the other one with “\textit{unfold}”.

In order to avoid unnecessary \textit{memo} annotations, one can slightly extend the original annotation procedure as follows:

- First, every size-change graph is labeled with a unique identifier (e.g., $G_1$, $G_2$, \ldots, as in Fig. 1).
- Then, the concatenation of graphs is performed as before, but now every concatenation keeps a set with the identifiers of the graphs involved in the concatenation. We note that the set of identifiers is not taken into account during the concatenation process, i.e., two size-change graphs that only differ in the associated set of identifiers are considered equal (therefore, the complexity of the concatenation process, the most expensive part of the analysis, remains the same).

For instance, the labeled idempotent size-change graphs of Example 2 would now be as depicted in Fig. 2.
The computed idempotent size-change graphs can now be grouped into equivalence classes so that two idempotent size-change graphs belong to the same class if they are labeled with the same set of identifiers.

Finally, we should only annotate with “memo” one predicate for every equivalence class of idempotent size-change graphs.

For instance, as mentioned in Example 3, both `incList` and `iList` are marked with `memo` in the original framework. Now, however, only one of them would be marked with `memo` (and the other one with `unfold`).

Clearly, there is a degree of freedom when choosing which is the idempotent size-change graph of a given class that should be marked with `memo`. For this purpose, one can define appropriate heuristics that minimize the number of `memo` annotations by, e.g., assigning a higher priority to those predicates that belong to more than one class.

### 3.3 Right-Propagation of Bindings

An advantage of the size-change analysis of [8] is that it is independent of a particular selection rule. As mentioned in the introduction, this property makes the associated binding-time analysis much faster; unfortunately, it is also less accurate.

In some cases, we can improve this situation by assuming some partial knowledge on the evaluation order. For instance, we could first run a left-termination analysis (like, e.g., the one based on the binary unfoldings [2]) or rely on user’s annotations that identify some atoms as “completely unfoldable” (note that an annotation `unfold` only means that the atom can be unfolded one step; then the annotations of the predicates in the unfolded goal should be followed).

In this case, we can improve the accuracy of the size-change analysis by using an inter-argument size analysis like that calculated from the convex hull of [1]. For instance, given the program

\[
p(X) \leftarrow q(X,Y), p(Y).
\]
\[
q(s(0),0).
\]
\[
q(s(X),Y) \leftarrow q(X,Y).
\]

We thank Maurice Bruynooghe for suggesting this improvement.
the size-change graph associated to $p/1$ originally contains no edge (since we do not know the size relation between $X$ and $Y$). Now, if we assume that $q/2$ is completely unfoldable, then we can use the output of the convex hull of [1] (using a term-size norm):

$$q(A, B) \leftarrow \{ A > B, \ B = 0, \ A \geq 1 \}$$

for propagating some additional constraints to the right of $q$. In this way, one can easily infer that the size-change graph for $p/1$ should contain an edge $1_p \xrightarrow{>} 1_p$.

Let us note that, in principle, the accuracy of the size-change analysis of [8] could not be improved by adding inter-argument size relations to size-change graphs, since inter-argument relations usually require the atoms to be completely unfolded (i.e., they represent relations that hold for success patterns). This assumption is not generally true in the setting of [8] where partial evaluations are possible.

4 Discussion

We have recently undertaken the implementation of a binding-time analysis for the offline partial evaluation of Prolog programs which is based on the size-change analysis of [8]. In this paper, we have introduced several improvements that may allow us to overcome the main weaknesses of [8]. An experimental evaluation will be conducted in order to assess their effectiveness in practice.

References

A Lightweight Combination of Semantics for Non-deterministic Functions *

Francisco Javier López-Fraguas, Juan Rodríguez-Hortalá, and Jaime Sánchez-Hernández

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, jrodrigu@fdi.ucm.es, jaime@sip.ucm.es

Abstract. The use of non-deterministic functions is a distinctive feature of modern functional logic languages. The semantics commonly adopted is call-time choice, a notion that at the operational level is related to the sharing mechanism of lazy evaluation in functional languages. However, there are situations where run-time choice, closer to ordinary rewriting, is more appropriate. In this paper we propose an extension of existing call-time choice based languages, to provide support for run-time choice in localized parts of a program. The extension is remarkably simple at three relevant levels: syntax, formal operational calculi and implementation, which is based on the system Toy.

1 Introduction

Non-strict non-deterministic functions are a distinctive feature of modern functional logic languages (see [5] for a recent survey). It is known that the introduction of non-determinism in a functional setting gives rise to a variety of semantic decisions (see e.g. [12]). For term-rewriting based specifications, Hussmann [7] established a major distinction between call-time choice and run-time choice.

Call-time choice is closely related to call-by-value and, in the case of strict semantics, it is easily implemented by innermost rewriting. In the case of non-strict semantics, things are more complicated, since the call-by-value view of call-time choice must include partial values. Operationally, this needs something similar to the sharing mechanism followed, for efficiency reasons, in (deterministic) functional languages under lazy evaluation. In contrast, run-time choice does not share, corresponds rather to call-by-name, and is realized by ordinary rewriting.

For deterministic programs, run-time and call-time are able to produce the same set of values, but in general the set of values reachable by run-time choice is larger than that of call-time choice.

Non-deterministic functions with non-strict and call-time choice semantics were introduced in the functional logic setting with the CRWL framework [4].

* This work has been partially supported by the Spanish projects MERIT-FORMS-UCM (TIN2005-09207-C03-03), FAST-STAMP (TIN2008-06622-C03-01/TIN) and Promesas-CAM (S-0505/TIC/0407).
A Lightweight Combination of Semantics for Non-deterministic Functions

in which programs are possibly non-confluent and non-terminating constructor-based term rewriting systems (CTRS). Since then, they are common part of daily programming in systems like Curry [6] or Toy [11]. Run-time choice has been rarely [1] considered as a valuable global alternative to call-time choice.

However, there might be parts in a program or individual functions for which run-time choice could be a better option, and therefore it would be convenient to have both possibilities (run-time/call-time) at programmer’s disposal. The following example illustrates the interest of combining both semantics.

Example 1. Modeling grammar rules for string generation can be directly done by CTRS like the following (non-confluent and non-terminating) one, in which we assume that texts (terminals) are represented as strings (lists of characters), that can be concatenated with ++ (defined in a standard way):

\[
\begin{align*}
\text{letter} & \rightarrow \text{"a"} \ldots \\
\text{letter} & \rightarrow \text{"z"} \\
\text{word} & \rightarrow \text{""} \\
\text{word} & \rightarrow \text{letter}++\text{word}
\end{align*}
\]

Disregarding syntax, this CTRS is a valid program in functional logic systems like Curry or Toy. The program acts as a non-deterministic generator of the texts in the language defined by the grammar. Each individual reduction leads to a string in the language.

The generation of palindromes (of even length, for simplicity) could be done by the rewrite rules:

\[
\begin{align*}
\text{palindrome} & \rightarrow \text{palAux(word)} \\
\text{palAux(X)} & \rightarrow X++\text{reverse(X)}
\end{align*}
\]

where reverse is defined in any standard way. It is important to remark that the definition of palindrome/palAux works fine only if call-time choice is adopted for non-determinism, meaning operationally that in the (partial) reduction

\[
\text{palindrome} \rightarrow \text{palAux(word)} \rightarrow \text{word}++\text{reverse(word)}
\]

the two occurrences of word created by the rule of palAux must be shared. If run-time choice (i.e., ordinary rewriting) were used, the two occurrences of word could follow independent ways, and therefore palindrome could be reduced, for instance, to "oops", which is not a palindrome. Two useful operators to structure grammar specifications are the alternative ‘|’ and Kleene’s ‘∗’ for repetitions:

\[
\begin{align*}
X | Y & \rightarrow X \quad X | Y \rightarrow Y \\
\text{star X} & \rightarrow \text{""} \\
\text{star X} & \rightarrow X++\text{star(X)}
\end{align*}
\]

With them letter and word could be redefined as follows:

\[
\begin{align*}
\text{letter} & \rightarrow \text{"a"} | \text{"b"} | \ldots | \text{"z"} \\
\text{word} & \rightarrow \text{star(letter)}
\end{align*}
\]

The annoying fact is that this does not work! At least not under call-time choice, which implies that this is an uncertain definition of star in systems like Curry or Toy. The problem with call-time choice here is that all the occurrences of letter created by star will be shared and therefore word will only generate words like aaa, nnnn, . . . , made with repetitions of the same letter. To overcome this problem, we would like that in the definition of word, the application of
the star operation to the string generator letter could follow a run-time choice regime, so that each of the two occurrences of letter created in the rewriting steps

\[ \text{word} \rightarrow \text{star(letter)} \rightarrow \text{letter} ++ \text{star(letter)} \]

could evolve independently. In our proposed extension this would be expressed by writing the definition of word as follows:

\[ \text{word} \rightarrow \text{star(rt(letter))} \]

where rt is a special unary function symbol indicating that its argument (letter in this case) is not going to be shared in the evaluation of the surrounding application (\text{star(rt(letter))}) in this case.

We remark that in this example neither call-time nor run-time choice are a good single option as semantics for the whole program. The definition of palindrome requires call-time choice, while the use of star in word requires run-time choice. To the best of our knowledge, no existing implementation for functional logic programming offers the possibility of combining in the same program both kind of semantics. This paper addresses that problem at a practical level, aiming at a solution that can be easily realized by modifying existing Prolog-based functional logic systems. Although our main interest is easiness of implementation, we provide also formal calculi attempting to reflect at an abstract level the operational behavior of the extended language. These calculi could be the technical basis for a thorough investigation of the formal properties of our proposal, a matter that is out of the scope of this paper.

2 A tiny functional logic language with run-time choice annotations

We shortly present here a functional logic language with run-time choice annotations. To keep the presentation simple, we consider only a first order untyped core with the usual first order syntax of term rewriting systems. However, the implementation described in Sect. 5 extends the existing system Toy, which is a HO typed system using curried notation.

We consider a signature \( \Sigma \) made of constructor symbols \( c, d, \ldots \in CS \), function symbols \( f, g, \ldots \in FS \), the special unary symbol \( rt \), and a set of variables \( X, Y, \ldots \in V \). We sometimes write \( c \in CS^n \ (f \in FS^n) \) to denote a constructor (function) symbol of arity \( n \). Constructor terms (or \( c \)-terms) \( t, s, \ldots \in CTerm \) follow the syntax: \( t ::= X | c(t_1, \ldots, t_n) \), and expressions (with run-time choice annotations) \( e, \ldots \in RtExpr \) follow the syntax: \( e ::= X | c(e_1, \ldots, e_n) | f(e_1, \ldots, e_n) | rt(e) \). An intermediate set between \( CTerm \) and \( RtExpr \) is the set \( RtCTerm \) of annotated \( c \)-terms \( RtCTerm \ni t ::= X | c(t_1, \ldots, t_n) | rt(e) \), where \( t_1, \ldots, t_n \) are also from \( RtCTerm \) and \( e \) is any expression.

A program is a set of function defining rules, each of the form

\[ f(t_1, \ldots, t_n) \rightarrow e \]
where \((t_1, \ldots, t_n)\) is a linear tuple of \(c\)-terms from \(CTerm\), and \(e\) is any expression from \(RtExpr\). We remark that annotated \(c\)-terms play no special role in the syntax of programs, but play an important role in the parameter passing mechanism, which informally can be explained as follows: to apply a program rule \(f(t_1, \ldots, t_n) \rightarrow e\) to a function application \(f(e_1, \ldots, e_n)\), a matching substitution \(\theta\) such that \(f(t_1, \ldots, t_n) \theta \equiv f(e_1, \ldots, e_n)\) must exist, and then \(f(e_1, \ldots, e_n)\) reduces to \(r\theta\), but following the informal criterion about sharing: the copies of subexpressions \(e\) of \(f(e_1, \ldots, e_n)\) created in \(r\theta\) are not shared –i.e. follow run-time choice– if \(e\) is under a \(rt\) annotation, and shared –i.e. follow call-time choice– otherwise. These ideas are formalized in the next section in the form of two alternative operational calculi.

3 Formal operational calculi

In this section we will try to design some calculi able to express an extension of the standard call-time choice semantics for FLP [4], to support the primitive \(rt\) for run-time choice evaluation. Our approach to formalize this extension is based in two main ideas:

- The new calculus will be a modification of the simple rewrite calculus presented in [9]. As we will have to express run-time evaluation for parts of the computation, we will need to have partially evaluated expressions at our disposal. A calculus in the line of those used in [4] would not be a suitable tool, as it returns only partial values for the expressions, but no intermediate states of the computation.
- Instead of giving a semantics for annotations \(rt(e)\) directly, we will think about it as a syntactic sugar for the annotation of the function symbols that appear in \(e\) with a \(rt\) superscript, indicating that those function symbols will be treated as a constructor symbol as far sharing and parameter passing is concerned. Therefore, an expression containing only variables, constructor symbols and function symbols annotated with \(rt\) could be copied freely, thus getting a run-time behaviour for it, as a function argument. We write \(FS^{rt}\) for the set of function symbols with superscript \(rt\), \(FS^t\) for \(FS \cup FS^{rt}\) and \(f^t\) for function symbols in \(FS^t\), i.e., for possibly superscripted function symbols.

The desugaring of expressions to eliminate the \(rt\) primitive transforming it into \(rt\) annotations is performed according to the following definition:

**Definition 1 (Desugaring of the \(rt\) primitive).**

\[
\begin{align*}
\text{desugar}(rt(X)) &= X & \text{if } X \in V \\
\text{desugar}(rt(c(e_1, \ldots, e_n))) &= c(\text{desugar}(rt(e_1)), \ldots, \text{desugar}(rt(e_n))) & \text{if } c \in CS \\
\text{desugar}(rt(f(e_1, \ldots, e_n))) &= f^t(\text{desugar}(rt(e_1)), \ldots, \text{desugar}(rt(e_n))) & \text{if } f \in FS \\
\text{desugar}(rt(rt(e))) &= \text{desugar}(rt(e))
\end{align*}
\]

According to this syntactic desugaring for \(rt(e)\), the syntax of annotated \(c\)-terms and expressions can be reformulated as follows:
To express parameter passing in function applications with rt-annotated arguments we will need to consider rt-c-substitutions, defined by: $\theta \in \text{RtCSubst}$ iff $X\theta \in \text{RtCTerm}, \forall X \in \mathcal{V}$.

Now we will define calculi to work with annotated expressions. In [9] two rewrite notions for call-time choice were defined, each of them being interesting for different applications. Here we will modify both of them to get two (hopefully) equivalent characterizations of a semantics for annotated run-time choice under a call-time choice environment.

(Fapp) $f'(\overline{p})\theta \rightarrow_{\text{r}} r\theta$, if $(f(\overline{p}) \rightarrow r) \in \mathcal{P}$, $\theta \in \text{RtCSubst}$

(LetIn) $h(\ldots,e,\ldots) \rightarrow_{\text{r}} \text{let } X = e \text{ in } h(\ldots,X,\ldots)$, if $h \in \Sigma$, $e \equiv f(\overline{v})$ with $f \in \text{FS}$ or $e \equiv \text{let } Y = e' \text{ in } e''$, and $X$ is a fresh variable.

(Bind) $\text{let } X = t \text{ in } e \rightarrow_{\text{r}} e[X/t]$, if $t \in \text{RtCTerm}$

(Elim) $\text{let } X = e_1 \text{ in } e_2 \rightarrow_{\text{r}} e_2$, if $X \not\in \text{FV}(e_2)$

(Flat) $\text{let } X = (\text{let } Y = e_1 \text{ in } e_2) \text{ in } e_3 \rightarrow_{\text{r}} \text{let } Y = e_1 \text{ in } (\text{let } X = e_2 \text{ in } e_3)$ if $Y \not\in \text{FV}(e_3)$

(Context) $\mathcal{C}[e] \rightarrow_{\text{r}} \mathcal{C}[e']$, if $\mathcal{C} \neq [], e \rightarrow_{\text{r}} e'$ using any of the previous rules, and in case $e \rightarrow_{\text{r}} e'$ is a (Fapp) step using $(f(\overline{p}) \rightarrow r)\theta \in [\mathcal{P}]$ then $\nu\text{Ran}(\theta|_{\text{var}(\overline{p})}) \cap \text{BV}(\mathcal{C}) = \emptyset$.

Fig. 2. Rules of let-rewriting extended with rt annotations
Note how in the rule (LetIn), in the case a function application is extracted
to a let, it is needed that \( f \) is not marked with \( \text{rt} \), which tell us that it is not
allowed to duplicate it, and therefore it may be needed to put it in a let in order
to progress with the evaluation (for example if it appears in an argument of
another function application whose reduction is needed).

Example 1. Given the program

\[
\begin{align*}
\text{coin} & \rightarrow 0 \\
\text{coin} & \rightarrow 1
\end{align*}
\]
\[
\begin{align*}
f(X) & \rightarrow g(X, \text{coin}) \\
g(X, Y) & \rightarrow (X, X, Y, Y)
\end{align*}
\]

we want to evaluate the expression \( \text{rt}(f(\text{coin})) \), which is desugared as \( f^{\text{rt}}(\text{coin}^{\text{rt}}) \).

With the calculus of Fig. 1 we can do:

\[
\begin{align*}
f^{\text{rt}}(\text{coin}^{\text{rt}}) & \rightarrow g(\text{coin}^{\text{rt}}, \text{coin}) \rightarrow g(\text{coin}^{\text{rt}}, 0) \rightarrow (\text{coin}^{\text{rt}}, \text{coin}^{\text{rt}}, 0, 0) \\
& \rightarrow (0, \text{coin}^{\text{rt}}, 0, 0) \rightarrow (0, 1, 0, 0)
\end{align*}
\]

Note how in the first step the expression \( f^{\text{rt}}(\text{coin}^{\text{rt}}) \) can be evaluated as every
function symbol present in \( \text{coin}^{\text{rt}} \) is annotated with \( \text{rt} \). On the other hand we
cannot apply (OR) to \( g(\text{coin}^{\text{rt}}, \text{coin}) \), as one of its arguments contains a function
symbol that it is not annotated for run-time, and thus the value \((0, 1, 0, 1)\) is not
reachable from \( f^{\text{rt}}(\text{coin}^{\text{rt}}) \). This is even more evident in the version of this
evaluation got with the calculus of Fig. 2:

\[
\begin{align*}
f^{\text{rt}}(\text{coin}^{\text{rt}}) & \rightarrow g(\text{coin}^{\text{rt}}, \text{coin}) \rightarrow g(\text{coin}^{\text{rt}}, 0) \rightarrow (\text{coin}^{\text{rt}}, \text{coin}^{\text{rt}}, 0, 0) \\
& \rightarrow (0, \text{coin}^{\text{rt}}, 0, 0) \rightarrow (0, 1, 0, 0)
\end{align*}
\]

When we reach the expression \( \text{let } X = \text{coin} \text{ in } (\text{coin}^{\text{rt}}, \text{coin}^{\text{rt}}, X, X) \) it is clear
that the first two components of the tuple may evolve in different ways while
the values of the last two components will be shared.

4 A variant of run-time annotations

In the present section we will show another primitive to express run-time choice
that we will build on top of the previous primitive \( \text{rt} \), through a simple program
transformation. We will call that primitive \( \text{rRt} \), and define its behaviour by the
following inference rule that should be added to the CRWL logic [4]:

\[
\frac{e \rightarrow_{\mathcal{P}} e' \quad \text{or} \quad t \sqsubseteq |e'|}{\mathcal{P}' \vdash_{\text{CRWL}} \text{rRt}(e) \rightarrow t} \text{ (rRt)}
\]

where \( \mathcal{P}' \) is the program resulting of adding to \( \mathcal{P} \) the new rule \( \text{rRt}(e) \rightarrow e \),
and \( e \rightarrow_{\mathcal{P}} e' \) indicates that \( e' \) can be reached from \( e \) by zero or more steps of
ordinary rewriting [2] using the program \( \mathcal{P}' \). The approximation ordering \( t \sqsubseteq t' \)
between partial values expresses that \( t \) is less defined than \( t' \) (see [4] for details).
The rule \((rRt)\) itself is already suggesting a possible implementation for \(rRt\). This implementation will be based on the fact that, for any program in which every function symbol that appears in a right hand side of a program rule is \(rt\)-annotated, the evaluation of an expression that has each of its function symbols \(rt\)-annotated too returns the same results as it was evaluated under run-time choice but discarding the annotations. This ideas are formalized in the following definition:

**Definition 2.** Given a CRWL-program \(\mathcal{P}\):

- We build the signature of a new program \(\overline{\mathcal{P}}\) adding to it any constructor symbol in the signature of \(\mathcal{P}\), and for any function symbol \(f\) in the signature of \(\mathcal{P}\) considering a fresh function symbol \(\overline{f}\) which we add to the signature of \(\overline{\mathcal{P}}\).

- We define the transformation of expressions \(rRt\) as:
  
  \[
  \begin{align*}
  rRtT(X) &= X & \text{if } X \in \mathcal{V} \\
  rRtT(c(e_1, \ldots, e_n)) &= c(rRtT(e_1), \ldots, rRtT(e_n)) & \text{if } c \in CS \\
  rRtT(f(e_1, \ldots, e_n)) &= f^{rt}(rRtT(e_1), \ldots, rRtT(e_n)) & \text{if } f \in FS \\
  rRtT(rRt(e)) &= rRt(e)
  \end{align*}
  \]

- For any \((f(p_1, \ldots, p_n) \rightarrow r) \in \mathcal{P}\) we add the rule \((f(p_1, \ldots, p_n) \rightarrow rRtT(r))\) to \(\overline{\mathcal{P}}\).

Finally, any expression \(rRt(e)\) to be evaluated under \(\mathcal{P}\) is desugared into \(rRtT(e)\) and evaluated under \(\mathcal{P} \cup \overline{\mathcal{P}}\).

**Example 2.** Starting with the program of Example 1 we get the program

\[
\{ coin \rightarrow 0, coin \rightarrow 1, f(X) \rightarrow g(X, coin), g(X, Y) \rightarrow (X, X, Y, Y) \}
\]

\[
\cup
\{ \overline{\text{coin}} \rightarrow 0, \overline{\text{coin}} \rightarrow 1, f(X) \rightarrow \overline{g^{rt}}(X, \overline{\text{coin}}), \overline{g}(X, Y) \rightarrow (X, X, Y, Y) \}
\]

under which we can do:

\[
rRt(f(\overline{\text{coin}})) \equiv \overline{f^{rt}}(\overline{\text{coin}^{rt}}) \rightarrow \overline{\text{g}^{rt}}(\overline{\text{coin}^{rt}}, \overline{\text{coin}^{rt}}) \\
\rightarrow (\overline{\text{coin}^{rt}}, \overline{\text{coin}^{rt}}, \overline{\text{coin}^{rt}}, \overline{\text{coin}^{rt}}) \rightarrow^* (0, 1, 0, 1)
\]

### 5 Implementation issues

In order to study the practicability of the proposal we have implemented it as an extension of the functional logic system \(Toy\) ([3]). This system, as well as other modern systems like \(Curry\) ([6]), operates under call-time choice. We introduce the new syntactic construct \(rt\ e\) into the syntax of \(Toy\) to instruct the system to evaluate the expression \(e\) under a run-time choice regime. The system will use run-time choice for evaluating the expressions annotated with \(rt\), and call-time choice as usual for the rest of computations, i.e., we have within the same language both regimes of evaluation.
The extension is well supported by the system and requires only some lightweight modifications. In fact, the traditional problem is how to achieve sharing in a non-deterministic language like this, and our goal now is to inhibit this sharing mechanism at the points required by the programmer with \( rt \).

\( Toy \) is implemented in Prolog and uses Prolog as target code (see [8, 3] for details). Sharing is implemented by means of suspensions, that are Prolog terms of the form:

\[
susp(FunctionName, Arguments, Result, Evaluated)
\]

The \( FunctionName \) and its \( Arguments \) represent the expression \( e \) to be evaluated, while \( Result \) is the resulting value (if evaluated, variable in other case) and \( Evaluated \) is a flag that indicates if the expression has been evaluated (flag \( on \)) or not (flag variable). Every function call is translated into a suspension in order to share its value when the expression is passed as argument and copied. As an example of the use of this representation consider the following program:

\begin{verbatim}
coin = 0
coin = 1
double X = X + X
test1 = double coin
test2 = rt (double coin)
\end{verbatim}

Consider the evaluation of \( test1 \). As all the function calls are translated into suspended forms, in particular \( coin \) will have the form \( susp(coin, [], R, E) \). The evaluation of \( double \) does not demand the evaluation of its argument \( coin \), so it will produce

\[
susp(coin, [], R, E) + susp(coin, [], R, E)
\]

Later, when one of the calls to \( coin \) is evaluated, for example to 0, the other one automatically gets the same value:

\[
susp(coin, [], 0, on) + susp(coin, [], 0, on)
\]

The result of the addition is 0, that is a value obtained for \( test1 \). If we evaluate \( coin \) to 1 we have

\[
susp(coin, [], 1, on) + susp(coin, [], 1, on)
\]

and then result 2, that is the other value obtained for \( test1 \). With this sharing mechanism we can not obtain the value 1 for \( double coin \) as it would require to evaluate both calls to \( coin \) to two different values.

For the function function \( test2 \) we would want to obtain the values 0 and 2 as before, but also the value 1 (evaluating separately both calls to \( coin \)). In this case \( rt \) will deactivate the sharing mechanism. This can be easily achieved by translating the call \( coin \) into the suspended form \( susp(coin, [], R, rt) \). The flag \( rt \) will indicate to the system that the value of this expression must not be shared (and neither kept in the variable \( R \)). For \( test2 \) we evaluate
The first suspension can be reduced to 0 (without annotating the result in \( R \)), and the second one to 1, obtaining 1 for \( \text{test2} \) as expected.

The extension implemented in \( \text{Toy} \) provides this behaviour with \( \text{test1} \) and \( \text{test2} \). In fact, for \( \text{test2} \) it obtains 0, 2 and 1 twice (evaluating the first \( \text{coin} \) to 0 and the second to 1 and viceversa). As another example, consider the problem of generating numbers as combinations of the digits 0, 1 and 2. Using \( \text{take, repeat} \) and the alternative operator ‘|’ (introduced in Sec. 1) we could define:

\[
\text{number } N = \text{take } N (\text{repeat} (0 \mid 1 \mid 2))
\]

but then the expression \( \text{number } 3 \) will produce only the answers \([0,0,0] \), \([1,1,1] \) and \([2,2,2] \), because the expression \( 0 \mid 1 \mid 2 \) is evaluated only once and then its value is shared when evaluating \( \text{repeat} \). For achieving the expected behaviour we have to instruct the system for choosing the digits under run-time choice (to avoid sharing):

\[
\text{number } N = \text{take } N (\text{repeat} (\text{rt} (0 \mid 1 \mid 2)))
\]

Now we obtain the 27 possible combinations that include \([1,1,2] \) or \([3,1,2] \) as instance. The example of palindromes of Sect. 1 also works as expected.

The prototype and some examples can be found at https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems.

6 Conclusions

We have proposed a simple way of combining in the same program run-time choice and call-time choice, two semantics commonly adopted for non-determinism in rewriting-based declarative languages, but that cannot coexist within the same program in current systems.

The approach presented here starts from a call-time choice ambient (as given by most popular functional logic systems like \( \text{Curry} \) [6] or \( \text{Toy} \) [11]) and adds to it the possibility of annotating the evaluation of (sub)-expressions as following a run-time choice regime. We have proposed two variants of this idea, the first being more ‘local’ in the effect of an annotation \( \text{rt}(\epsilon) \), while the second is more global. In both cases we have proposed a formal definition of the intended semantics.

For the first variant we have given formal operational descriptions, by adapting to the new setting two one-step reduction relations proposed in [9] as a simple notion of rewriting adequate for call-time choice. As for implementation, this variant has been achieved by modifying of the system \( \text{Toy} \). Essentially, we have needed to change the management of suspensions, that are the technical key to implement sharing for call-time choice. The resulting prototype can be found at https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems.

For the second variant we give a logical semantics that extends, to cope with \( \text{rt} \) annotations, the proof calculus of the CRWL framework [4].
how to transform annotations of this variant into the first one. This mapping can be used to implement the second variant.

Recently, we have tried a different alternative to the combination of call-time and run-time choice [10], following a way complementary to the one in this paper: there we start from ordinary rewriting and enhance it with local bindings through a let construct to express sharing and call-time choice. The resulting framework seems to be more amenable to formal treatments, as shown by the good number of technical results obtained in [10]. On the other hand, the approach here seems to be more easily implementable, at least if one wants to reuse existing call-time-choice based implementations.

References

Abstract. Fuzzy reasoning is a very productive research field that during the last years has provided a number of theoretical approaches and practical implementation prototypes. Nevertheless, the classical implementations, like Fril, are not adapted to the latest formal approaches, like multi-adjoint logic semantics. Some promising implementations, like Fuzzy Prolog, are so general that the regular user/programmer does not feel comfortable because either representation of fuzzy concepts is complex or the results difficult to interpret.

In this paper we present a modern framework, Rfuzzy, that is modelling multi-adjoint logic. It provides some extensions as default values (to represent missing information, even partial default values) and typed variables. Rfuzzy represents the truth value of predicates through facts, rules and functions. Rfuzzy answers queries with direct results (instead of constraints) and it is easy to use for any person that wants to represent a problem using fuzzy reasoning in a simple way (by using the classical representation with real numbers).

Key words: Fuzzy reasoning, Implementation tool, Fuzzy Logic, Multi-adjoint logic, Logic Programming Application

1 Introduction

One of the reasoning models that is more useful to represent real situations is fuzzy reasoning. Indeed, world information is not represented in a crisp way. Its representation is imperfect, fuzzy, etc., so that the management of uncertainty is very important in knowledge representation. There are multiple frameworks for incorporating uncertainty in logic programming:

- fuzzy set theory [5, 38, 32],
- probability theory [8, 16, 20, 25, 26],
- multi-valued logic [7, 11, 12, 15, 17, 29],
- possibilistic logic [6, 36, 37]

** This work is partially supported by the project DESAFIOS - TIN 2006-15660-C02-02 from the Spanish Ministry of Education and Science and project PROMESAS - S-0505/TIC/0407 from the Madrid Regional Government.
Despite of the multitude of theoretical approaches to this issue, few of them resulted in actual practically usable tools. Since Logic Programming is traditionally used in Knowledge Representation and Reasoning, we argue it is perfectly well-suited to implement a fuzzy reasoning tool as ours.

1.1 Fuzzy Approaches in Logic Programming

The result of introducing Fuzzy Logic into Logic Programming has been the development of several fuzzy systems over Prolog. These systems replace the inference mechanism of Prolog, SLD-resolution, with a fuzzy variant that is able to handle partial truth. Most of these systems implement the fuzzy resolution introduced by Lee in [18], as the Prolog-Elf system [10], the FRIL Prolog system [2] and the F-Prolog language [19]. However, there is no common method for fuzzifying Prolog, as has been noted in [28].

Some of these Fuzzy Prolog systems only consider the predicates’ fuzziness whereas other systems consider fuzzy facts or fuzzy rules. There is no agreement about which fuzzy logic should be used. Most of them use min-max logic (for modelling the conjunction and disjunction operations) but other systems just use Lukasiewicz logic [13].

Furthermore, logic programming is considered a useful tool for implementing methods for reasoning with uncertainty in [38].

There is also an extension of constraint logic programming [3], which can model logics based on semiring structures. This framework can model min-max fuzzy logic, which is the only logic with semiring structure.

Another theoretical model for fuzzy logic programming without negation has been proposed by Vojtas in [35], which deals with many-valued implications.

1.2 Fuzzy Prolog

One of the most promising fuzzy tools for Prolog was the “Fuzzy Prolog” system [33, 9]. The most important advantages against the other approaches are:

1. A truth value will be a finite union of sub-intervals on [0, 1]. An interval is a particular case of union of one element, and a unique truth value is a particular case of having an interval with only one element.
2. A truth value is propagated through the rules by means of an aggregation operator. The definition of this aggregation operator is general and it subsumes conjunctive operators (triangular norms [14] like min, prod, etc.), disjunctive operators [31] (triangular co-norms, like max, sum, etc.), average operators (averages as arithmetic average, quasi-linear average, etc) and hybrid operators (combinations of the above operators [27]).
3. Crisp and fuzzy reasoning are consistently combined [24].

Fuzzy Prolog adds fuzziness to a Prolog compiler using CLP(R) instead of implementing a new fuzzy resolution, as other former fuzzy Prologs do. It represents intervals as constraints over real numbers and aggregation operators as operations with these constraints, so it uses Prolog’s built-in inference mechanism to handle the concept of partial truth.
1.3 Motivation and RFuzzy Approach

Over the last few years several papers have been published by Medina et al. ([22, 23, 21]) about multi-adjoint programming, which describe a theoretical model, but no means of serious implementations apart from promising prototypes [1].

Indeed, that was the reason for developing Fuzzy Prolog [9]. Fuzzy Prolog is a very expressive tool which allows the user\(^1\) to program almost everything, but we have to pay for this expressiveness. The cost is a complex syntax difficult to understand.

The motivation for developing Rfuzzy is mainly focused on reducing this complex syntax. This reduction is based on three ideas:

1. Use real numbers instead of intervals between real numbers to represent truth values. Fuzzy Prolog answers to user queries are intervals like \(\text{it\_will\_rain (tonight, [0, 1])}\). This is a bit difficult to understand by normal users. Truth value of this example is between 0 and 1, and this means that program can not conclude anything about the predicate truth value.

2. Whenever it is possible, do not answer user queries using constraints. A Fuzzy Prolog answer to an user query can be a constraint, like \(\text{it\_will\_rain (tomorrow, [X, Y]), X > 0, X < 1, Y > 0, Y < 1}\). The meaning of this example is exactly the same as the meaning of the previous one, but it is slightly more difficult to understand it.

3. Truth value variables do not need to be coded. Taking care of variables to manage the predicates truth value introduces errors and makes the code illegible, without giving us any advantage.

Rfuzzy uses real numbers to represent truth values and its replies are never constraints. Besides, it is able to distinguish between crisp and fuzzy predicates and it manages the introduction of truth value variables, so the user does not need to take care of them. Truth variables are always introduced at the end of the predicate arguments list, so it can be seen as some syntactic sugar. We explain this in subsection 2.6.

From the point of view of expressiveness, we can remark that RFuzzy offers to the user the ability to define types, general and conditioned default values and truth value representations by means of facts, functions or rules. Besides, it implements multi-adjoint logic with representation of the concept of credibility of the rules, so it is one of the first tools that are actually modelling multi-adjoint logic\(^2\).

\(^1\) We refer as ‘user’ to the programmer that wants to represent a fuzzy problem in a programming framework to make queries and obtain results.

\(^2\) A complete formalization of the semantics of RFuzzy with a description of a least model semantics, a least fixpoint semantics, an operational semantics and the prove of their equivalence can be downloaded at [http://babel.ls.fi.upm.es/software/rfuzzy/](http://babel.ls.fi.upm.es/software/rfuzzy/). This paper has been submitted and is pending of acceptance in an international conference.
2 Rfuzzy syntax

In this section we are going to describe RFuzzy’s syntax. RFuzzy defines the syntax of a new subset of Prolog predicates to work with truth values and to assign credibility to rules. The extensions that we have added to provide fuzziness of predicates are: type information, truth values (for facts, functions and rules) and default truth values.

RFuzzy shares with Fuzzy Prolog most of its nice expressive characteristics: Prolog-like syntax (based on using facts and clauses), use of any aggregation operator, flexibility of query syntax, constructivity of the answers, etc. Nevertheless, RFuzzy is simpler than Fuzzy Prolog for the user because the truth values are simple real numbers instead of the general structures of Fuzzy Prolog.

2.1 Type definition

Prolog does not have types. Prolog code are formulas and at execution time it looks for all of them to be true. To do that, it generates a Herbrand Universe and tries to substitute every variable with a Herbrand term. As we do not want programs to look for an answer infinitely, we offer the user a facility to restrict the set of possible solutions. This extension is named “types” and its syntax is shown in (1).

\[
\text{:- set\_prop pred/ar =\=> type\_pred,1/1 [\{, type\_pred,2/1 \}]}. \tag{1}
\]

where \text{set\_prop} is a reserved word, \text{pred} is the name of the typed predicate, \text{ar} is its arity and \text{type\_pred,\{n\}} is the predicate used to assure that the value given to the argument in the position \text{n} of a call to \text{pred/ar} is correctly typed. Predicate \text{type\_pred,\{n\}} must have arity 1. The example below shows that the two arguments of the predicate \text{has\_lower\_price/2} have to be of type \text{car/1} and which individuals belong to that type.

\[
\text{:- set\_prop has\_lower\_price/2 =\=> car/1, car/1.}
\]

\text{car(alfa\_romeo\_gt).}
\text{car(aston\_martin\_bulldog).}
\text{car(lamborghini\_urraco).}

2.2 Fact truth value

Fuzzy facts are facts to which we assign a truth value. To code them in programs we offer a special syntax, so Prolog can distinguish between normal facts and fuzzy facts. This syntax is shown in (2).

\[
\text{pred(args) value truth\_val}. \tag{2}
\]

Arguments ( \text{args} ) should be ground and the truth value ( \text{truth\_val} ) must be a real number between 0 and 1. The example below defines that the car \text{alfa\_romeo\_gt} is an \text{expensive\_car} with a truth value 0.6.

\text{expensive\_car(alfa\_romeo\_gt) value 0.6}.
2.3 Functional truth value

Fact truth value definition (see subsection 2.2) is worth for a finite (and relative small) number of individuals. As we may want to define a big amount of individuals, we need more than this.

Fuzzy truth values are usually represented using continuous functions. Fig. 1 shows an example in which the truth value function assigns the truth value of being *teenager* from the person’s age value.

![Fig. 1. Teenager credibility.](image)

A function can be defined in several ways, but the easiest one is via a sequence of ordered pairs whose first element is the fact and the second element is the truth value assigned to that fact.

Functions used to define the truth value of some group of facts are usually continuous and linear over intervals. To define those functions there is no necessity to write down the value assigned to each element in their domains. A better way to define them is by means of their inflexion points, so function values for the elements between the inflexion points are determined by means of interpolation.

RFuzzy provides the syntax for defining functions by stretches. This syntax is shown in (3). External brackets represent the Prolog list symbols and internal brackets represent cardinality in the formula notation. $argI, ..., argN$ should be ground terms (numbers) and $truth\_valI, ..., truth\_valN$ should be border truth values. The truth value of the rest of the elements (apart from the border elements) is obtained by interpolation.

\[
pred :\# ([(arg1, truth\_val1), (arg2, truth\_val2), ..., (arg3, truth\_val3)]) .
\] (3)

:- set_prop teenager/1 => people_age/1.
:- default(teenager/1, 0).

2.4 Rule truth value

A tool which only allows the user to define truth values through functions and facts leaks on allowing him to combine those truth values for representing more complex situations. A rule is the perfect tool to combine the truth values of facts, functions, and other rules.
Rules allow the user to combine truth values in the correct way (by means of aggregation operators, like maximum or product). Besides this combination truth value for the body of the rule, the rule can be given an overall credibility truth value.

Credibility is used to express how much we trust the rule we write. Suppose a small weather example in which we have the rule *it will rain if it is cloudy and it is hot*. As it might rain but it might not, we can assign the rule a credibility of 0.7. As expected, the truth value obtained from the body is combined with the credibility value of the rule to obtain a final truth value.

*Rfuzzy* offers the user a concrete syntax to define combinations of truth values by means of aggregation operations, and assign to that rules a credibility. This syntax extension is defined in (4). Indeed, the user can choose two aggregation operators: *op2* for combining the truth values of the subgoals of the rule’s body and *op1* for combining the previous result with the credibility of the rule.

\[
pred(\text{arg1}^*, \text{arg2}^*) \; [\; \text{cred} \; (\text{op1}, \text{value1})]^{0,1} : \sim \text{op2} \; \text{pred1}(\text{arg1}^*, \text{arg2}^*) \; [\; \text{cred2}(\text{arg1}^*, \text{arg2}^*)] . (4)
\]

The following examples show its usage. The second one uses the operator *prod* for aggregating truth values of the subgoals of the body and the operator *min* to aggregate the result with the credibility of the rule, 0.8. As can be deduced from syntax and examples, *cred* and *∼* are reserved words.

*tempting_restaurant*(R) : *∼ prod* low_distance(R), cheap(R), traditional(R).

*good_player*(J) *cred* (*min*, 0.8) : *∼ prod* swift(J), tall(J), experience(J).

### 2.5 General and Conditioned Default Truth Values

Unfortunately, information provided by the user is not complete in general. So there are many cases in which we have no information about the truth value of an individual or a set of them. Nevertheless, it is interesting not to stop a complex query evaluation just because we have no information about one or more subgoals if we can use a reasonable approximation. The solution to this problem is using default truth values for these cases. The *Rfuzzy* extension to define a default truth value for a predicate when applied to individuals for which the user has not defined an explicit truth value is named *general default truth value*.

*Conditioned default truth value* is used when the default truth value only applies to a subset of the function’s domain. This subset is defined by a membership predicate which is true only when an individual belongs to the subset.

---

3 Aggregation operators available are: *min* for minimum, *max* for maximum, *prod* for the product, *luka* for the Łukasiewicz operator, *dprod* for the inverse product, *dluka* for the inverse Łukasiewicz operator and *complement*. 

---
The membership predicate (\textit{membership\_predicate/ar}) and the predicate to which it is applied (\textit{pred/ar}) need to have the same arity (\textit{ar}). If not, an error message will be shown at compilation time.

The syntax for defining a general default truth value is shown in (5), and the syntax for assigning a conditioned default truth value is shown in (6). \textit{pred/ar} is in both cases the predicate to which we are defining default values. As expected, when defining the three cases (explicit, conditioned and default truth value) only one will be given back when doing a query. The precedence when looking for the truth value goes from the most concrete to the least one.

\begin{verbatim}
:- default(pred/ar, truth_value).
\end{verbatim} \hspace{1cm} (5)

\begin{verbatim}
:- default(pred/ar, truth_value) => membership\_predicate/ar.
\end{verbatim} \hspace{1cm} (6)

The example below shows how to assign a default truth value of 0.5 to all cars that do not have an explicit truth value nor have a default conditioned truth value. Besides, it shows how to assign a conditioned default truth value to all cars belonging to a small subset and not having an explicit truth value. This subset is determined by the membership predicate \textit{expensive\_type/1}, and default truth value for its elements is 0.9. So \textit{lamborghini\_urraco} is an \textit{expensive\_car} with truth value 0.9 but \textit{vw\_caddy} is an \textit{expensive\_car} with truth value 0.5. Both values are default approximations because we have no direct declaration (as for \textit{alfa\_romeo\_gt} that is an \textit{expensive\_car} with a truth value 0.6 as we show above).

\begin{verbatim}
:- set\_prop expensive\_car/1 => car/1.
:- default(expensive\_car/1, 0.9) => expensive\_type/1.
:- default(expensive\_car/1, 0.5).

expensive\_type(lamborghini\_urraco).
expensive\_type(aston\_martin\_bulldog).
\end{verbatim}

2.6 Doing queries with truth values

Indeed the program has to be run. When compiling, \textit{Rfuzzy} adds a new argument to the arguments list of each fuzzy predicate. This argument serves for querying about the predicate truth value. It can be seen as syntactic sugar, as truth value is not part of the predicate arguments, but metadata information.

Truth value argument is added to the predicates in a uniform way: it is always a new argument at the end of the arguments list of the predicate. In the previous example we wrote \textit{expensive\_car/1}, so to query the system we have to give the predicate two arguments instead of only one where the second one will represent the query’s truth value. This can be seen in the first example of subsection 2.7.

2.7 Constructive Answers

A fuzzy tool should be able to provide constructive answers for queries. The regular (easy) questions are asking for the truth value of an element. For example, how expensive is an \textit{alfa\_romeo\_gt}?
But the really interesting queries are the ones that ask for values that satisfy constraints over the truth value. For example, which cars are very expensive?. RFuzzy provides this constructive functionality.

?- expensive_car(X,V), V > 0.8.
V = 0.9, X = aston_martin_bulldog ? ;
V = 0.9, X = lamborghini_urraco ? ;
no

The RFuzzy package implements a meta-translation of the RFuzzy syntax to ISO Prolog, via CLP(R), this is the reason for its constructivity.

3 Applications

RFuzzy is mainly suitable for expert systems applications. As mentioned before, its main advantages in comparison to Fuzzy Prolog are its simpler syntax, the use of real numbers instead of intervals between them and the implicit handling of truth values. Besides, it presents facts’ truth values (explicit, default or conditioned default truth value), functions’ truth values and rules (with or without credibility) which simplifies the user development process a lot.

Although a medical expert system development were the best example of using RFuzzy, due to lack of space we prefer to show here one in which we decide which is the best restaurant for going out.

:- module(restaurant,_,[rfuzzy, clpq]).

:- prop restaurant/1.

:- set_prop tempting_restaurant/1 => restaurant/1.
:- default(tempting_restaurant/1, 0.1).
tempting_restaurant(R) :- prod low_distance(R), cheap(R),
   traditional(R).

restaurant(kenzo).
restaurant(burger_king).
restaurant(pizza_jardin).
restaurant(subway).
restaurant(derroscas).
restaurant(il_tempietto).
restaurant(kono_pizza).
restaurant(paellador).
restaurant(tapasbar).
crisp_distance(kenzo, 50).
crisp_distance(burguer_king, 100).
crisp_distance(pizza_jardin, 70).
crisp_distance(subway, 85).
crisp_distance(derroscas, 120).
crisp_distance(il_tempietto, 150).
crisp_distance(kono_pizza, 65).
crisp_distance(paellador, 55).
crisp_distance(tapasbar, 40).

low_distance(R, TV) :- crisp_distance(R, D),
                    low_distance_aux(D, TV).

:- set_prop low_distance_aux/1 => distance/1.
:- default(low_distance_aux/1, 0).
low_distance_aux :# 
                  [[ (0, 1), (50, 0.9), (100, 0.8), (200, 0.6),
                    (300, 0.5), (500, 0.4), (1000, 0.1),
                    (2000, 0) ]].

very_low_distance(X) :- crisp_distance(X, D), D < 100.

:- set_prop cheap/1 => restaurant/1.
:- default(cheap/1, 0.2) => very_low_distance/1.
:- default(cheap/1, 0.5).

cheap(kenzo) value 0.2.
cheap(el_rincon) value 1.
cheap(el_reventaero) value 1.

:- set_prop traditional/1 => restaurant/1.
:- default(traditional/1, 0.8) => very_low_distance/1.
:- default(traditional/1, 1).

traditional(kenzo) value 0.5.
traditional(el_reventaero) value 0.87.

distance(0).
distance(X) :- distance(Y), number(Y),
               X = Y + 1,
               ( X < 5000 ; ( X >= 5000, !, fail) ).

In the example we can see that we know the distance to all the restaurants in a crisp way. This crisp value is translated by means of low_distance and low_distance_aux into a fuzzy one which is used into tempting_restaurant to determine its truth value. This allows us to ask which is the truth value of each
tempting restaurant, which restaurant is a tempting restaurant with a truth value of, for example, 0.7 or list all tempting restaurants with their truth values.

The Rfuzzy module with installation instructions and examples can be downloaded from http://babel.ls.fi.upm.es/software/rfuzzy/.

4 Implementation details

RFuzzy is a logic programming language that is able to model all the extensions that are described in section 2. It is implemented as a Ciao Prolog [30] package because Ciao Prolog offers the possibility of dealing with a higher order compilation through the implementation of Ciao packages. Those packages serve as input for the “Ciao System Preprocessor” (CiaoPP) [4], a tool able to perform source-to-source transformations.

The reason beyond the implementation of Rfuzzy as a Ciao Prolog package is that the resultant code has to deal with two kinds of queries:

- queries in which the user asks for the truth value of an individual, and
- queries in which the user asks for an individual with a concrete truth value.

As can be seen in the following example, this is not an easy task.

?- A is 1, B is 2, C is A + B.
yes
?- C is 3, C is A + B.
{ERROR: illegal arithmetic expression}
{ERROR: illegal arithmetic expression}
no
?- 

Formula $C$ is $A + B$ only works if variables $A$ and $B$ are bound. Almost all predicates that are problematic with non-bound variables have inside comparisons and/or assignments. This aims us to translate Rfuzzy programs into CLP($\mathbb{R}$) programs. CLP($\mathbb{R}$) is a Ciao Prolog Package which translates real number operations into constraints applied to the variables involved in those operations.

Taking advantage of Rfuzzy and CLP($\mathbb{R}$) transformations, our tool compiles Rfuzzy programs into ISO Prolog programs, so the interpreter is able to work with them as it normally does. As a result, the global compilation process has two preprocessor steps: (1) the Rfuzzy program is translated into CLP($\mathbb{R}$) constraints by means of the Rfuzzy package and (2) those constraints are translated into ISO Prolog by using the CLP($\mathbb{R}$) package. Fig. 2 shows the whole process.

In the following example the predicate tempting_restaurant is translated from Rfuzzy syntax into ISO Prolog syntax. In the first step, the Rfuzzy package inserts truth value variables, the inject metapredicate call (one of its arguments is the aggregation operator to be used, prod) and inserts Rfuzzy comparisons to
take care at execution time that the rule’s truth value is always between zero and one. In the second step, CLP(\(\mathcal{R}\)) converts comparisons into constraints (via predicate calls).

\[
\text{\% Rfuzzy program}
\]
\[
\text{tempting\_restaurant}(R) :\sim \text{prod low\_distance}(R), \text{cheap}(R), \text{traditional}(R).
\]

\[
\text{\% CLP(\(\mathcal{R}\)) program}
\]
\[
\text{rfuzzy\_rule\_tempting\_restaurant}(R,\_1) :-
\text{low\_distance}(R,\_2),
\text{cheap}(R,\_3),
\text{traditional}(R,\_4),
inject([\_2,\_3,\_4],\text{prod},\_1),
\_1 .\geq . 0,
\_1 .\leq . 1.
\]

\[
\text{\% ISO Prolog program}
\]
\[
\text{rfuzzy\_rule\_tempting\_restaurant}(R,\_1) :-
\text{low\_distance}(R,\_2),
\text{cheap}(R,\_3),
\text{traditional}(R,\_4),
inject([\_2,\_3,\_4],\text{prod},\_1),
\text{solve\_generic\_1}(\text{le},0,\_1,-1),
\text{solve\_generic\_1}(\text{le},-1,\_1,1).
\]

Internally, Rfuzzy package unifies and translates all the information given by the user to each predicate (Types, default values with and without condition, truth values defined in facts and rules with and without credibility) into a single predicate body. A simplified version of the skeleton used for that predicate is shown below.

\emph{Rfuzzy package simplified skeleton}

\[
\text{Main} :- \text{Types}, (\text{Normal} ; \text{Default})
\]

\[
\text{Normal} :- (\text{Fact} ;
(\text{\textbackslash +}(\text{Fact\_Aux}), \text{Function}) ;
(\text{\textbackslash +}(\text{Fact\_Aux}), \text{\textbackslash +}(\text{Function\_Aux}), \text{Rule})
)
\]
The skeleton has three different parts: the one which takes care of allowing only queries or answers with the expected individuals, the one which looks for a concrete truth value (it can be defined by means of a fact, a function or a rule) and the one which looks for a default truth value (conditioned or not). Predicates ending in \textit{aux} do not take care on the truth value argument.

The first part is obtained from the type definitions (see 2.1), translating all types into a predicate which is called at first (Types) so we assure we only work with the expected individuals.

The second part looks for a concrete value whose arguments have to unify with the parameters the user has given. Precedence when looking for it is:

1. A fact (see subsection 2.2)
2. A function (see subsection 2.3)
3. A rule (see subsection 2.4)

The third part is only called when the second one (searching for a truth value) fails, and looks for a conditioned or default truth value.

5 Conclusions and Current Work

\textit{Rfuzzy} offers to the users a new framework to represent fuzzy problems over real numbers. \textit{Fuzzy Prolog} [34, 33, 9] is an existing framework for dealing with fuzzy problems representation. Main \textit{Rfuzzy} advantages over \textit{Fuzzy Prolog} are a simpler syntax and the elimination of answers with constraints, and \textit{Rfuzzy} is one of the first tools modelling multi-adjoint logic, as explained in subsection 1.3.

\textit{Rfuzzy} syntax is simpler that \textit{Fuzzy Prolog} syntax. Its fuzzy values are simple real numbers instead of intervals between real numbers, and it hides the management of truth value variables. As normal fuzzy problems do not use intervals to represent fuzziness and do not need to code an uncommon behaviour of fuzzy variables, this syntax reduction is an advantage. Programs written in \textit{Rfuzzy} syntax are more legible and more easy to understand than \textit{Fuzzy Prolog} programs.

\textit{Fuzzy Prolog} answers to user queries are difficult to understand due to the existence of constraints. As normal replies to final users are ground terms, the programmer has to code by hand how to reach them. To eliminate those constraints and answer queries with ground terms the programmer tries to substitute variables with ground terms until one makes true all of them. \textit{Rfuzzy} offers a powerful tool to deal with this task: \textit{Type definition}. \textit{Type definition} (see subsection 2.1) allows the user to define which terms are suitable for being substituted.
into a variable, so she does not have to code this behaviour again. Besides, the elimination of answers with constraints provides more human readable answers and more easy to test programs (because answers we test do not have constraints, just ground terms).

There is also an extension to introduce default truth values. As world information is sometimes incomplete, Rfuzzy offers to the user the possibility to define default truth values and default conditioned truth values (see subsection 2.5). This allows us to make inference with default truth values when we do not know anything about the truth of some fact.

Extensions added to Prolog by Rfuzzy are: Types, default truth values (conditioned or not), assignment of truth values to individuals by means of facts, functions or rules, and assignment of credibility to the rules.

Besides, the possibility to provide constructive answers to the queries increase its usage, as can be seen in subsection 2.7.

There are countless applications and research lines which can benefit from the advantages of using the fuzzy representations offered by Rfuzzy. Some examples are: Search Engines, Knowledge Extraction (from databases, ontologies, etc.), Semantic Web, Business Rules, and Coding Rules (where the violation of one rule can be given a truth value).

Current work on Rfuzzy tries to apply constructive negation to the engine. RFuzzy needs to define types in a constructive way (by means of predicates that are able to generate all their individuals by backtracking) so we cannot use constraints. Future research will be done in this line for widening the definition of types.

References


Constraint solving for high-level WCET analysis*

Adrian Prantl¹, Jens Knoop¹, Markus Schordan², and Markus Triska¹

¹ Institut für Computersprachen,
Vienna University of Technology, Vienna, Austria
email: {adrian,knoop,triska}@complang.tuwien.ac.at
² University of Applied Sciences Technikum Wien, Vienna, Austria
email: schordan@technikum-wien.at

Abstract The safety of our day-to-day life depends crucially on the correct functioning of embedded software systems which control the functioning of more and more technical devices. Many of these software systems are time-critical. Hence, computations performed need not only to be correct, but must also be issued in a timely fashion. Worst case execution time (WCET) analysis is concerned with computing tight upper bounds for the execution time of a system in order to provide formal guarantees for the proper timing behaviour of a system. Central for this is to compute safe and tight bounds for loops and recursion depths. In this paper, we highlight the TuBound approach to this challenge at whose heart is a constraint logic based approach for loop analysis.

1 Motivation

Embedded software systems are virtually ubiquitous today to control the functioning of technical devices we routinely use and rely on in our day-to-day life. Many of these systems are safety-critical. Think of applications in the avionics and automotive field such as fly-by-wire or its foreseeable companion technology drive-by-wire, where there is no longer any mechanical linkage between the pilot stick and the steering gear of an aircraft or the steering wheel and the tires of a car. Applications like these demonstrate that it is not only the comfort and convenience of our day-to-day life but also its safety, which depends crucially on the correct functioning of these systems. Many of these systems are also time-critical. This means that calculations performed by such a system need not only to be correct but also have to be issued in a timely fashion. Worst case execution time (WCET) analysis is concerned with providing formal guarantees for the proper timing behaviour of a system by computing tight upper bounds for the execution time of a system.

* This work has been partially supported by the Austrian Science Fund (Fonds zur Förderung der wissenschaftlichen Forschung) under contract No P18925-N13, “Compiler Support for Timing Analysis”, http://costa.tuwien.ac.at/ and the 7th EU R&D Framework Programme under contract No 215068, “Integrating European Timing Analysis Technology” (ALL-TIMES).
State-of-the-art WCET analysis tools rely on supporting analyses to provide them with information on the execution behaviour of the program such as loop bounds or maximum recursion depths. Typically, both steps are performed on the binary code of the program. While this is in fact mandatory for the WCET analysis in the narrow sense in order to get an upper bound of the execution time of the code that is actually executed, it is not for the supporting analyses. This is important to note because it is usually more difficult to implement and perform the supporting analyses on the binary code of a program, since type or control-flow information, which is readily available in the source code, is not in the binary code; it also imposes a particular hardship on the programmer, when demanded to provide information manually as is occasionally necessary e.g. because of the undecidability of the involved analysis problems.

The TuBound approach [16], which we pursue in the CoSTA project [3], is to improve on this by lifting the supporting analyses for WCET analysis to the source code level of a program. The information computed on this level and annotated in the code is then conjointly transformed throughout the compilation and optimization of the program to the binary code level to make it accessible to the WCET analysis component of our TuBound tool. Currently, all optimizations are performed on the source code level, too. The transformed and optimized code is then fed into a specific WCET-aware [9] variant of the Gnu-C compiler [12], which is tailored for preserving the validity of code annotations it is provided with in the compiled code. The binary code it generates is finally passed to a retargetable WCET analysis component, which computes the desired upper bound of the execution time of the program in the worst case. Currently, this is the WCET analyzer CalcWCET [2].

The outcome of the recent participation of the TuBound tool in the 2008 WCET Tool Challenge, which has been held as a part of the 8th International Workshop on Worst-Case Execution Time Analysis (WCET’08) shows the practicality and the power of the TuBound approach and tool [11]. Analyzing the results of the WCET Tool Challenge shows that the key to the success of the TuBound tool is the generality and precision of the constraint logic based approach for loop analysis to fully automatically compute safe and tight loop bounds and flow constraints for most of the benchmark programs subject to this challenge [5]. The usage of logic and constraint-logic programming was fundamental to achieve this and to obtain a stable prototype in short time and with moderate effort.

In this paper, which is an elaborated version of a recent oral presentation [15] at the Workshop on Resource Analysis (ResAn’08), we focus on the essence of our constraint-logic based loop analysis and its implementation in Prolog. Below, we present a summary of key components of the TuBound tool including our constraint-logic solver, before presenting our constraint-logic based loop analysis in detail.
2 TuBound Architecture: Key Components

2.1 SATIrE, ROSE and PAG

Most important to implementing the source-to-source analysis and optimization approach of TuBound are the usage and integration of the SATIrE, LLNL-ROSE, and PAG systems of TU Vienna, Lawrence Livermore National Laboratory (LLNL) and AbsInt GmbH, respectively [17,4,13]. SATIrE is the Static Analysis Integration Engine [18], which seamlessly connects the C++ source-to-source compiler infrastructure LLNL-ROSE [19] with the Program Analysis Generator PAG [13]. In TuBound, this enables us to create data-flow analyses which operate on the abstract syntax tree (AST) of C++ programs. Moreover, SATIrE supports the import and export of an external term representation of the AST using Prolog syntax. This term representation is automatically annotated with any results from preceding analysis steps and contains all necessary information to correctly unparse the program, including line and column information for each expression. This term representation is the key to specifying analyses and program transformations in Prolog, which we make strong use of in TuBound. We are thereby benefiting from many advantages over using C++ for the specification, including pattern matching and access to tools and methods offered by the world of logic programming. Most outstandingly, in TuBound this has been used to implement a flow constraint analysis by means of our generalized finite domain constraint solver (cf. Section 2.2), a loop bound analysis written in SWI-Prolog (cf. Appendix), and an interprocedural interval analysis specified with PAG [13] (cf. Appendix).

2.2 CLP(FD)

Generally speaking, constraint logic programming over finite domains, denoted as CLP(FD), is a declarative formalism for modeling and solving combinatorial problems over integers. A constraint satisfaction problem (CSP) consists of:

- a set $X$ of variables, $X = \{x_1, \ldots, x_n\}$
- for each variable $x_i$, a set $D(x_i)$ of values that $x_i$ can assume, which is called the domain of $x_i$
- a set of constraints, which are simply relations among variables in $X$, and which can further restrict their domains.

In TuBound, we use the solver library clpfd [20]. This is a generalised CLP(FD) solver that we developed, and which was recently included in the SWI-Prolog distribution [21]. Two new features which are implemented in this solver make it especially well-suited for the loop analysis presented in Section 3.2: First, the solver can reason over arbitrarily large integers, and can thus also be used to analyse a large number of nested loops that can range over large bounds. Second, constraint propagation in our solver always terminates. While this weakens propagation when domains are still unbounded, this property guarantees that the loop analysis itself always terminates. It is this property, which makes the clpfd solver particularly useful for the TuBound approach described next.
3 Loop constraint analysis in TuBound

3.1 Preliminaries

TuBound derives loop bounds and constraints for iteration-variable based loops. This type of loop is very common in embedded applications. The debie program [11], a real-world space-craft control system used in the WCET tool challenge, for example, contains 88% iteration-variable based loops. We call a loop \( L \) iteration-variable based if

- it is preceded by an initialization statement \([i := a]\) \( l_1\),
- it contains at least one exit condition \([i \ rel \ b]\) \( l_2\),
- it contains exactly one monotone iteration step statement \([i := i + c]\) \( l_3\),

where \( i \) is an integer variable that is not a field member with a scope larger than the loop body \((L \subseteq \text{scope}(i))\). The labels \( l_1, l_2, l_3 \) are used to reference these statements.

We assume that the program has been run through an interval analysis\(^3\) that generates variable-interval pairs for each variable \( v \) at each sequence point, denoted \( v_{(min, max)} \). In order to classify the results, we call the value of a variable at a given location loop invariant, if the value does not change over all paths through \( L \). We call the value of a variable constant, if \( v_{\text{min}} = v_{\text{max}} \). This implies that every constant variable is also loop invariant. Having this information, we can verify that the loop satisfies the following safety conditions:

\( C_1 \). There is no statement \( s \in L \setminus \{l_1, l_2, l_3\} \) where \( i \) appears on the left-hand side.

\( C_2 \). There is no statement \( s \in \text{scope}(i) \) in that the address of \( i \) is taken.

\( C_3 \). The loop must not be an infinite loop, i.e. the analyzable intervals of \( a \) and \( b \) must either be disjoint or overlapping in at most one value. Further, the direction of the loop must be unambiguous \((\text{sgn}(c_{\text{min}}) = \text{sgn}(c_{\text{max}}))\) and the property \( \text{sgn}(b - a) = \text{sgn}(c_{\text{min}}) \) must hold.

\( C_4 \). The exit condition’s relational operator \( \text{rel} \) must induce a partial order \( \{\leq, \geq\} \). For equivalence operators \( \{=, \neq\} \) it is also necessary to prove that the loop terminates at all, before they can safely be replaced with \( \{\leq, \geq\} \). For the case that \( a, b, c \) are loop invariant and constant,

\[ b - a \ (\text{mod} \ c) \equiv 0 \]

is a sufficient condition for termination. The operators < and > can be transformed by adding \( \pm 1 \) to \( b \).

\(^3\) This analysis is implemented in TuBound, too, but it is beyond the core focus of this paper. Details can be found in the Appendix.
3.2 The constraint analysis for nested loops

TuBound contains an implementation of a loop-bound algorithm that works for nested loops. If the iteration space described by the iteration variables is rectangular or cuboid-shaped, the resulting bounds will even be optimal. Often, however, the iteration variables of nested loops depend on each other, forming e.g. a triangular iteration space. Loop bounds would then be an overestimation of the iteration space, describing the enclosing rectangle. It is thus desirable to formulate more general flow constraints in addition to loop bounds. The flow constraints we are generating describe the execution counts of the loop bodies in relation to the scope containing the outermost loop. Our constraint analysis works by transforming the whole loop nest into finite domain logic constraints.

Each loop in the loop nest must be iteration-variable based. In contrast to the traditional loop bound analysis, a few additional restrictions are imposed on the loop: The step size must be loop invariant. When the loop has a stride greater than 1 (|c| > 1), a should be constant. Otherwise the results produced by the analysis will be an overestimation which is bounded by a factor of a_{max} - a_{min}. Furthermore, the exit test expression must either test for ≤ or ≥; a < b can be transformed into the equivalent a ≤ b − 1.

The algorithm works recursively, beginning with the outermost loop. First, a new logic variable I is created that is associated with the iteration variable. Then, the init, test and step statements are translated into constraints, as sketched in Table 1. The remaining arithmetic expressions can then recursively be translated into corresponding constraints. After the constraints are posted, the constraint solver is used to report the number n of possible combinations of all iteration variables that were encountered so far. Since explicit enumeration of all solutions can be infeasible, we added a new labeling option upto in to our constraint solver, which can be used to count the number of possible instantiations if all remaining constraints are trivial. With this method, the running time and memory consumption of the solver is no longer depending on the size of the iteration space.

The resulting n is then an upper bound for the number of times the current (=innermost regarded) loop is executed relative to the scope containing the outermost loop. If the constraint analysis is applied to a single loop only, the resulting constraint degenerates into a loop bound.

By using this approach, we can leverage a great deal of features from our constraint solver for the loop analysis:

<table>
<thead>
<tr>
<th>Direction</th>
<th>Init</th>
<th>Test</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>up</td>
<td>I #&gt;=</td>
<td>InitExpr</td>
<td>I #&lt;= TestExpr</td>
</tr>
<tr>
<td>down</td>
<td>I #=&lt;</td>
<td>InitExpr</td>
<td>I #&gt;= TestExpr (I-InitExpr) mod StepExpr #= 0</td>
</tr>
</tbody>
</table>

Table 1. Deriving the constraints
The order in which the constraints are posted does not influence the behaviour of the solver.

The termination of the constraint solver is guaranteed.

The strategy of the solver can be customized through labeling options to improve its efficiency (cf. Section 3.3).

Through the implicit enumeration of the iteration space the results are generally more precise than those of the traditional loop bound analysis.

Since much of the complexity is offloaded into the constraint solver, the implementation is very concise and easy to maintain.

### 3.3 Example

We illustrate the general principle using the following loop nest, for which we want to determine the number of times the inner loop is executed:

```plaintext
for (i = 0; i < 10; ++i)
    for (j = i; j > 0; j -= 2)
```

By translating the loop nest accordingly, we get the following constraint program:

```plaintext
I #>= 0, I #< 10, I mod 1 #= 0,
J #=< I, J #> 0, (J-I) mod 2 #= 0,
findall((I,J),labeling([], [I,J]), IS),
length(IS, IterationCount).
```

By solving the constraint system, we explicitly enumerate the iteration space $IS$ described by $(i, j)$:

```
[ (1, 1),
  (2, 2),
  (3, 1), (3, 3),
  (4, 2), (4, 4),
  (5, 1), (5, 3), (5, 5),
  (6, 2), (6, 4), (6, 6),
  (7, 1), (7, 3), (7, 5), (7, 7),
  (8, 2), (8, 4), (8, 6), (8, 8),
  (9, 1), (9, 3), (9, 5), (9, 7), (9, 9) ]
```

The number of pairs in the iteration space is then an upper bound for the innermost loop body. In our case, exactly 25 times. For larger bounds, explicit enumeration of all solutions is infeasible. We therefore added a new labeling option to our constraint solver, which can be used to count the number of possible instantiations if all remaining constraints are trivial. Thus we can reduce or avoid explicit enumeration in many cases. For example:

```plaintext
I #>= 0, I #=< 10000,
J #>= 0, J #=< 500,
labeling([upto_in(IterationCount)], [I,J]).
```

yields $IterationCount = 5010501$. 

---

82 Adrian Prantl, Jens Knoop, Markus Schordan, and Markus Triska
4 Experimental results

To evaluate the constraint analysis, we compare its power with that of a pure loop bound analysis. This second analysis was implemented at an earlier stage and is also part of TuBound (cf. Appendix). The loop bound analysis works by solving linear equations that are derived from the loop parameters.

We are using the standardized WCET benchmark suite from Mälardalen University [1], consisting of over 30 prototypical embedded programs and the debie benchmark from the WCET Tool Challenge 2008 [11]. Since the prerequisites for applying the constraint analysis are slightly more restrictive than for the loop bound analysis, we expect the constrained loops to be a subset of the bounded loops. This is confirmed by the results, which are shown in Table 2 for the Mälardalen University benchmarks and in Table 3 for the debie benchmark.

The first column lists the name of the benchmark, the second column the number of loops that are contained in that benchmark. Column three gives the percentage of loops that could be analyzed with the traditional loop bound algorithm discussed in the Appendix. The running times of the algorithm in seconds is shown in the next column. The last two columns contain the percentage of loops that could be analyzed with the constraint analysis and the corresponding running times. From examining the table we can see that the constraint analysis can analyze almost 90% of the loops that are analyzable with the traditional approach and more than 70% of all loops contained in the benchmarks. Moreover, the constraint analysis inherently outperforms the traditional approach on nested loops with non-rectangular iteration space, due to the higher expressivity of flow constraints.

When comparing the runtime performance of the two approaches, it is apparent that the loop bound analysis mostly depends on the depth of the `init`, `test` and `step` expressions, whereas the worst-case running time of the constraint analysis is correlated with the size of the iteration space, if the solver has to fallback to enumeration. For typical embedded code that we target with TuBound, this has little significance, since analyzing even the outliers is a matter of seconds. The average execution time of both analyses together is well below one second on current hardware. Methods like in [6] could be used to complement this with a more theoretical performance statement.

5 Conclusion and perspectives

We have presented our design and implementation for a generalized loop constraint analysis, which plays an important role as a supporting analysis in our WCET analysis tool, TuBound. Our results demonstrate that this analysis can determine tighter flow constraints for nested loops than our traditional loop bound analysis.

---

4 Measurements were made on a 3 GHz Xeon, running SWI-Prolog 5.6.59 under Linux.
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Loops</th>
<th>Loopbounds</th>
<th>Runtime</th>
<th>Constraints</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>adpcm</td>
<td>18</td>
<td>83.3%</td>
<td>0.02s</td>
<td>83.3%</td>
<td>0.02s</td>
</tr>
<tr>
<td>bs</td>
<td>1</td>
<td>0%</td>
<td>&lt; 0.01s</td>
<td>0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>bsort100</td>
<td>3</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>cnt</td>
<td>4</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>compress</td>
<td>7</td>
<td>28.5%</td>
<td>0.01s</td>
<td>14.2%</td>
<td>0.08s</td>
</tr>
<tr>
<td>cover</td>
<td>3</td>
<td>100.0%</td>
<td>0.01s</td>
<td>100.0%</td>
<td>0.01s</td>
</tr>
<tr>
<td>crc</td>
<td>3</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>des</td>
<td>10</td>
<td>90.0%</td>
<td>0.09s</td>
<td>90.0%</td>
<td>0.09s</td>
</tr>
<tr>
<td>duff</td>
<td>2</td>
<td>50.0%</td>
<td>&lt; 0.01s</td>
<td>50.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>edn</td>
<td>12</td>
<td>100.0%</td>
<td>0.02s</td>
<td>91.6%</td>
<td>0.05s</td>
</tr>
<tr>
<td>expint</td>
<td>3</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>fdct</td>
<td>2</td>
<td>100.0%</td>
<td>0.01s</td>
<td>100.0%</td>
<td>0.01s</td>
</tr>
<tr>
<td>ft1</td>
<td>11</td>
<td>54.5%</td>
<td>&lt; 0.01s</td>
<td>18.1%</td>
<td>0.41s</td>
</tr>
<tr>
<td>fibcall</td>
<td>1</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>fir</td>
<td>2</td>
<td>50.0%</td>
<td>0.03s</td>
<td>50.0%</td>
<td>0.03s</td>
</tr>
<tr>
<td>insertsort</td>
<td>2</td>
<td>50.0%</td>
<td>&lt; 0.01s</td>
<td>0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>janne_complex</td>
<td>2</td>
<td>50.0%</td>
<td>&lt; 0.01s</td>
<td>0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>jfdctint</td>
<td>3</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>lcdnum</td>
<td>1</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>lms</td>
<td>10</td>
<td>60.0%</td>
<td>0.02s</td>
<td>60.0%</td>
<td>0.01s</td>
</tr>
<tr>
<td>ludcmp</td>
<td>11</td>
<td>100.0%</td>
<td>0.01s</td>
<td>81.8%</td>
<td>0.01s</td>
</tr>
<tr>
<td>matmult</td>
<td>5</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>minver</td>
<td>17</td>
<td>94.1%</td>
<td>0.01s</td>
<td>82.3%</td>
<td>0.28s</td>
</tr>
<tr>
<td>ndes</td>
<td>12</td>
<td>100.0%</td>
<td>0.04s</td>
<td>100.0%</td>
<td>0.04s</td>
</tr>
<tr>
<td>ns</td>
<td>4</td>
<td>100.0%</td>
<td>0.02s</td>
<td>100.0%</td>
<td>0.02s</td>
</tr>
<tr>
<td>nsichneu</td>
<td>1</td>
<td>0%</td>
<td>0.06s</td>
<td>0%</td>
<td>0.06s</td>
</tr>
<tr>
<td>qsort-exam</td>
<td>6</td>
<td>0%</td>
<td>&lt; 0.01s</td>
<td>0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>qurt</td>
<td>1</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>recursion</td>
<td>0</td>
<td>0%</td>
<td>&lt; 0.01s</td>
<td>0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>select</td>
<td>4</td>
<td>0%</td>
<td>&lt; 0.01s</td>
<td>0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>stateemate</td>
<td>1</td>
<td>0%</td>
<td>0.02s</td>
<td>0%</td>
<td>0.03s</td>
</tr>
<tr>
<td>sqrt</td>
<td>1</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>st</td>
<td>5</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
<td>100.0%</td>
<td>&lt; 0.01s</td>
</tr>
<tr>
<td>whet</td>
<td>10</td>
<td>100.0%</td>
<td>0.01s</td>
<td>80.0%</td>
<td>0.02s</td>
</tr>
</tbody>
</table>

**Total Percentage** 80.8% 72.3%
Loop bound and constraint analysis together succeed in analyzing both standardized benchmarks and real-world programs such as the debie spacecraft control system used in the WCET Tool Challenge 2008 [11], with only a handful of necessary manual annotations remaining.

Since the constraint analysis can also be adopted to derive loop bounds, we plan to replace the traditional loop bound analysis implementation by the constraint analysis eventually. Moreover, by offloading complexity into a separately maintained library, the analysis will automatically benefit from future improvements made to the solver. Thanks to its clean interface, we also retain the possibility to switch to different constraint solvers in the future.

References

APPENDIX

Interval Analysis

The interval analysis is an interprocedural data-flow problem. The variant we implemented in TuBound is an extension of the constant propagation analysis specified by Nielson, Nielson and Hankin [14]. Earlier work on interval analysis, also called value range propagation, was done by Harrison [10] and also by Cousot and Cousot [8]. The design parameters are sketched in Table 4. Just as constant propagation, the interval analysis is a forward-directed data-flow problem. The carrier of the analysis is a lattice of pairs of integers that are mapped to each integer variable. The members of the pairs denote the lower and upper bounds of the variables, respectively. If a bound is unknown, it is reported as $\pm \infty$. The $\perp$ element of the lattice means that a value has not yet been calculated, whereas $\top$ represents an unknown bound, which is equivalent to $(-\infty, \infty)$. At a control-flow join, the combine function is applied pairwise for each variable and merges the interval information coming from the different branches. The transfer functions for each statement capture the ramifications of the statement on the State lattice by abstractly interpreting the statement with interval arithmetic (function $A_{Itvl}$) [7]. The widening operator, which is used to speed up the fixed-point search is defined very aggressive, and can be used to fine-tune the trade-off between execution speed and analysis precision.

The transfer functions for conditional branches return different results for the true and false edges. If the branch condition statically evaluates to either $(1, 1)$ or $(0, 0)$, the state for the other branch is set to $\perp$, such that dead code cannot influence the analysis result for live branches.

The accuracy of the interval analysis can further be improved by increasing the memory and run-time budget: It can be modified to report a set of possible intervals instead of one merged interval for each variable.

Traditional loop bound analysis

The loop bound analysis is a control flow insensitive analysis that builds upon the results of the above interval analysis. The analysis takes as input

1. an iteration-variable based loop $L$,
2. variable intervals
3. and context information (such as the scope of $i$).

The analysis works on all iteration-variable based loops, with the restriction that the step size must be either positive or negative:

$$sgn(step_{min}) = sgn(step_{max})$$

The result of the analysis is an upper bound $n$ for the number of times the loop entry is executed in relation to its direct predecessor statements outside of the
Direction: forward
Lattice: $\text{State} = (\text{Var} \rightarrow (\mathbb{Z}^{-\infty}, \mathbb{Z}^{\infty}), \sqsubseteq, \sqcup, \sqcap, \bot, \lambda x.(-\infty, \infty))$
Init function: $\lambda x.(-\infty, \infty)$
Combine function: $\text{comb}((a_{\text{min}}, a_{\text{max}}), (b_{\text{min}}, b_{\text{max}})) = (\min(a_{\text{min}}, b_{\text{min}}), \max(a_{\text{max}}, b_{\text{max}}))$
Widening operator: $\text{widen}((a_{\text{min}}, a_{\text{max}}), (b_{\text{min}}, b_{\text{max}})) = (c_{\text{min}}, c_{\text{max}})$

where $c_{\text{min}} = \begin{cases} a_{\text{min}} & \text{if } a_{\text{min}} = b_{\text{min}} \\ -\infty & \text{otherwise} \end{cases}$
$c_{\text{max}} = \begin{cases} a_{\text{max}} & \text{if } a_{\text{max}} = b_{\text{max}} \\ \infty & \text{otherwise} \end{cases}$

Transfer functions:
$x := a' : f^{I}l_{t}(\sigma) = \begin{cases} \bot & \text{if } \sigma = \bot \\ \sigma[x \mapsto A_{I}l_{t}(a) \sigma] & \text{otherwise} \end{cases}$
$\text{if}(c)'_{\text{edge}} : f^{I}l_{t}(\sigma) = \begin{cases} \bot & \text{if } \sigma = \bot \\ \bot & \text{if } [A_{I}l_{t}(a) \sigma] = \text{edge} \\ f^{I}l_{t}^{'}(\sigma), [c]' & \text{otherwise} \end{cases}$

where
$A_{I}l_{t}[x] \sigma = \sigma(x)$
$A_{I}l_{t}[n] \sigma = (n, n)$
$A_{I}l_{t}[a \ op b] \sigma = A_{I}l_{t}[a] \sigma \ op_{I}l_{t} A_{I}l_{t}[b] \sigma$

Interval arithmetic:
$+_{I}l_{t}(a, b) = (a_{\text{min}} + b_{\text{min}}, a_{\text{max}} + b_{\text{max}})$
$-_{I}l_{t}(a, b) = (a_{\text{min}} - b_{\text{max}}, a_{\text{max}} - b_{\text{min}})$
$\cdot_{I}l_{t}(a, b) = (\min(a_{\text{min}}, b_{\text{min}}), \max(a_{\text{min}}, b_{\text{max}}))$
$/_{I}l_{t}(a, b) = (\max(a_{\text{min}}, a_{\text{max}}), \max(a_{\text{max}}/b_{\text{min}}, a_{\text{max}}/b_{\text{max}}))$
$=_{I}l_{t}(a, b) = \begin{cases} \bot & \text{if } a_{\text{min}} = b_{\text{min}} \land a_{\text{max}} = b_{\text{max}} \\ \top & \text{otherwise} \end{cases}$
$\neq_{I}l_{t}(a, b) = \begin{cases} \bot & \text{if } a_{\text{min}} = b_{\text{min}} \land a_{\text{max}} = b_{\text{max}} \\ true & \text{if } a_{\text{max}} < b_{\text{min}} \lor a_{\text{min}} > b_{\text{max}} \\ \top & \text{otherwise} \end{cases}$
$<_{I}l_{t}(a, b) = \begin{cases} \bot & \text{if } a_{\text{max}} < b_{\text{min}} \\ true & \text{if } a_{\text{min}} > b_{\text{max}} \\ \top & \text{otherwise} \end{cases}$

...
loop, where \( excnt(s) \) denotes the execution count of statement \( s \):

\[
\sum_{p \in \text{pred}(c) \setminus L} excnt(p) \leq n \times excnt(c)
\]

Since the discrete function described by the iteration step statement is monotone and its gradient is constant, we can set up the following equation for the loop bound:

\[
n = \frac{val_{\text{max}} - val_{\text{min}}}{|val_{\text{stepsize}}|}
\]

where \( val_{\text{min}}, val_{\text{max}} \) are lower and upper bounds for \( i \), whereas \( val_{\text{step}} \) is the minimum step size of \( i \) on a path through the loop \( L \). We call these values loop parameters. To derive the loop parameters, it is necessary to examine the relational operator of the exit condition, which must be one of \(<, >, \leq, \geq\).

<table>
<thead>
<tr>
<th>case rel of</th>
<th>LowExpr = a, HighExpr = b</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \leq )</td>
<td>LowExpr = a, HighExpr = b + 1</td>
</tr>
<tr>
<td>( &gt; )</td>
<td>LowExpr = b, HighExpr = a</td>
</tr>
<tr>
<td>( \geq )</td>
<td>LowExpr = b, HighExpr = a - 1</td>
</tr>
</tbody>
</table>

\[ StepExpr = c \]

**Table 5.** Deriving the loop parameters

As shown in Table 5, the assignment of \( LowExpr \) and \( HighExpr \) depends on the direction of the loop. In our implementation, concrete values of the loop parameters are calculated in two phases:

1. **Simplify.** In this phase, algebraic identities are exploited to simplify the expression \( (HighExpr - LowExpr)/StepExpr \). This is implemented by a set of rewrite rules that are applied to the expression until a fixed point is reached. This simplification operates on purely symbolic expressions and disregards the analyzed intervals of variables. It can, however, use the information that an expression is loop invariant or constant, i.e. no variable occurring in it appears on the left-hand side of any statement in \( L \).
2. **Evaluate.** Using the results of the interval analysis as state, we can evaluate the simplified expression using interval arithmetic \([7]\) \( (A_{Itvl}) \). The return value is an interval \((m, n)\) where \( n \) is the upper bound for the iteration count of the loop \( L \).

The complexity of this algorithm is bounded by the number of exit conditions in the loop, the depth of \( Low \), \( High \) and \( Step \) expressions and the number of rules in the simplification term replacing system.
Better Termination for Prolog with Constraints

Markus Triska¹, Ulrich Neumerkel¹, and Jan Wielemaker²

¹ Technische Universität Wien, Austria
² Universiteit van Amsterdam, The Netherlands

Abstract. Termination properties of actual Prolog systems with constraints are fragile and difficult to analyse. The lack of the occurs-check, moded and overloaded arithmetical evaluation via is/2 and the occasional nontermination of finite domain constraints are all sources for invalidating termination results obtained by current termination analysers that rely on idealized assumptions. In this paper, we present solutions to address these problems on the level of the underlying Prolog system. Improved unification modes meet the requirements of norm based analysers by offering dynamic occurs-check detection. A generalized finite domain solver overcomes the shortcomings of conventional arithmetic without significant runtime overhead. The solver offers unbounded domains, yet propagation always terminates. Our work improves Prolog’s termination and makes Prolog a more reliable target for termination and type analysis. It is part of SWI-Prolog since version 5.6.50.

1 Introduction

Termination plays a central role in Prolog programs. Prolog’s complex control mechanism often taxes a programmer’s intuition about termination. Tools to support both experts and beginners are therefore highly valuable and the development of such systems has received considerable attention [8, 4, 12]. One of the authors was particularly interested in developing termination tools for supporting beginners within the learning environment GUPU [14]. In a collaborative effort, the termination inference system cTI [12] was developed that featured not only a web interface but was designed to specifically meet the incremental demands for an on-the-fly analyser by employing a strict bottom up approach.

Much to our chagrin, the resulting system soon showed the limitations of current approaches for our original goals. cTI worked quite impressively for current benchmarks but did not reflect the entire spectrum of termination properties of actual Prolog implementations. cTI—like most other norm based approaches [4]— was founded on some assumptions that are not true for existing Prolog systems. As a consequence, the termination conditions inferred with cTI are not literally applicable to the target system—at that time SICStus Prolog. We note that these problems do not show in existing termination benchmarks, but are frequently occurring in the incorrect programs beginners write. The source of
the problem is the lack of the occurs-check in existing Prolog implementations giving way to rational trees that can no longer be mapped onto the integers. While there are approaches to determine occurs-check freeness statically [6], as well as finite trees [2] we finally chose to go for the maximum which is performing the occurs-check dynamically.

With the addition of constraints to Prolog’s core language, new sources of unforeseen nontermination opened, further complicating procedural reasoning. The traditional \texttt{is/2} predicate with its overloaded semantics posed even more problems. To meet all these needs we implemented a new version of a generalized finite domain solver. This library subsumes the functionality of integer arithmetic and constraint programming, combines their strengths, and terminates always, permitting better termination results.

\textit{Content.} We first describe our new approach to the old occurs-check problem and then discuss our improvement to clpfd to subsume \texttt{is/2}-functionality. Finally we present our new always terminating implementation of clpfd.

2 Occurs-check

Most existing Prolog implementations use rational tree unification [5] to avoid overheads caused by the occurs-check of finite tree unification. While rational trees are an interesting domain in their own right, they are often an indication for programming errors. For beginners, it is very common to accidentally confuse assignment and unification. Goals like \texttt{Xs = [X|Xs]} are often written with the intention to add to the list \texttt{Xs} an element. Also misunderstandings concerning the scoping of variables lead to infinite terms. Exactly such cases are not covered by existing norm based approaches that assume the finiteness of terms.

We added two new standard conforming unification modes that prevent the creation of infinite terms. Apart from traditional occurs-check that fails silently, a new mode was added to better localize attempts to create infinite terms. By issuing \texttt{?- set_prolog_flag(occurs_check, error).} at runtime all attempts to create infinite terms are detected and an error is issued. In this manner all programs are identified that create infinite terms. Also, most programs subject to occurs-check (STO) are detected, that are ruled out by the ISO standard [10].

Our implementation tries to avoid the costly occurs-check scan for the most frequent cases of passing variables. Current Prolog implementations allocate variables that do not occur within a structure in a separate storage area, mostly known as the goal or environment stack. Those variables are unified in constant time with structured terms, as they cannot be the subterm of a structure. In this manner most uses of difference lists and differences with other data structures do not require the occurs-check. The actual testing can be further reduced taking into account that Prolog compilers emit specialised unification instructions where possible, based on its knowledge about the arguments involved in the unification. Only the cases of instructions of general unification are subject to occurs check. All other cases do not involve any overhead. As of version 5.7, all
overheads for handling the list differences of DCGs are completely removed for an initial goal \texttt{phrase/2}. For \texttt{phrase/3} there is a single occurs-check for each solution found.

3 Overcoming \texttt{is/2}

Using \texttt{is/2} in pure programs has many disadvantages. For one, \texttt{is/2} works only for certain restricted modes thereby limiting the relational view of a predicate. This relational view permits to test programs more extensively—testing them with generalized modes. Even if those generalized modes are not used in the final application, they help to detect otherwise undiscovered problems. Consider for example McCarthy’s “mysterious” 91-function. With the following query we search for results different to 91.

```prolog
mc_carthy_91(X, Y) :-
    X #> 100, Y #= X - 10.
mc_carthy_91(X, Y) :-
    X #=< 100, Z #= X + 11,
    mc_carthy_91(Z, Z1),
    mc_carthy_91(Z1, Y).
?- Y \#\= 91, mc_carthy_91(X, Y).
```

Attempts to emulate with \texttt{is/2} different modes require the explicit usage of \texttt{var/1} and \texttt{nonvar/1}, two built-ins that lead frequently to errors due to forgotten modes.

The overloading of integer and floating-point arithmetic is another source of frequent errors with \texttt{is/2}. An accidentally introduced float might lead to unexpected failures. Modeling without knowing whether or not a variable is a float is not reliably possible, thereby weakening termination analysis [8].

For these reasons we propose to use in place of \texttt{is/2} the corresponding \texttt{=/2} of clpfd and the corresponding comparison relations. To make this shift more practical we removed the common limits of \texttt{=/2} to small integers and improved execution for such simple moded cases. While using \texttt{=/2} in place of \texttt{is/2} incurred overheads greater than two orders of magnitude for small loops, our improved implementation is only about 30\% slower than naive \texttt{is/2}. In this manner, we obtain predicates that are simpler to type and that are not moded.

The original version of \texttt{factorial/2} is not tail recursive due to the modedness of \texttt{is/2}. The space for allocating the environments in the original version is traded for allocating constraints on the global stack. \texttt{factorial/2} now terminates if either the first argument is finite, or the second argument is finite and not equal zero.
factorial(0, 1).
factorial(N, F) :-
    N #> 0,
    F #= F0*N,
    N1 #= N - 1,
    factorial(N1, F0).

?- Y in 1..5, factorial(X,Y).
Y = 1,
X = 0 ;
Y = 1,
X = 1 ;
Y = 2,
X = 2 ;
false.

4 Terminating constraints

Current implementations of finite domain constraints are optimized for the traditional usage pattern of constraint satisfaction. First, variables get their associated domains, then the constraints between variables are posted, and finally labeling searches for actual solutions. In current implementations, the declaration of a variable’s domain is just a simple goal. (Original systems required a static declaration.) The extension from this limited view toward a general constraint systems over integers, a kind of CLP(Z), is straightforward.

By accepting variables without a finite domain, we open the door to nonterminating constraint propagation. Consider the query ?- X#>Y, Y#>X, X#>0. Existing constraint solvers will try to reduce the domains until the maximal domain value is encountered, then failing or yielding a representation error. We therefore consider this case the same as genuine nontermination. Note that non-termination does not only occur due to posting a constraint but also may happen during labeling.

?- X#>Y, Y#>X, X#>B*Y, B in -1..0, labeling([],B)

Termination within constraint propagation is ensured by propagating domain changes in infinite domains only once. At the price of weakening consistency we can now guarantee that clpfd and all unifications with constrained variables terminate.

4.1 Observing termination

The notion of termination and nontermination are idealizations of actual observable behavior that lead to seemingly paradoxical situations. The query ?- X#>X*X. terminates rapidly in SICStus 3 with a representation error. Still, we consider this a case of non-termination. For ?- abs(X)#<7^7^7,X#>Y,Y#>X. in SWI,
termination is not observable within reasonable time. However, we consider this case terminating.

Another rather unintuitive consequence concerns the termination property of the entire program. While our improvement guarantees termination for unification and all clpfd-goals, and therefore might improve termination of the entire program, there are cases where a stronger propagation that does not terminate in the general case will nevertheless result in better termination of the entire program. This may happen, if the stronger propagation results in failure preventing an infinite loop, while terminating propagation yields inconsistency.

4.2 Ad hoc termination proofs

With an always terminating clpfd, we are able to perform some simple forms of termination testing when using labeling. One frequent problem with larger constraint problems concerns the time span to wait for the first solution. Quite often labeling is considered to be inefficient, when in reality the actual predicate definition that posts the constraints does not terminate. To avoid this situation we separate the actual relation from labeling. In place of the original predicate $p/n$ we define a new relation $p_+/n+1$ (“core relation”) that contains an additional argument for the list of variables to be labeled. Consider for example as original query 
\[- \texttt{queens}(Ds), \]
 describing solutions for a given fixed length of $Ds$. This query is now formulated as 
\[- \texttt{queens}_(Ds,Zs), \texttt{labeling}([],Zs). \]
 Suppose now that the answer does not appear immediately. Should we wait for an answer? What, if the query does not terminate? To better understand the termination properties involved we can consider the following query. If 
\[- \texttt{queens}_(Ds,Zs), \texttt{false}. \]
 terminates (by observation), we know also that the query followed by labeling will terminate, since in our implementation $\texttt{labeling/2}$ is guaranteed to terminate. We thus obtain a proof for termination by observing the termination of another related predicate. In systems without our favorable termination property, a terminating 
\[- \texttt{queens}_(Ds,Zs), \texttt{false}. \]
 does not constitute a termination proof of the goal followed by a search with $\texttt{labeling/2}$.

4.3 Black-box testing

While developing and testing library(clpfd), it soon became evident that manual testing and testing with given applications is not sufficient. We noted as one of the most prominent coding errors the omission of certain rare cases of instantiations. The current implementation in Prolog based on hProlog-style attributed variables [7] does not guarantee any properties concerning the correctness of the implementation. The concerns consistency and correctness must be dealt with on the same level - thereby increasing the chance for errors. As one of the authors experienced similar problems with other constraint implementations prior to SWI, it was evident that a more systematic approach was needed. Existing approaches to testing and specifying finite domain constraints [1] were also not very attractive, as they require considerable effort for specifying the actual propagation mechanism. Such complex specifications may again be a further source
of errors. We therefore focused on testing with strictly minimal information - thereby minimizing demotivating cases of false alarms.

We concentrated on testing a fixed set of algebraic properties for small finite domains. So far, all encountered correctness errors could be shown to violate those properties. We illustrate our approach with an error located in this manner (i3a#98). The query \texttt{?- X in 0..2, 0/X#0.} should succeed, but failed. Even to the experienced constraint programmer it is not obvious by naively inspecting this query what the correct result should be. The bug was located automatically by detecting a difference between the following two queries:

\begin{verbatim}
?- X in 0..2, 0/X #= 0, X = 1.
?- X = 1, X in 0..2, 0/X #= 0.
\end{verbatim}

The first query failed, the second succeeded. Evidently, there must be at least one error—either in the first or second query, or in both. Most errors found are related to the implementation of nonlinear constraints like general multiplication. Also, sharing of variables was a frequent cause for errors. In total, more than 30 errors of this kind were found by systematically exploring a tiny slice of all possible formulae.

For efficient testing, (rapid) termination of clpfd’s propagation is indispensable. This permits to test many queries simultaneously. On systems with nonterminating propagation, we would have to rely on timeout mechanisms to interrupt certain queries that cannot be tested in this way.

4.4 Related work

SICStus Prolog [3] was the first system to generalize finite domain constraints without sacrificing correctness. It uses small integers for domains but signals domain overflows as representation errors and not as silent failures.

Built-in support for the occurs-check has been implemented with similar techniques in Sepia Prolog [13] and its successor Eclipse Prolog [16].

5 Conclusions

The presented improvements constitute a more solid target for termination analysis than prior implementations. We hope that they will lead to the development of more powerful analysers.

Acknowledgements. We would like to thank Philipp Kolmann of ZID for his generous support on the Condor based [15] WINZIG-grid [11] at TU Wien that we use to perform large scale tests.
References

Author Index

Adachi, Yoshihiro .......................................................... 8
Albert, Elvira ............................................................... 26
Bendisposto, Jens ........................................................... 18
Endrijautzki, Ian ........................................................... 18
Gómez-Zamalloa, Miguel .............................................. 26
King, Andy ................................................................. 1
Knoop, Jens ................................................................. 77
Leuschel, Michael ......................................................... 18, 44
López-Fraguas, Francisco Javier ..................................... 52
Muñoz Hernández, Susana ............................................. 62
Neumerkel, Ulrich ......................................................... 90
Pablos Ceruelo, Victor .................................................... 62
Prantl, Adrian .............................................................. 77
Puebla, Germán ............................................................ 26
Rodríguez-Hortalá, Juan .................................................. 52
Sánchez-Hernández, Jaime ............................................. 52
Schneider, David .......................................................... 18
Schordan, Markus ........................................................ 77
Strass, Hannes ............................................................. 62
Tamarit, Salvador ........................................................ 44
Triska, Markus ............................................................. 77, 90
Vidal, Germán .............................................................. 44
Wielemaker, Jan ............................................................ 90