RWTH Aachen

Department of Computer Science
Technical Report

Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014

Thomas Ströder and Terrance Swift (Editors)

ISSN 0935–3232 · Aachener Informatik-Berichte · AIB-2014-09

RWTH Aachen · Department of Computer Science · June 2014 (revised version)

Table of Contents

VIENNA
SUMMER
OF LOGIC
2014

In the summer of 2014, Vienna hosted the largest scientific conference in the history of logic. The Vienna Summer of Logic (VSL, http://vsl2014.at) consisted of twelve large conferences and 82 workshops, attracting more than 2000 researchers from all over the world. This unique event was organized by the Kurt Gödel Society at Vienna University of Technology from July 9 to 24, 2014, under the auspices of the Federal President of the Republic of Austria, Dr. Heinz Fischer.

The conferences and workshops dealt with the main theme, logic, from three important angles: logic in computer science, mathematical logic, and logic in artificial intelligence. They naturally gave rise to respective streams gathering the following meetings:

Logic in Computer Science / Federated Logic Conference (FLoC)

- 26th International Conference on Computer Aided Verification (CAV)
- 27th IEEE Computer Security Foundations Symposium (CSF)
- 30th International Conference on Logic Programming (ICLP)
- 7th International Joint Conference on Automated Reasoning (IJCAR)
- 5th Conference on Interactive Theorem Proving (ITP)
- Joint meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 29th ACM/IEEE Symposium on Logic in Computer Science (LICS)
- 25th International Conference on Rewriting Techniques and Applications (RTA) joint with the 12th International Conference on Typed Lambda Calculi and Applications (TLCA)
- 17th International Conference on Theory and Applications of Satisfiability Testing (SAT)
- 76 FLoC Workshops
- FLoC Olympic Games (System Competitions)

Mathematical Logic

- Logic Colloquium 2014 (LC)
- Logic, Algebra and Truth Degrees 2014 (LATD)
- Compositional Meaning in Logic (GeTFun 2.0)
- The Infinity Workshop (INFINITY)
- Workshop on Logic and Games (LG)
- Kurt Gödel Fellowship Competition

Logic in Artificial Intelligence

- 14th International Conference on Principles of Knowledge Representation and Reasoning (KR)
- 27th International Workshop on Description Logics (DL)
- 15th International Workshop on Non-Monotonic Reasoning (NMR)
- 6th International Workshop on Knowledge Representation for Health Care 2014 (KR4HC)

The VSL keynote talks which were directed to all participants were given by Franz Baader (Technische Universität Dresden), Edmund Clarke (Carnegie Mellon University), Christos Papadimitriou (University of California, Berkeley) and Alex Wilkie (University of Manchester); Dana Scott (Carnegie Mellon University) spoke in the opening session. Since the Vienna Summer of Logic contained more than a hundred invited talks, it is infeasible to list them here.

The program of the Vienna Summer of Logic was very rich, including not only scientific talks, poster sessions and panels, but also two distinctive events. One was the award ceremony of the *Kurt Gödel Research Prize Fellowship Competition*, in which the Kurt Gödel Society awarded three research fellowship prizes endowed with 100.000 Euro each to the winners. This was the third edition of the competition, themed *Logical Mind: Connecting Foundations and Technology* this year.

The other distinctive event was the *1st FLoC Olympic Games* hosted by the Federated Logic Conference (FLoC) 2014. Intended as a new FLoC element, the Games brought together 12 established logic solver competitions by different research communities. In addition to the competitions, the Olympic Games facilitated the exchange of expertise between communities, and increased the visibility and impact of state-of-the-art solver technology. The winners in the competition categories were honored with Kurt Gödel medals at the FLoC Olympic Games award ceremonies.

Organizing an event like the Vienna Summer of Logic has been a challenge. We are indebted to numerous people whose enormous efforts were essential in making this vision become reality. With so many colleagues and friends working with us, we are unable to list them individually here. Nevertheless, as representatives of the three streams of VSL, we would like to particularly express our gratitude to all people who have helped to make this event a success: the sponsors and the honorary committee; the organization committee and the local organizers; the conference and workshop chairs and program committee members; the reviewers and authors; and of course all speakers and participants of the many conferences, workshops and competitions.

The Vienna Summer of Logic continues a great legacy of scientific thought that started in Ancient Greece and flourished in the city of Gödel, Wittgenstein and the Vienna Circle. The heroes of our intellectual past shaped the scientific world-view and changed our understanding of science. Owing to their achievements, logic has permeated a wide range of disciplines, including computer science, mathematics, artificial intelligence, philosophy, linguistics, and many more. Logic is everywhere – or in the language of Aristotle, πάντα πλήρη λογικῆς τέχνης.

Vienna, July 2014

Matthias Baaz, Thomas Eiter, Helmut Veith

CICLOPS-WLPE 2014: PREFACE

Thomas Ströder Terrance Swift

July 17-18, 2014 · Vienna, Austria

`http://vsl2014.at/ciclops-wlpe/`

Software plays a crucial role in modern society. While the continuous advent of faster, smaller and more powerful computing devices makes the development of new and interesting applications feasible, it puts even more demands on the software developer. Indeed, while software keeps on growing in size and complexity, it is more than ever required to be delivered on time, free of error and to meet the most stringent efficiency requirements. Consequently, it is widely recognized that there is a need for methods and tools that support the programmer in every aspect of the software development process.

Having logic as the underlying formalism means that logic-based analysis techniques are often successfully used for program verification and optimization. Emerging programming paradigms and growing complexity of the properties to be verified pose new challenges for the community, while emerging reasoning techniques can be exploited.

The International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS) provides a forum to discuss the design, implementation, and optimization of logic, constraint (logic) programming systems, and other systems based on logic as a means of expressing computations. Experience backed up by real implementations and their evaluation is given preference, as well as descriptions of work in progress in that direction.

The aim of the Workshop on Logic-based methods in Programming Environments (WLPE) is to provide an informal meeting for researchers working on logic-based methods and tools that support program development and analysis. As in recent years, these topics include not only environmental tools for logic programming, but increasingly also logic-based environmental tools for programming in general and frameworks and resources for sharing in the logic programming community.

The combination of these two areas of interest in this year's joint workshop provides a forum to discuss together the states of the art for using logic both in the evaluation of programs and in meta-reasoning about programs.

Due to the strong overlap between the CICLOPS-WLPE community and several FLoC communities (in particular logic (programming), verification, automated reasoning, rewriting techniques, and SAT solving), the workshop is affiliated to several conferences:

- 30th International Conference on Logic Programming (ICLP)
- 26th International Conference on Computer Aided Verification (CAV)
- 7th International Joint Conference on Automated Reasoning (IJCAR)
- Joint meeting of the 23rd EACSL Annual Conference on Computer Science Logic (CSL) and the 9th ACM/IEEE Symposium on Logic in Computer Science (LICS)
- 25th International Conference on Rewriting Techniques and Applications (RTA) joined with the 12th International Conference on Typed Lambda Calculi and Applications (TLCA)
- 17th International Conference on Theory and Applications of Satisfiability Testing (SAT)

In 2014, CICLOPS-WLPE also joins its program with the 11th International Workshop on Constraint Handling Rules (CHR) and has one joint session together with the Workshop on Probabilistic Logic Programming (PLP).

The International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014 consists of nine regular submissions and two invited talks. These informal proceedings contain the regular papers and the abstracts of the two invited talks.

We would like to thank all people involved in the preparation and execution of the workshop, including the participants, the members of the program committee, and the local organizers.

Thomas Ströder and Terrance Swift
CICLOPS-WLPE 2014 Program Co-Chairs

Program Committee

Michael Codish	Ben-Gurion University
Daniel De Schreye	Katholieke Universiteit Leuven
Carsten Fuhs	University College London
John Gallagher	Roskilde University
Marco Gavanelli	University of Ferrara
Michael Hanus	CAU Kiel
Gerda Janssens	Katholieke Universiteit Leuven
Yoshitaka Kameya	Meijo University
Matthias Knorr	CENTRIA, Universidade Nova de Lisboa
Jael Kriener	University of Kent
Joachim Schimpf	Coninfer Ltd, London
Peter Schneider-Kamp	University of Southern Denmark
Tobias Schubert	Albert-Ludwigs-University Freiburg
Thomas Ströder	RWTH Aachen University
Terrance Swift	Universidade Nova de Lisboa
Christian Theil Have	University of Copenhagen
German Vidal	MiST, DSIC, Universitat Politecnica de Valencia
Jan Wielemaker	VU University Amsterdam

Additional Reviewers

Broes De Cat	Katholieke Universiteit Leuven
Bart Demoen	Katholieke Universiteit Leuven

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

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

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

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

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

More information can be found in [1].

References

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

* Supported by the DFG under grants GI 274/5-3 and GI 274/6-1, the DFG Research Training Group 1298 (*AlgoSyn*), and the Danish Council for Independent Research, Natural Sciences.

Proofs for Optimality of Sorting Networks by Logic Programming

Michael Codish[1], Luís Cruz-Filipe[2], Michael Frank[1], and Peter Schneider-Kamp[2]

[1] Department of Computer Science, Ben-Gurion University of the Negev, Israel
{mcodish,frankm}@cs.bgu.ac.il

[2] Department of Mathematics and Computer Science, University of Southern Denmark, Denmark
{lcf,petersk}@imada.sdu.dk

Abstract. We present a computer-assisted non-existence proof of nine-input sorting networks consisting of 24 comparators, hence showing that the 25-comparator sorting network found by Floyd in 1964 is optimal. As a corollary, we obtain that the 29-comparator network found by Waksman in 1969 is optimal when sorting ten inputs. This closes the two smallest open instances of the optimal size sorting network problem, which have been open since the results of Floyd and Knuth from 1966 proving optimality for sorting networks of up to eight inputs.
The entire implementation is written in SWI-Prolog and was run on a cluster of 12 nodes with 12 cores each, able to run a total of 288 concurrent threads, making extensive use of SWI-Prolog's built-in predicate `concurrency/3`. The search space of 2.2×10^{37} comparator networks was exhausted after just under 10 days of computation. This shows the ability of logic programming to provide a scalable parallel implementation while at the same time instilling a high level of trust in the correctness of the proof.

Refining definitions with unknown opens using XSB for IDP^3

Joachim Jansen, Gerda Janssens

Department of Computer Science, KU Leuven
joachim.jansen, gerda.janssens@cs.kuleuven.be

Abstract. FO$(\cdot)^{\text{IDP}}$ is a declarative modeling language that extends first-order logic with inductive definitions, partial functions, types and aggregates. Its model generator IDP^3 grounds the problem into a low-level (propositional) representation and consequently use a generic solver to search for a solution. Recent work introduced a technique that evaluates all definitions that depend on fully known information before the grounding step. In this paper, we extend this technique, which allows us to refine the interpretation of defined symbols when they depend on information that is only partially given instead of completely given. We use our existing transformation of FO$(\cdot)^{\text{IDP}}$ definitions to Tabled Prolog rules and extend it to support definitions that depend on information that is possibly partially unknown. In this paper we present an algorithm that uses XSB Prolog to evaluate these rules in such a way that we achieve the most precise possible refinement of the defined symbols. Experimental results show that our technique derives extra information for the defined symbols.

1 Introduction

Recent proposals for declarative modeling use first-order logic as their starting point. Examples are Enfragmo [1] and FO$(\cdot)^{\text{IDP}}$, the instance of the FO$(\cdot)$ family that is supported by IDP^3, the current version of the IDP Knowledge Base System [6]. FO$(\cdot)^{\text{IDP}}$ extends first-order logic (FO) with inductive definitions, partial functions, types and aggregates. IDP^3 supports model generation and model expansion [11, 4] as inference methods.

IDP^3 supports these inference methods using the ground-and-solve approach. First the problem is grounded into an Extended CNF (ECNF) theory. Next a SAT-solver is used to calculate a model of the propositional theory. The technique that is presented in this paper is to improve the efficiency and robustness of the grounding step. One of the problems when grounding is the possible combinatorial blowup of the grounding. A predicate $p(x_1, x_2 \ldots x_n)$ with s as the size of the domain of its arguments has s^n possible instances. A grounding that has to represent all these possible instances is therefore possibly very large. Most Answer Set Programming (ASP) systems solve this problem by using semi-naive bottom-up evaluation [8, 9] with optimizations. On a high level IDP^3 uses three techniques to manage the complexity of the grounding process: definition evaluation [10], Lifted Unit Propagation (LUP) [15] and Grounding With Bounds (GWB) [16].

Our previous work [10] is a pre-processing step that calculates in advance the two-valued interpretations for defined predicates that depend on fully known information. We call such defined predicates input$*$ predicates. The definitions of these predicates and the information on which they depend are translated into a XSB Prolog [12] program that tables the defined input$*$ predicates, supporting the well-founded semantics [14, 13]. This Tabled Prolog program is then queried to retrieve the atoms for which the tabled predicates are true. The input structure (the initially given partial structure) is extended with the calculated information. The input$*$ predicates become completely known: they are true for the tabled atoms and false for all the other instances. As a result, definitions of the input$*$ predicates are no longer needed and they are repoved from the problem specification.

Lifted Unit Propagation (LUP) is another preprocessing step that further refines the partial structure. LUP propagates knowledge about `true` and `false` atoms in the formulas of the $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ theory. For the definitions of the $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ theory LUP uses an approximation by propagating on the completion of the definitions. The method of this paper is an alternative for using LUP on the completion of the definitions. We extend our existing preprocessing step [10] to be able to refine the interpretation of defined predicates in the partial structure when the predicates depend on information that is only partially given. This extension can then be used as an alternative to executing LUP on the completion of definitions. Our method uses XSB to compute the atoms (instances of the predicate) that are `true` and others that are `unknown`. The computed atoms are used to refine the partial structure. Moreover, XSB's support for the well-founded semantics makes atoms false when XSB detects unfoundedness. This detection of unfoundedness is not present in the approach that uses LUP on the completion of definitions to refine them.

Grounding With Bounds (GWB) uses symbolic reasoning when grounding subformulas to derive bounds. Because GWB uses the input structure, it can benefit from the extra information that is inferred thanks to the refinement done by LUP. Using this extra information, possibly tighter bounds can be derived. Therefor it is beneficial to refine the input structure as much as possible before grounding (using GWB). Because of this, we will measure the effectiveness of the discussed methods by how much they are able to refine the input structure. The actual grounding process that will benefit from this refined structure is considered out of scope for this paper.

Our contribution is a new way to perform lifted propagation for definitions. Experimental results compare the new technique with the old one of performing LUP for the completion of the definition.

In Section 2 we introduce IDP^3 and $\mathrm{FO}(\cdot)$. Section 3 explains our approach using an example. In Section 4 we describe the extensions to the transformation to Tabled Prolog rules and the workflow of our interaction with XSB. Section 5 presents the high-level algorithm that is used to refine all defined symbols as much as possible using the previously mentioned XSB interaction. In Section 6 we present experimental results. Section 7 contains future work and concludes.

2 Terminology and Motivation

2.1 The $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ language

We focus on the aspects of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ that are relevant for this paper. More details can be found in [6] and [2], where one can find several examples. An $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ model consists of a number of logical components, a.o. vocabularies, structures, and theories.

A *vocabulary* declares the symbols to be used.

A *structure* is used to specify the domain and data; it provides an interpretation of the symbols in the vocabulary. The interpretation of a symbol specifies for this symbol which atoms (instances) are `true`, `unknown`, and `false`. Interpretations that contain elements that are `unknown` are also called a partial (or three-valued) interpretation. Otherwise, the interpretation is said to be two-valued.

A *theory* consists of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ formulas and definitions. An $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ *formula* differs from FO formulas in two ways. Firstly, $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ is a many-sorted logic: every variable has an associated *type* and every type an associated domain. Moreover, it is order-sorted: types can be subtypes of others. Secondly, besides the standard terms in FO, $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ formulas can also have aggregate terms: functions over a set of domain elements and associated numeric values which map to the sum, product, cardinality, maximum or minimum value of the set.

An $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$*definition* is a set of *rules* of the form $\forall \bar{x} : p(\bar{x}) \leftarrow \phi[\bar{x}]$. where $\phi[\bar{x}]$ is an $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ formula. We call $p(\bar{x})$ the *head* of the rule and $\phi[\bar{x}]$. the *body* of the rule. The *defined* symbols of a theory are the symbols that appear in a head of any rule. The other symbols, which appear only in bodies of definitions are the *open* symbols. We remind the reader that previous work [10] describes a transformation of $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ definitions into Tabled Prolog rules. This includes a transformation of the interpretation of the open symbols to (Tabled) Prolog facts.

2.2 The IDP^3 system

IDP^3 is a Knowledge Base System [6], meaning it supports a variety of problem-solving inferences. One of these inferences is model expansion. The model expansion of IDP^3 extends a partial structure (an interpretation) into a two-valued structure that satisfies all constraints specified by the $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ model. Formally, the task of model expansion is, given a vocabulary V, a theory T over V and a partial structure S over V (at least interpreting all types), to find a two-valued structure M that satisfies T and extends S, i.e., M is a model of the theory and the input structure S is a subset of M.

As mentioned before, IDP^3 uses the ground-and-solve approach. It grounds the problem and then uses the solver MiniSAT(ID) [3,5], based on the solver MiniSAT [7].

There are three techniques that IDP^3 uses to optimise its grounding process: definition evaluation [10], Lifted Unit Propagation (LUP) [15] and Grounding With Bounds (GWB) [16].

Our previous work [10] introduces a pre-processing step that reduces the IDP^3 grounding by calculating some definitions in advance. We calculate the two-valued interpretations for defined predicates that depend on completely known information. We transform the relevant definition into Tabled Prolog rules, we add the relevant fragment of the input structure as Prolog facts, and we query XSB for the desired interpretation. We use the computed atoms to complete the two-valued interpretation for the defined symbols. The definitions are no longer needed and can be removed from the theory.

LUP can most easily be explained based on what SAT solvers do. Most SAT solvers start by performing Unit Propagation (UP) on the input to derive new information about the search problem. LUP is designed to refine the input structure using unit propagation, but on the $\mathrm{FO}(\cdot)^{\mathtt{IDP}}$ formulas instead of on the ground representation, which is why it is called “lifted”. It is important to note that LUP only refines the structure with respect to the *formulas* and not w.r.t. the *definitions*. To resolve this, LUP is executed for the completion of the definitions, but this is an approximation of what can be derived from definitions. In this paper, we extend the technique used to evaluate definitions to perform lifted propagation on definitions that have opens with a three-valued interpretation. This extension can then be used as an alternative to executing LUP on the completion of definitions.

GWB uses symbolic reasoning when grounding subformulas. Given the input structure, it derives bounds for certainly true, certainly false and unknown for quantified variables over (sub)formulas. Consequentially, since GWB uses the structure, it can benefit from the extra information that is inferred thanks to the refinement done by LUP. Using this extra information, possibly tighter bounds can be derived.

3 Example of refining structures

The example shown in Figure 1 expresses a reachability problem for colored nodes using undirected edges. We use this example to illustrate some of the concepts.

The theory T in the example contains one formula and two definitions: one definition defines the symbol $uedge/2$ and the other definition defines $reach/2$. We abuse notation and use “$uedge/2$ definition” to denote the definition defining the $uedge/2$ symbol. The $uedge/2$ definition has only one open symbol: $edge/2$. Because $edge/2$ has a two-valued interpretation, our original method [10] is applicable, so we perform definition evaluation for the $uedge/2$ definition. The calculated interpretation for $uedge/2$ can be seen in $S2$, depicted in Figure 2.

```
vocabulary V {
  type node isa int      type color constructed from {RED, BLUE}
  edge(node,node)        uedge(node,node)
  color(node,color)      reach(node,color)
  start(node)
}
theory T : V {
  { uedge(x,y)  ← edge(x,y) ∨ edge(y,x). }
  { reach(x,c) ← start(x).
    reach(y,c) ← reach(x,c) ∧ uedge(x,y) ∧ color(y,c). }
  ∀x: color(x,RED) ⇔ ¬color(x,BLUE).
}
structure S : V {
  node       = {1..6}              start      = {1}
  color<ct> = {2,RED}             color<cf> = {3,RED}
  edge       = {1,2; 3,1; 3,5; 4,2; 4,3; 6,6}
}
```

Fig. 1. An IDP^3 problem specification example. The notation color<ct> and color<cf> is used to specify which elements are certainly true, respectively certainly false for the color(node,color) relation in structure S. Tuples that are in neither of these specifications are unknown.

The *reach*/2 definition has three open symbols: *start*/1, *uedge*/2, and *color*/2. Because *color*/2 has a three-valued interpretation, we cannot perform definition evaluation for the *reach*/2 definition. This concludes what can be done with regards to definition evaluation and we proceed with the theory $T2$ and $S2$ as depicted in Figure 2. For compactness, $S2$ only shows symbols for which the interpretation has changed with regards to S. The vocabulary remains the same as in Figure 1.

Next, we can perform Lifted Unit Propagation for the formula in $T2$. This formula expresses that when a node is RED, it cannot be BLUE and vice versa. Since the structure $S2$ specifies that node 3 cannot be RED, we derive that node 3 has to be BLUE. In the same manner we can derive that node 2 cannot be BLUE. This results in structure $S3$ as depicted in Figure 2. The theory remains the same as in Figure 2.

This leaves us with the *reach*/2 definition to further refine $S3$. There are two approaches to performing lifted propagation on this definition: first we can perform LUP on the completion of the *reach*/2 definition or alternatively, we use the new method introduced in this paper. Structure $S4$ in Figure 4 shows what can be derived using the existing LUP method on the completion of the *reach*/2 definition, which is the following equivalence:

$$\forall y\ c : reach(y,c) \Leftrightarrow start(y) \lor \exists x : (reach(x,c) \land uedge(x,y) \land color(y,c)).$$

Note that in $S4$ node 1 is reachable using BLUE as well as RED because the first rule in the *reach*/2 definition says the starting node is always reachable

```
theory T2 : V {
  { reach(x,c) ← start(x).
    reach(y,c) ← reach(x,c) ∧ uedge(x,y) ∧ color(y,c). }
  ∀x: color(x,RED) ⇔ ¬color(x,BLUE).
}
structure S2 : V {
  uedge = { 1,2; 1,3; 2,1; 2,4; 3,1; 3,4; 3,5; 4,2;
            4,3; 5,3; 6,6 }
}
```

Fig. 2. The theory and structure after performing definition evaluation on T and S. The interpretation of *node*, *start*/1, *color*/2 and *edge*/2 remains the same as in S.

```
structure S3 : V {
  color<ct> = {2,RED; 3,BLUE}      color<cf> = {2,BLUE; 3,RED}
}
```

Fig. 3. The structure after performing LUP on the formula in $T2$ using $S2$. The interpretation of *node*, *start*/1, *edge*/2 and *uedge*/2 remains the same as in $S2$.

with all colors. Also note that $S4$ specifies that $reach(5, RED)$ is false because there is no edge from a `RED` reachable node to 5.

```
structure S4 : V {
  reach<ct> = { 1,BLUE; 1,RED; 2,RED; 3,BLUE }
  reach<cf> = { 2,BLUE; 3,RED; 5,RED }
}
```

Fig. 4. The structure after LUP on the completion of the *reach*/2 definition using $S3$. The interpretation of all other symbols remains the same as in $S3$

Structure $S5$ in Figure 5 shows the result after executing definition refinement. Structure $S5$ is more refined than structure $S4$, since it specifies the atoms (6,`RED`) and (6,`BLUE`) to be `false`, whereas these atoms are `unknown` in structure $S4$. These atoms can be derived to be false because they form an *unfounded set* under the Well-Founded Semantics [14]. An unfounded set is a set of atoms that only have a rule making them true that depends on themselves. The definition, which is shown below for $y = 6$ and $x = 6$, illustrates that the above derived atoms are an unfounded set. The first rule of the definition is not applicable since $start(6)$ is `false`. The second rule shows that the truth value of

```
structure S5 : V {
  reach<ct> = { 1,BLUE; 1,RED; 2,RED; 3,BLUE }
  reach<cf> = { 2,BLUE; 3,RED; 5,RED; 6,RED; 6,BLUE }
}
```

Fig. 5. The structure after definition refinement on the *reach*/2 definition using *S*3. The interpretation of all other symbols remains the same as in *S*3

$reach(6, c)$ depends on $reach(6, c)$ itself.

$$\left\{ \begin{array}{l} reach(6, c) \leftarrow start(6). \\ reach(6, c) \leftarrow reach(6, c) \wedge uedge(6, 6) \wedge color(6, c). \end{array} \right\}$$

This concludes our example of the different ways of performing lifted propagation to refine the input structure. Our next section presents the changes we had to make to our original approach for evaluation definition to extend it for definition refinement. Section 4.3 contains the complete interaction between the XSB interface and IDP^3 that is needed to perform the above definition refinement for the *reach*/2 definition.

4 Updating the XSB interface

Our new method differs only in a few ways from our original technique's usage of XSB [10]. The transformation of the inductive definitions to an XSB program does not need to change. Here we discuss the necessary extensions:

- Provide support for translating the interpretation of symbols that are not completely two-valued to XSB.
- Update the interaction with XSB to also query the possible `unknown` answers for the queried definition.

4.1 Translating unknown opens

Open symbols can now have an interpretation for which the union of the *certainly true* and *certainly false* tables does not contain all elements. Therefore we need to provide a translation for the *unknown* elements in the interpretation of an open symbol. We illustrate this using the following example: $q(x)$ is an open symbol and the type of x ranges from 1 to 5. Say $q(x)$ is known to be `true` for $\{1, 2\}$ and known to be `false` for $\{4, 5\}$. As a result, the truth value for $q(3)$ is not known. The open symbol $q(x)$ for the above interpretation will be represented in XSB as follows, given that `xsb_q(X)` is the corresponding symbol present in the XSB program for $q(x)$:

```
:- table xsb_q/1.
xsb_q(1).
xsb_q(2).
xsb_q(3) :- undef

:- table undef/0.
undef :- tnot(undef).
```

Calling `xsb_q(X)` results in `X = 1`, `X = 2`, and `X = 3`, with `X = 3` being annotated as "`undefined`". This is because XSB detects the loop over negation for `X = 3`. Note the use of `tnot/1` instead of the regular `not/1` to express negation. This is because `tnot/1` expresses the negation under the Well-Founded Semantics for tabled predicates, whereas `not/1` expresses Prolog's negation by failure.

4.2 Updating the interaction with XSB

We explain the change in interaction using an example. Say we are processing a definition that defines symbol $p(x)$. Let `xsb_p(X)` be the corresponding symbol present in the XSB program. The original interaction between XSB and IDP^3 [10] queries XSB with

```
:- call_tv(xsb_p(X), true).
```

which computes all values of `X` for which `xsb_p(X)` is `true` and retrieves the table of results, which we shall call t_t. Next, we change the interpretation of $p(x)$ in the partial structure into a two-valued one in which the atoms in the table t_t are `true` and all the others are `false`.

The new XSB interface uses the same query as above and additionally queries XSB with

```
:- call_tv(xsb_p(X), undefined).
```

which computes all values of `X` for which `xsb_p(X)` is annotated as `undefined` and retrieves the table of results, which we shall call t_u. Next, we change the interpretation of $p(x)$ in the partial structure into a three-valued one in which the atoms in the table t_t are `true`, the atoms in table t_u are `unknown` and all the others are `false`.

4.3 Example of a complete run

This section give a complete overview of all the actions for performing definition refinement on the *reach*/2 definition from Section 3. First, the definition is translated into an XSB program:

```
:- set_prolog_flag(unknown, fail).
:- table xsb_reach/2.
xsb_reach(X,C) :- xsb_start(X), xsb_color_type(C).
xsb_reach(Y,C) :- xsb_reach(X,C), xsb_uedge(X,Y), xsb_color(Y,C).
```

And the structure is also translated into a corresponding XSB program:

```
xsb_start(1).

xsb_color_type(xsb_RED).
xsb_color_type(xsb_BLUE).

xsb_uedge(1,2).
xsb_uedge(1,3).
xsb_uedge(2,1).
xsb_uedge(2,4).
xsb_uedge(3,1).
xsb_uedge(3,4).
xsb_uedge(3,5).
xsb_uedge(4,2).
xsb_uedge(4,3).
xsb_uedge(5,3).
xsb_uedge(6,6).

:- table xsb_color/2.
xsb_color(1,xsb_RED) :- undef.
xsb_color(1,xsb_BLUE) :- undef.
xsb_color(2,xsb_RED).
xsb_color(3,xsb_BLUE).
xsb_color(4,xsb_RED) :- undef.
xsb_color(4,xsb_BLUE) :- undef.
xsb_color(5,xsb_RED) :- undef.
xsb_color(5,xsb_BLUE) :- undef.
xsb_color(6,xsb_RED) :- undef.
xsb_color(6,xsb_BLUE) :- undef.

:- table undef/0.
undef :- tnot(undef).
```

These two programs are then loaded, along with some utility predicates. Next, we query XSB with the following queries:

```
| ?- call_tv(xsb_reach(X,Y),true).
X = 3, Y = xsb_BLUE;
X = 2, Y = xsb_RED;
X = 1, Y = xsb_BLUE;
X = 1, Y = xsb_RED;
no
| ?- call_tv(xsb_reach(X,Y),undefined).
X = 5, Y = xsb_BLUE;
X = 4, Y = xsb_BLUE;
X = 4, Y = xsb_RED;
```

```
no
```

As a final step, the interpretation for *reach*/2 is changed so that it is `true` for $\{(3, BLUE)\ (2, RED)\ (1, BLUE)\ (1, RED)\}$ and that it is `unknown` for $\{(5, BLUE)\ (4, BLUE)\ (4, RED)\}$, and `false` for everything else. This is depicted in Figure 5.

5 Lifted Propagation

The previous section explains how we construct an interface to XSB to retrieve a refined interpretation for the defined symbols in a single definition. Algorithm 1 shows an algorithm that uses this XSB interface to refine a structure as much as possible when there are multiple definitions in a theory. For the scope of this algorithm, the XSB interface is called as it if were a subroutine (called XSB-INTERFACE). We maintain the set of definitions that need to be processed as *Set*. Initially, *Set* contains all definitions and until *Set* is empty, we take one definition from it and process it using the XSB interface. The most important aspect of the presented algorithm is in line 12, where definitions that may have been processed before, but have an open symbol that was "updated" by processing another definition, are put back into *Set* to be processed again.

```
   input  : A structure $S$ and a set $\Delta$ of definitions in theory $T$
   output: A new structure $S'$ that refines $S$ as much as possible using $\Delta$
 1 $Set \leftarrow \Delta$
 2 while $Set$ is not empty do
 3     $\delta \leftarrow$ an element from $Set$
 4     XSB-INTERFACE ($\delta$,$S$)
 5     if inconsistency is detected then
 6         return an inconsistent structure
 7     end
 8     Insert the new interpretation for the defined symbols in $S$
 9     $\Sigma \leftarrow$ The symbols for which the interpretation has changed
10     for $\delta'$ in $\Delta$ do
11         if $\delta'$ has one of $\Sigma$ in its opens then
12             add $\delta'$ to $Set$
13         end
14     end
15     remove $\delta$ from $Set$
16 end
```

Algorithm 1: Lifted Propagation for multiple definitions

On line 5 we need to detect when an inconsistency arises from processing a definition. On line 9 we retrieve all symbols for which the interpretation has changed by processing definition δ. Since these features were not mentioned in the previous section we shortly explain here how these can be achieved. When the XSB interface processes a definition (say, XSB-INTERFACE (δ,S) is called), it does not use the interpretation of the defined symbols in δ in S for any of its calculations. We use I_σ to denote the interpretation of defined symbol σ in structure S. XSB calculates a "new" interpretation for every defined symbol σ in δ, which we will call I'_σ. If the number of `true` or the number of `false` atoms in I_σ and I'_σ differ, XSB has changed the interpretation of symbol σ and this symbol will be present in Σ as displayed in line 9. If there is an atom that is `true` in I_σ and and `false` in I'_σ, or vice versa, there is inconsistency and the check on line 5 will succeed.

A possible point of improvement for this algorithm is the selection done in line 3. One could perform a dependency analysis and stratify the definitions that have to be refined. In this way, the amount of times each definition is "processed" is minimized. This stratification is ongoing work.

A worst case performance for the proposed algorithm is achieved when there are two definitions that depend on each other, as given in the following example:

$$\left\{\begin{array}{l} P(0). \\ P(x) \leftarrow Q(x-1). \end{array}\right\}$$
$$\left\{ Q(x) \leftarrow P(x-1). \right\}$$

If we start with processing the $P/1$ definition, we derive $P(0)$. Processing $Q/1$ then leads to deriving $Q(1)$. Since the interpretation of $Q/1$ changed and it is an open symbol of the $P/1$ definition, the $P/1$ definition has to be processed again. This continues for as many iterations as there are elements in the type of x. Since every call to the XSB interface for a definition incurs inter-process overhead, this leads to a poor performance. This problem can be alleviated by detecting that the definitions can safely be joined together into a single definition. The detection of joining definition to improve the performance of the proposed algorithm is part of future work.

6 Experimental evalutation

In this section we evaluate our new method of refining definitions by comparing it to its alternative: performing Lifted Unit Propagation (LUP) on the completion of the definitions. We will refer to our new method for Definition Refinement as "the DR approach". In the IDP^3 system, there are two ways of performing LUP on a structure: using an Approximating Definition (AD) [4] or using Binary Decision Diagrams (BDD) [15]. We will refer to these methods as "the AD approach" for the former and "the BDD approach" for the latter. The AD approach expresses the possible propagation on SAT level using an IDP^3 definition. This definition is evaluated to derive new `true` and `false` bounds for the structure. Note that this approximating definition is entirely different from any other possible definitions originally present in the theory. The BDD approach works by creating Binary Decision Diagrams that represent the formulas (in this case the formulas for the completion of the definitions) in the theory. It works symbolically and is approximative: it will not always derive the best possible refinement of the structure. The AD approach on the other hand is not approximative.

Table 1 shows our experiment results for 39 problems taken from past ASP competitions. For each problem we evaluate our method on 10 or 13 instances. The problem instances are evaluated with a timeout of 300 seconds. We present the following information for each problem, for the DR approach:

– s^{DR} The number of runs that succeeded
– t^{DR}_{avg} The average running time (in seconds)
– t^{DR}_{max} The highest running time (in seconds)
– a^{DR}_{avg} The average number of derived atoms

The same information is also given for the BDD and the AD approach, with the exception that a^{BDD}_{avg} and a^{AD}_{avg} only take into account runs that also succeeded for the DR approach. This allows us to compare the number of derived atoms, since it can depend strongly on the instance of the problem that is run.

Comparing the DR approach with the AD approach, one can see that the DR approach is clearly better. The AD approach fails to refine the structure for even a single problem instance for 20 out of the 39 problems. When both the DR and the AD approaches do succeed, AD derives as much information as the DR approach. One can conclude from this that there is no benefit to using the AD approach over the DR approach. Moreover, the DR approach is faster in most of the cases.

Comparing the DR approach with the BDD approach is less straightforward. The BDD approach has a faster average and maximum running time for each of the problems. Additionally, for 11 out of the 39 problems the BDD approach had more problem instances that did not reach a timeout. These problems are indicated in **bold** in the s^{BDD} column. For some problems the difference is small, as for example for the *ChannelRouting* problem where average running times are respectively 7.16 and 5.67. For other problems however, the difference is very large, as for example for the *PackingProblem* problem where average running times are respectively 199.53 with 7 timeouts and 0.1 with 0 timeouts. Although the BDD approach is clearly faster, there is an advantage to the DR approach: for 8 problems, it derives extra information compared to the BDD approach. These instances are indicated in **bold** in the a^{DR}_{avg} column. This difference is sometimes small (80 vs. 8 for SokobanDecision) and sometimes large (25119 vs. 0 for Tangram). This shows that there is an advantage to using our newly proposed DR approach.

There is one outlier, namely the NoMystery problem in which more information is derived with the BDD approach than by the DR approach. This is because DR does lifted propagation in the “direction” of the definition: for the known information of the body of a rule, try to derive more information about the defined symbol. However, sometimes it is possible to derive information about elements in the body rules using information that is known about the head of the rule. Since LUP performs propagation along both directions and DR only along the first one, it is possible that LUP derives more information. As one can see in the experiment results, it is only on very rare occasions (1 problem out of 39) where this extra direction makes a difference. Integrating this other direction of propagation into the DR approach is ongoing work.

	Definition Refiniement				Binary Decision Diagrams				Approximating Definition			
Problem Name	s^{DR}	t^{DR}_{avg}	t^{DR}_{max}	a^{DR}_{avg}	s^{BDD}	t^{BDD}_{avg}	t^{BDD}_{max}	a^{BDD}_{avg}	s^{AD}	t^{AD}_{avg}	t^{AD}_{max}	a^{AD}_{avg}
15Puzzle	10/10	5.3	6.24	**482**	10/10	0.07	0.08	256	0/10	-	0	-
BlockedNQueens	10/10	7.16	12.56	0	10/10	5.67	10.44	0	10/10	5.3	10.01	0
ChannelRouting	10/10	5.62	7.28	0	10/10	4.58	6.1	0	10/10	4.23	5.35	0
ConnectedDominatingSet	10/10	0.39	1.13	0	10/10	0.01	0.02	0	7/10	55.79	129.04	0
EdgeMatching	10/10	4.46	8.35	0	10/10	0.28	0.37	0	1/10	2.68	2.68	0
GraphPartitioning	13/13	0.27	0.48	0	13/13	0.02	0.02	0	13/13	14.34	48.73	0
HamiltonianPath	10/10	1.43	2.12	1	10/10	0.02	0.02	1	0/10	-	0	-
HierarchicalClustering	10/10	9.11	89.86	0	10/10	6	58.82	0	10/10	6.42	63.24	0
MazeGeneration	10/10	3.2	7.08	1	10/10	2.38	4.63	1	2/10	65.27	65.84	1
SchurNumbers	10/10	0.01	0.01	0	10/10	0	0.01	0	10/10	0.03	0.04	0
TravellingSalesperson	10/10	0.51	0.71	73	10/10	0.05	0.06	73	10/10	0.72	0.84	73
WeightBoundedDominatingSet	10/10	0.03	0.04	0	10/10	0.02	0.03	0	10/10	0.03	0.05	0
WireRouting	10/10	1.25	2.26	7	10/10	0.12	0.18	7	0/10	-	0	-
GeneralizedSlitherlink	0/10	-	0	-	**10/10**	0.13	0.28	-	0/10	-	0	-
FastfoodOptimalityCheck	1/10	18.43	18.43	36072	**10/10**	2.49	3.77	36072	0/10	-	0	-
SokobanDecision	10/10	58.61	114.92	**80**	10/10	0.11	0.14	8	0/10	-	0	-
KnightTour	6/10	9.03	31.9	0	**10/10**	1.14	4.07	0	8/10	53.06	293.07	0
DisjunctiveScheduling	10/10	4.49	10.67	0	10/10	2.69	5.74	0	10/10	3.52	9.67	0
PackingProblem	3/10	199.53	243.46	17	**10/10**	0.1	0.13	17	0/10	-	0	-
Labyrinth	2/10	103.77	192.16	**105533**	**10/10**	0.62	1.2	104668	0/10	-	0	-
Numberlink	9/10	8.91	30.4	0	**10/10**	0.28	1.34	0	0/10	-	0	-
ReverseFolding	1/10	49.84	49.84	**978**	**10/10**	1	2.11	16	0/10	-	0	-
HanoiTower	10/10	28.42	56.87	**28020**	10/10	0.2	0.25	25042	0/10	-	0	-
MagicSquareSets	10/10	0.44	1.01	**1659**	10/10	0.12	0.15	0	0/10	-	0	-
AirportPickup	10/10	40.09	100.78	1267	10/10	0.16	0.28	1267	0/10	-	0	-
PartnerUnits	10/10	13.43	14.87	100	10/10	0.02	0.03	100	0/10	-	0	-
Tangram	13/13	37.84	41.33	**25119**	13/13	0.33	0.4	0	0/13	-	0	-
Permutation-Pattern-Matching	10/10	33.53	115.63	0	10/10	0.01	0.01	0	7/10	85.6	258.84	0
Graceful-Graphs	10/10	0.01	0.02	0	10/10	0.01	0.02	0	10/10	0.03	0.04	0
Bottle-filling-problem	10/10	1.43	5.09	0	10/10	0.96	3.57	0	10/10	0.81	2.68	0
NoMystery	4/10	64.82	137.36	207804	**10/10**	0.57	1.3	**207821**	0/10	-	0	-
Sokoban	6/10	86.79	196.84	**18254**	**10/10**	0.11	0.12	8	0/10	-	0	-
Ricochet-Robot	0/10	-	0	-	10/10	2.68	3.16	-	0/10	-	0	-
Weighted-Sequence-Problem	10/10	0.25	0.3	0	10/10	0.22	0.23	0	10/10	0.16	0.2	0
Incremental-Scheduling	0/10	-	0	-	**8/10**	54.34	229.59	-	0/10	-	0	-
Visit-all	3/10	3.49	3.97	15	**10/10**	0.03	0.04	15	0/10	-	0	-
Graph-Colouring	10/10	0.12	0.15	0	10/10	0.02	0.03	0	10/10	0.22	0.28	0
LatinSquares	10/10	0.02	0.02	0	10/10	0.01	0.02	0	10/10	0.03	0.04	0
Sudoku	10/10	0.2	0.3	0	10/10	0.16	0.26	0	10/10	0.14	0.23	0

Table 1. Experiment results comparing Definition Refinement (DR) with the Binary Decision Diagram (BDD) and Approximating Definition (AD) approach

Our experiments show the added value of our DR approach, but also indicate that more effort should be put into this approach towards optimising runtime.

7 Conclusion

In this paper we described an extension to our existing preprocessing step [10] for definition evaluation to be able to refine the interpretation of defined predicates in the partial structure when the predicates depend on information that is only partially given. Our method uses XSB to compute the atoms (instances of the predicate) that are `true` and others that are `unknown`. This method is an alternative for using LUP on the completion of the definitions. Because LUP for the completion of the definition is an approximation of what can be derived for that definition, our method is able to derive stricly more information for the defined symbols than the LUP alternative. The extra information that can be derived is the detection of unfounded sets for a definition. Because GWB uses the information in the structure to derive bounds during grounding, this extra derived information possibly leads to stricter bounds and an improved grounding.

Our experiments show the added value of our new method, but also indicate that it is not as robust in performance as LUP (using BDDs). This paper indicates two ways in which the performance of the proposed method might be improved:

– Perform an analysis of the dependencies of definitions and query them accordingly to minimize the number of times a definition is re-queried
– Similar to the element above, try to detect when definitions can be joined together to minimize XSB overhead

These improvements are future work. Another part of future work is combining the new method for lifted propagation for definitions with the LUP for formulas in the theory. Combining these two techniques might lead to even more derived information, since the formulas might derive information that allows the definition the perform more propagation and vice versa.

References

1. Amir Aavani, Xiongnan (Newman) Wu, Shahab Tasharrofi, Eugenia Ternovska, and David G. Mitchell. Enfragmo: A system for modelling and solving search problems with logic. In Nikolaj Bjørner and Andrei Voronkov, editors, *LPAR*, volume 7180 of *LNCS*, pages 15–22. Springer, 2012.
2. Maurice Bruynooghe, Hendrik Blockeel, Bart Bogaerts, Broes De Cat, Stef De Pooter, Joachim Jansen, Marc Denecker, Anthony Labarre, Jan Ramon, and Sicco Verwer. Predicate logic as a modeling language: Modeling and solving some machine learning and data mining problems with IDP3. *CoRR*, abs/1309.6883, 2013.
3. Broes De Cat, Bart Bogaerts, Jo Devriendt, and Marc Denecker. Model expansion in the presence of function symbols using constraint programming. In *ICTAI*, pages 1068–1075. IEEE, 2013.
4. Broes De Cat, Joachim Jansen, and Gerda Janssens. IDP3: Combining symbolic and ground reasoning for model generation. In *Workshop on Grounding and Transformations for Theories with Variables, La Coruña, 15 Sept 2013*, 2013.
5. Broes De Cat and Maarten Mariën. MiniSat(ID) website. `http://dtai.cs.kuleuven.be/krr/software/minisatid`, 2008.
6. Stef De Pooter, Johan Wittocx, and Marc Denecker. A prototype of a knowledge-based programming environment. In *International Conference on Applications of Declarative Programming and Knowledge Management*, 2011.
7. Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

8. Wolfgang Faber, Nicola Leone, and Simona Perri. The intelligent grounder of DLV. *Correct Reasoning*, pages 247–264, 2012.
9. Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. Advances in *gringo* series 3. In James P. Delgrande and Wolfgang Faber, editors, *LPNMR*, volume 6645 of *LNCS*, pages 345–351. Springer, 2011.
10. Joachim Jansen, Albert Jorissen, and Gerda Janssens. Compiling input$*$ FO$(\cdot)$ inductive definitions into tabled Prolog rules for IDP^3. *TPLP*, 13(4-5):691–704, 2013.
11. David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 430–435. AAAI Press / The MIT Press, 2005.
12. T. Swift and D.S. Warren. XSB: Extending the power of Prolog using tabling. *TPLP*, 12(1-2):157–187, 2012.
13. Terrance Swift. An engine for computing well-founded models. In Patricia M. Hill and David Scott Warren, editors, *ICLP*, volume 5649 of *LNCS*, pages 514–518. Springer, 2009.
14. Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
15. Johan Wittocx, Marc Denecker, and Maurice Bruynooghe. Constraint propagation for first-order logic and inductive definitions. *ACM Trans. Comput. Logic*, 14(3):17:1–17:45, August 2013.
16. Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *Journal of Artificial Intelligence Research*, 38:223–269, 2010.

Intelligent Visual Surveillance Logic Programming: Implementation Issues

Alexei A. Morozov[1,2] and Alexander F. Polupanov[1,2]

[1] Kotel'nikov Institute of Radio Engineering and Electronics of RAS, Mokhovaya 11, Moscow, 125009, Russia
[2] Moscow State University of Psychology & Education, Sretenka 29, Moscow, 107045, Russia
`morozov@cplire.ru,sashap55@mail.ru`

Abstract. The main idea of the logic programming approach to the intelligent video surveillance is in using a first order logic for describing complex events and abstract concepts like anomalous human activity, i.e. brawl, sudden attack, armed attack, leaving object, loitering, pick pocketing, personal theft, immobile person, etc. We consider main implementation issues of our approach to the intelligent video surveillance logic programming: object-oriented logic programming of concurrent stages of video processing, translating video surveillance logic programs to fast Java code, embedding low-level means for video storage and processing to the logic programming system.

Keywords: intelligent visual surveillance, object-oriented logic programming, concurrent logic programming, abnormal behavior detection, anomalous human activity, Actor Prolog, complex events recognition, computer vision, technical vision, Prolog to Java translation

1 Introduction

Human activity recognition is a rapidly growing research area with important application domains including security and anti-terrorist issues [1,11,12]. Recently logic programming was recognized as a promising approach for dynamic visual scenes analysis [7,24,23,14,22]. The idea of the logic programming approach is in usage of logical rules for description and analysis of people activities. Knowledge about object co-ordinates and properties, scene geometry, and human body constraints is encoded in the form of certain rules in a logic programming language and is applied to the output of low-level object / feature detectors. There are several studies based on this idea. In [7] a system was designed for recognition of so-called long-term activities (such as fighting and meeting) as temporal combinations of short-term activities (walking, running, inactive, etc.) using a logic programming implementation of the Event Calculus. The ProbLog probabilistic logic programming language was used to handle the uncertainty that occurs in human activity recognition. In [24] an extension of predicate logic with the bilattice formalism that permits processing of uncertainty in the reasoning was proposed. The VidMAP visual surveillance system that combines real time computer vision algorithms with the Prolog based logic programming had been announced by the same team. In [23] the VERSA general-purpose framework for defining and recognizing events in live or recorded surveillance video streams is described. According to [23], VERSA ensures more advanced spatial and temporal reasoning than VidMAP and is based on SWI-Prolog. In [14] a real time complex audio-video event detection based on Answer Set Programming approach is proposed. The results indicate that this solution is robust and can easily be run on a chip.

The distinctive feature of our approach to the visual surveillance logic programming is in application of general-purpose concurrent object-oriented logic programming features to it, but not in the development of a new logical formalism. We use the Actor Prolog object-oriented logic language [15,16,17,19,18] for implementation of concurrent stages of video processing. A state-of-the-art Prolog-to-Java translator is used for efficient implementation of logical inference on video scenes. Special built-in classes of the Actor Prolog language were developed and implemented for the low-level video storage and processing.

Basic principles of video surveillance logic programming are described in Section 2. The concurrent logic programming issues linked with the video processing are described in Section 3. The problems of efficient implementation of logical inference on video scenes and translation of logic programs to fast Java code are discussed in Section 4. The problems of video storage and low-level processing are considered in Section 5.

2 Basic principles of video surveillance logic programming

A common approach to human activity recognition includes low-level and high-level stages of the video processing. An implementation of the logic programming approach requires consideration of different mathematical and engineering problems on each level of recognition:

1. Low-level processing stages usually include recognition of moving objects, pedestrians, faces, heads, guns, etc. The output of the low-level procedures is an input for the high-level (semantic) analyzing procedures. So, the low-level procedures should be fast enough to be used for real time processing and the high-level logical means should utilize effectively and completely the output of the low-level procedures.
2. Adequate high-level logical means should be created / used to deal with temporal and spatial relationships in the video scene, as well as uncertainty in the results of low-level recognition procedures.
3. A proper hierarchy of the low-level and high-level video processing procedures should be constructed. The procedures of different levels are usually implemented on basis of different programming languages.
4. Selection of logic programming system for implementation of the logical inference is of great importance, because it should provide high-level syntax

means for implementation of required logical expressions as well as computation speed and efficiency in real time processing of big amount of data.

Let us consider an example of logical inference on video. The input of a logic program written in Actor Prolog is the *Fight_OneManDown* standard sample provided by the CAVIAR team [8]. The program will use no additional information about the content of the video scene, but only co-ordinates of 4 defining points in the ground plane (the points are provided by CAVIAR), that are necessary for estimation of physical distances in the scene.

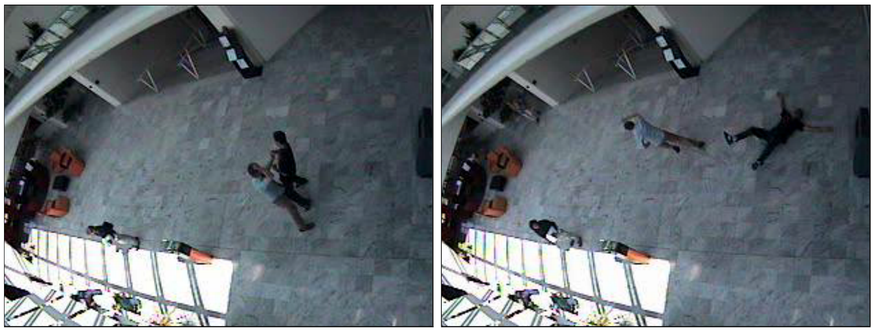


Fig. 1. An example of CAVIAR video with a case of a street offence: one person attacks another.

The video (see Fig. 1) demonstrates a case of a street offence—a probable conflict between two persons. These people meet in the scope of the video camera, then one person attacks another one, the second person falls, and the first one runs away. This incident could be easily recognized by a human; however an attempt to recognize it automatically brings to light a set of interesting problems in the area of pattern recognition and video analysis.

First of all, note that probably the main evidence of an anomalous human activity in this video is so-called abrupt motion of the persons. Abrupt motions can be easily recognized by a human as motions of a person's body and / or arms and legs with abnormally high speed / acceleration. So, a logic programmer has a temptation to describe an anomalous human activity in terms of abrupt motions, somehow like this: "Several persons have met sometime and somewhere. After that they perform abrupt motions. This is probably a case of a street fighting." It is not a problem to implement this definition in Prolog using a set of logical rules, however real experiments with video samples show that this naive approach is an impractical one or simply does not work. The problem is that in the general case computer low-level procedures recognize abrupt motions much worse than a human and there are several serious reasons for this:

1. Generally speaking, it is very difficult to determine even the exact co-ordinates of a person in a video scene. A common approach to the problem is

usage of so-called ground plane assumption, that is, the computer determines co-ordinates of body parts that are situated inside a pre-defined plane and this pre-defined plane usually is a ground one. So, one can estimate properly the co-ordinates of person's shoes, but a complex surface of a ground and / or presence of stairs and other objects, etc. make the problem much more complex.
2. Even computing the first derivative of moving person's co-ordinates is a problem usually, because the silhouette of the person changes unpredictably in different lighting conditions and can be partially overlapped by other objects. As a consequence, the trajectory of a person contains a big amount of false co-ordinates that makes numerical differentiation senseless.
3. One can make abrupt motions even standing still. This means that in the general case the program should recognize separate parts of person's body to determine abrupt motions in a robust and accurate way.

All these issues relate to the low-level video processing and probably are not to be implemented in a logic language. Nevertheless, they illustrate a close connection between the principles to be used for logical description of anomalous human activity and the output of low-level video processing procedures. We take into account this connection in our research, when we address the problem of the high-level semantic analysis of people activities.

In the example under consideration, we will solve the problem of anomalous human activity recognition using automatic low-level algorithms that trace persons in video scene and estimate average velocity in different segments of the trajectories [22]. This low-level processing includes extraction of foreground blobs, tracking of the blobs over time, detection of interactions between the blobs, creation of connected graphs of linked tracks of blobs, and estimation of average velocity of blobs in separate segments of tracks. This information is received by the logic program in a form of Prolog terms describing the list of connected graphs. We will use the following data structures for describing connected graphs of tracks[3]:

```
DOMAINS:
ConnectedGraph = ConnectedGraphEdge*.
ConnectedGraphEdge = {
     frame1: INTEGER,
     x1: INTEGER, y1: INTEGER,
     frame2: INTEGER,
     x2: INTEGER, y2: INTEGER,
     inputs: EdgeNumbers,
     outputs: EdgeNumbers,
     identifier: INTEGER,
     coordinates: TrackOfBlob,
```

[3] Note, that the DOMAINS, the PREDICATES, and the CLAUSES program sections in Actor Prolog have traditional semantics developed in the Turbo / PDC Prolog system.

```
     mean_velocity: REAL
     }.
EdgeNumbers = EdgeNumber*.
EdgeNumber  = INTEGER.
TrackOfBlob = BlobCoordinates*.
BlobCoordinates = {
     frame: INTEGER,
     x: INTEGER, y: INTEGER,
     width: INTEGER, height: INTEGER,
     velocity: REAL
     }.
```

That is, connected graph of tracks is a list of underdetermined sets [15] denoting separate edges of the graph. The nodes of the graph correspond to points where tracks cross, and the edges are pieces of tracks between such points. Every edge is directed and has the following attributes: numbers of first and last frames ($frame1$, $frame2$), co-ordinates of first and last points ($x1$, $y1$, $x2$, $y2$), a list of edge numbers that are direct predecessors of the edge ($inputs$), a list of edge numbers that are direct followers of the edge ($outputs$), the identifier of corresponding blob (an integer $identifier$), a list of sets describing the co-ordinates and the velocity of the blob in different moments of time ($coordinates$), and an average velocity of the blob in this edge of the graph ($mean\_velocity$).

The logic program will check the graph of tracks and look for the following pattern of interaction among several persons: "If two or more persons met somewhere in the scene and one of them has run after the end of the meeting, the program should consider this scenario as a kind of a running away and a probable case of a sudden attack or a theft." So, the program will alarm if this kind of sub-graph is detected in the total connected graph of tracks. In this case, the program marks all persons in the inspected graph by yellow rectangles and outputs the "Attention!" warning in the middle of the screen (see Fig. 2).

One can describe the concept of a running away formally using defined connected graph data type:

```
PREDICATES:
is_a_kind_of_a_running_away(
     ConnectedGraph,
     ConnectedGraph,
     ConnectedGraphEdge,
     ConnectedGraphEdge,
     ConnectedGraphEdge) - (i,i,o,o,o);
```

We will define the $is\_a\_kind\_of\_a\_running\_away(G, G, P1, E, P2)$ predicate with the following arguments: G—a graph to be analyzed (note that the same data structure is used in the first and the second arguments), E—an edge of the graph corresponding to a probable incident, $P1$—an edge of the graph that is a predecessor of E, $P2$—an edge that is a follower of E. Note that G is an input

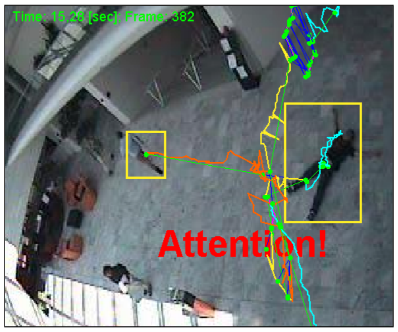


Fig. 2. A logical inference has found a possible case of a sudden attack in the graph of blob trajectories. All probable participants of the conflict are marked by yellow rectangles. The tracks are depicted by lines.

argument of the predicate and $P1$, E, and $P2$ are output ones. Here is an Actor Prolog program code with brief explanations:

```
CLAUSES:
is_a_kind_of_a_running_away([E2|_],G,E1,E2,E3):-
     E2 == {inputs:O,outputs:B|_},
     B == [_,_|_],
     contains_a_running_person(B,G,E3),
     is_a_meeting(O,G,E2,E1),!.
is_a_kind_of_a_running_away([_|R],G,E1,E2,E3):-
     is_a_kind_of_a_running_away(R,G,E1,E2,E3).
contains_a_running_person([N|_],G,P):-
     get_edge(N,G,E),
     is_a_running_person(E,G,P),!.
contains_a_running_person([_|R],G,P):-
     contains_a_running_person(R,G,P).
is_a_meeting(O,_,E,E):-
     O == [_,_|_],!.
is_a_meeting([N1|_],G,_,E2):-
     get_edge(N1,G,E1),
     E1 == {inputs:O|_},
     is_a_meeting(O,G,E1,E2).
get_edge(1,[Edge|_],Edge):-!.
get_edge(N,[_|Rest],Edge):-
     N > 0,
     get_edge(N-1,Rest,Edge).
```

In other words, the graph contains a case of a running away if there is an edge $E2$ in the graph that has a follower $E3$ corresponding to a running person and predecessor $E1$ that corresponds to a meeting of two or more persons. It is requested also that $E2$ has two or more direct followers (it is a case of branching in the graph). Note, that in the Actor Prolog language, the == operator corresponds to the = ordinary equality of the standard Prolog.

A fuzzy definition of the running person concept is as follows:

```
is_a_running_person(E,_,E):-
     E == {mean_velocity:V,frame1:T1,frame2:T2|_},
     M1== ?fuzzy_metrics(V,1.0,0.5),
     D== (T2 - T1) / sampling_rate,
     M2== ?fuzzy_metrics(D,0.75,0.5),
     M1 * M2 >= 0.5,!.
is_a_running_person(E,G,P):-
     E == {outputs:B|_},
     contains_a_running_person(B,G,P).
```

The graph edge corresponds to a running person if the average velocity and the length of the track segment correspond to the fuzzy definition. Note that Actor Prolog implements a non-standard functional notation, namely, the ? prefix informs the compiler that the *fuzzy_metrics* term is a call of a function, but not a data structure.

An auxiliary function that calculates the value of the fuzzy metrics is represented below. The first argument of the function is a value to be checked, the second argument is a value of a fuzzy threshold, and the third one is the width of the threshold ambiguity area. The = delimiter defines an extra output argument that is a result to be returned by the function:

```
fuzzy_metrics(X,T,H) = 1.0 :-
     X >= T + H,!.
fuzzy_metrics(X,T,H) = 0.0 :-
     X <= T - H,!.
fuzzy_metrics(X,T,H) = V :-
     V== (X-T+H) * (1 / (2*H)).
```

Note that 37 lines of the Actor Prolog code considered above correspond to 301 lines of the optimized Java source code implementing graph search operations and this is perhaps the best demonstration of the reasons why the Prolog language is a good choice for this application.

This example illustrates the basic principles of logical description of anomalous human activity and logical inference on video data. However, even a simplest scheme of video surveillance logic program should contain much more elements, including video information gathering, low-level image analysis, high-level logical inference control, and reporting the results of intelligent visual surveillance.

3 Concurrent video processing

It is a good idea to divide a visual surveillance logic program to concurrent subprocesses implementing different stages of video processing, because the working intensity of different sub-processes is various. For instance, video data gathering and low-level analysis require a big amount of computational resources and other sub-processes that implement high-level analysis and visualizing results of video surveillance could wait for the output of the former sub-process.

In the example under consideration, we will create two concurrent processes with different priorities[4]. The first process has higher priority and implements video data gathering. This process reads JPEG files and sends them to the instance of the *'ImageSubtractor'* predefined class that implements all low-level processing of video frames. The sampling rate of the video is 25 frames per second, so the process loads a new JPEG file every 40 milliseconds. The second concurrent process implements logical analysis of collected information and outputs results of the analysis. The analysis of video frames requires more computational resources, but it does not suspend the low-level analysis, because the second process has less priority. The analysis includes creation of connected graphs of linked tracks of blobs and estimation of average velocity of blobs in separate segments of tracks. This information is received by the logic program in a form of connected graphs described in the previous section.

The total text of the logic program is not given here for brevity; we will discuss only the structure of the main class of the program.

```
class 'Main' (specialized 'Alpha'):
constant:
     data_directory   = "data";
     target_directory = "Fight_OneManDown";
     sampling_rate    = 25.0;
     stage_one = (('ImagePreprocessor',
               data_directory,
               target_directory,
               sampling_rate,
               low_level_analyzer,
               stage_two));
     stage_two = (('ImageAnalyzer',
               low_level_analyzer,
               sampling_rate));
internal:
     low_level_analyzer = ('ImageSubtractor',
               extract_blobs= 'yes',
               track_blobs= 'yes',
               ...);
```

The *'Main'* class has the *stage_one* and the *stage_two* slots[5] containing two above mentioned processes. The *low_level_analyzer* slot contains an instance of the *'ImageSubtractor'* built-in class implementing low-level video analysis procedures (see Fig. 3). The *data_directory*, the *target_directory*, and the

[4] See [17] for details of Actor Prolog model of asynchronous concurrent computations.

[5] The slot is a synonym for the instance variable in the Actor Prolog language.

sampling_rate auxiliary slots contain information about source data files location and the sampling rate of the video clip.

The idea of the schema is the following: the instance of the *'ImageSubtractor'* built-in class is used as a container for storing video data and intermediate results of the low-level processing. The instances of the *'ImagePreprocessor'* and the *'ImageAnalyzer'* classes accept the *low_level_analyzer* slot as an argument and use it for the inter-process data exchange. So, the object-oriented features give us an essential solution of the problem of big data storage in the logic language; all low-level video data is encapsulated in the special class and the logic program handles medium-size terms describing the results of the low-level analysis only.

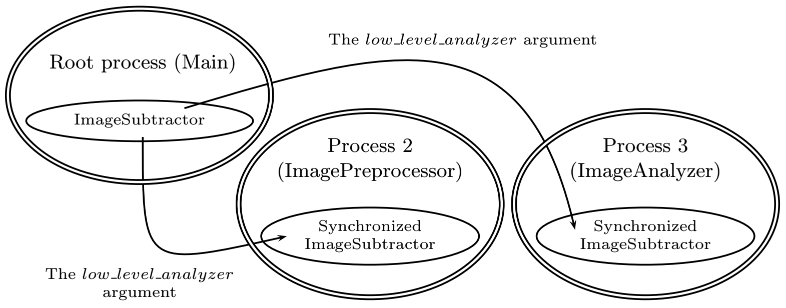

The *low_level_analyzer* argument
Root process (Main)
ImageSubtractor
Process 2 (ImagePreprocessor)
Synchronized ImageSubtractor
Process 3 (ImageAnalyzer)
Synchronized ImageSubtractor
The *low_level_analyzer* argument

Fig. 3. The *'SynchronizedImageSubtractor'* class implements synchronization of three concurrent processes and ensures a safe access to the internal data arrays of the *'ImageSubtractor'* class instance.

The *'ImagePreprocessor'* class implements video data gathering and all low-level processing of video frames. The *'ImageAnalyzer'* class contains the rules considered in the previous section, implements high-level stage of video analysis, and outputs the results of intelligent video surveillance.

Note, that Actor Prolog prohibits a process from invoking a predicate from a class instance belonging to another process. This means that the sharing of the *'ImageSubtractor'* class instance mentioned above requires an additional care. In detail, the *'ImageAnalyzer'* class uses the *'SynchronizedImageSubtractor'* built-in class as an envelope for the *'ImageSubtractor'* class instance.

```
class 'ImageAnalyzer' (specialized 'Alpha'):
constant:
     sampling_rate;
     low_level_analyzer;
internal:
     subtractor = ('SynchronizedImageSubtractor',
               image_subtractor= low_level_analyzer);
...
```

The $'SynchronizedImageSubtractor'$ built-in class implements the same methods of low-level video processing as the $'ImageSubtractor'$ class. The only difference is that the instance of the $'SynchronizedImageSubtractor'$ class is created inside the $'ImageAnalyzer'$ process and there are no restrictions on the usage of its methods inside this process (see Fig. 3).

Thus, the $'SynchronizedImageSubtractor'$ class ensures a safe access to the internal data arrays of the $'ImageSubtractor'$ class from concurrent processes and implements all necessary operations on video data. The data access synchronization is implemented inside the built-in classes, but not at the level of the programming language; Actor Prolog is asynchronous concurrent language, it has no syntactical means for concurrent processes synchronization and supports syntactically only the asynchronous inter-process communications.

4 Prolog to Java translation

A high CPU / memory consumption is a characteristic of the video processing. So, the visual surveillance application is a good way to test whether the logic programming system is mature enough to be used in the industry. One could enumerate the following requirements for a logic programming system / language selected as a basis for the visual surveillance application:

1. Firstly, the logic programming system should generate a fast executable code. A deep code optimization is absolutely necessary even for the logic programs that use 2D graphic intensively for reporting intelligent visual surveillance results.
2. The executable code should be robust; absence of any memory leak should be guaranteed. An incorrect memory handling and / or incorrect concurrent access to data structures produce an unavoidable crash of the program.
3. The logic programming system should be an open one; the extension of the system by specialized classes / procedures implementing low-level processing should be easy and transparent for application programmers.

These requirements are contradictory because considerable code optimization implies usage of complex compilation algorithms and low-level code generation that are potential reasons for difficult-to-locate errors, memory leaks, and unstable operation of executable code. Even if the compiler is well-debugged the permanent development of built-in classes and libraries is a constant potential source of such errors. There is a fundamental contradiction between the openness and the optimization of the programming system.

In particular, application of a compilation schema based on C / C++ intermediate code generation (Mercury [10], KLIC [9], wamcc [3]) was recognized as an appropriate way to obtain maximal speed of the executable code. On the other hand, generation of Java intermediate code (Actor Prolog [20], PrologCafe [2], KLIJava [13], SAE-Prolog [6], jProlog [5]) ensures platform independence of the application software and guarantees absence of difficult-to-locate errors caused by memory leaks and out-of-range array operations. We use a compilation from the Actor Prolog language to Java, because, from our point of view, modern processors are fast enough to give up the speed of the executable code for the sake of robustness, readability, and openness of the logic program. Moreover, using an industrial Java virtual machine as a basis for the logic programming system ensures its flexibility and quick adaptation to new operational systems and processor architectures.

In contrast to conventional approaches, we use neither WAM (PrologCafe, wamcc) nor binarization of the logic program (jProlog, BinProlog [25]). The Actor Prolog compiler generates a kind of an idiomatic (i.e., well-readable) source code (SAE-Prolog, P# [4]), but in contrast to the SAE-Prolog project [6] we use domains / predicates declarations to process non-deterministic, deterministic, and imperative predicates in different ways. In contrast to the P# project [4] we implement non-idiomatic predicate calls from idiomatic predicates and vice versa.

The Actor Prolog logic language differs from the Clocksin&Mellish Prolog a lot. Turbo-Prolog style Domain and Predicate declarations of Actor Prolog are very important for the industrial application programming and help in executable code optimization. On the other hand, object-oriented features and supporting concurrent programming make translation of an Actor Prolog code to be a non-trivial problem.

The state-of-the-art compilation schema of the Actor Prolog system includes the following steps:

1. *Source text scanning and parsing.* Methods of thinking translation preventing unnecessary processing of already translated source files are implemented. That is, after the update of source codes, the compiler tries to use information collected / computed during its previous run. This feature is very important for the industrial programming.
2. *Inter-class links analysis.* On this stage of global analysis, the translator collects information about usage of separate classes in the program, including data types of arguments of all class instance constructors. This information is necessary for the global flow analysis and the global optimization of the program (unused predicates are eliminated from the program).
3. *Type check.* The translator checks data types of all predicate arguments and arguments of all class instance constructors.
4. *Determinism check.* The translator checks whether predicates are deterministic or non-deterministic. A special kind of so-called imperative predicates is supported, that is, the compiler can check whether a predicate is deterministic and never fails.
5. *A global flow analysis.* The compiler tracks flow patterns of all predicates in all classes of the program.
6. *Generation of an intermediate Java code.*
7. *Translation of this Java code by a standard Java compiler.*

The determinism check ensures a possibility to use different optimization methods for different kinds of predicates:

1. The imperative predicates check is the most complex stage in the translation schema, because it requires a check of all separate clauses as well as mutual influence of the clauses / predicates. Nevertheless, this check is of critical importance, because the imperative predicates usually constitute the main part of the program and the check gives information for very high level optimization of these predicates—the logic language clauses are translated to Java procedures directly.
2. Deterministic predicates are translated to Java procedures too (all clauses of one predicate correspond to one Java procedure). Backtracking is implemented using a special kind of light-weight Java exceptions.
3. Non-deterministic predicates are implemented using a standard method of continuation passing. Clauses of one predicate correspond to one or several automatically generated Java classes.

Tail recursion optimization is implemented for recursive predicates; that is critically important for the video processing applications. Recursive predicates are implemented using the *while* Java command. Moreover, the Actor Prolog language supports explicit definition of ground / non-ground domains and the translator uses this information for deep optimization of ground term unification.

Table 1. Prolog benchmark testing (Intel Core i5-2410M, 2.30 GHz, Win7, 64-bit)

Test	Iter. No.	Actor Prolog to Java	SWI-Prolog v. 7.1.10
NREV	3,000,000	109,090,909 lips	15,873,523 lips
CRYPT	100,000	1.758510 ms	2.03347 ms
DERIV	10,000,000	0.055747 ms	0.0104318 ms
POLY_10	10,000	3.756900 ms	4.3681 ms
PRIMES	100,000	0.042540 ms	0.13478 ms
QSORT	1,000,000	0.042924 ms	0.059561 ms
QUEENS(9)	10,000	17.495200 ms	31.729 ms
QUERY	10,000	3.141500 ms	0.3713 ms
TAK	10,000	4.010800 ms	10.2836 ms

The described compilation schema ensures an acceptable performance of the executable code (see Table. 1). Deterministic and imperative predicates with ground arguments are optimized quite well (the NREV test demonstrates more than 100 millions lips). At the same time, non-deterministic predicates work slowly (CRYPT, QUEENS, QUERY); this is a fundamental disadvantage of the approach based on continuation passing and translation to the high-level intermediate language, because it cannot handle possible run-time optimization of Prolog stacks. Arithmetical predicates work fast enough in Actor Prolog (PRIMES, QSORT, TAK), but there is a possibility for better optimization of symbolic computations (DERIV, POLY_10).

Note that development of standard benchmark set relevant to the visual surveillance application domain is still a challenge, because different stages of video processing (low-level and high-level) demand different performance requirements. At present, we can demonstrate only that the Actor Prolog system is fast enough for real-time analyzing clips of the standard data set [8].

The translator creates Java classes corresponding to the classes of an object-oriented Actor Prolog program. Given external Java classes can be declared as ancestors of these automatically created classes and this is the basic principle of the implementation of built-in classes [21] and integration of Actor Prolog programs with external libraries. The possibility of easy extension of the Actor Prolog programming system by new built-in classes is a benefit of the selected implementation strategy. For instance, the Java2D and the Java3D libraries are connected with the Actor Prolog system in this way.

5 Low-level video processing

A typical intelligent video surveillance system includes high-level procedures (that is, anomalous human activity recognition) and low-level video processing procedures (for instance, background subtraction, discrimination of foreground blobs, tracking blobs over time, detection of interactions between the blobs, etc.). We have developed and implemented in Java the $'ImageSubtractor'$ built-in Actor Prolog class supporting all necessary low-level procedures. This class implements the following means:

1. Video frames pre-processing including 2D-gaussian filtering, 2D-rank filtering, and background subtraction.
2. Recognition of moving blobs and creation of Prolog data structures describing the co-ordinates of the blobs in each moment.
3. Recognition of tracks of blob motions and creation of Prolog data structures describing the co-ordinates and the velocity of the blobs. The tracks are divided into separate segments; there are points of interaction between the blobs at the ends of a segment.
4. Recognition and ejection of immovable and slowly moving objects. This feature is based on a simple fuzzy inference on the attributes of the tracks (the co-ordinates of the tracks and the average velocities of the blobs are considered).
5. Recognition of connected graphs of linked tracks of blob motions and creation of Prolog data structures describing co-ordinates and velocities of blobs. We consider two tracks as linked if there are interactions between the blobs of these tracks. In some applications, it is useful to eject tracks of immovable and slowly moving objects from the graphs before further processing of the video scenes.

We have started our experiments with low-level procedures implemented in pure Java; however, it is clear that further development of video surveillance methods requires usage of advanced computer vision libraries. A promising approach for implementation of the low-level recognition procedures in a logic language is usage of the OpenCV computer vision library and we are planning to link Actor Prolog with the JavaCV library that is a Java interface to OpenCV.

6 Conclusion

We have created a research software platform based on the Actor Prolog concurrent object-oriented logic language and a state-of-the-art Prolog-to-Java translator for experimenting with the intelligent visual surveillance. The platform includes the Actor Prolog logic programming system and an open source Java library of Actor Prolog built-in classes [21]. It is supposed to be complete for facilitation of research in the field of intelligent monitoring of anomalous people activity and studying logical description and analysis of people behavior.

Our study has demonstrated that translation from a concurrent object-oriented logic language to Java is a promising approach for application of the logic programming to the problem of intelligent monitoring of people activity; the Actor Prolog logic programming system is suitable for this purpose and ensures essential separation of the recognition process into concurrent sub-processes implementing different stages of high-level analysis.

7 Acknowledgements

Authors are grateful to Abhishek Vaish, Vyacheslav E. Antciperov, Vladimir V. Deviatkov, Aleksandr N. Alfimtsev, Vladislav S. Popov, and Igor I. Lychkov for cooperation.

The valuable comments of the anonymous referees are gratefully appreciated.

We acknowledge a partial financial support from the Russian Foundation for Basic Research, grant No 13-07-92694.

References

1. Aggarwal, J., Ryoo, M.: Human activity analysis: A review. ACM Computing Surveys (CSUR) 43(3), 16:1–16:43 (Apr 2011), http://doi.acm.org/10.1145/1922649.1922653
2. Banbara, M., Tamura, N., Inoue, K.: Prolog Cafe: A Prolog to Java translator system. In: Umeda, M., Wolf, A., Bartenstein, O., Geske, U., Seipel, D., Takata, O. (eds.) Declarative Programming for Knowledge Management. pp. 1–11. LNAI 4369, Springer, Heidelberg (2006)
3. Codognet, P., Diaz, D.: wamcc: Compiling Prolog to C. In: Sterling, L. (ed.) ICLP 1995. pp. 317–331. MIT Press (1995)
4. Cook, J.J.: Optimizing P#: Translating Prolog to more idiomatic C#. In: CICLOPS 2004. pp. 59–70 (2004)
5. Demoen, B., Tarau, P.: jProlog home page (1997), http://people.cs.kuleuven.be/~bart.demoen/PrologInJava/
6. Eichberg, M.: Compiling Prolog to idiomatic Java. In: Gallagher, J.P., Gelfond, M. (eds.) ICLP 2011. pp. 84–94. Dagstuhl Publishing, Saarbrücken/Wadern (2011)
7. Filippou, J., Artikis, A., Skarlatidis, A., Paliouras, G.: A probabilistic logic programming event calculus (2012), http://arxiv.org/abs/1204.1851
8. Fisher, R.: CAVIAR test case scenarios. The EC funded project IST 2001 37540. (2007), http://homepages.inf.ed.ac.uk/rbf/CAVIAR/

9. Fujise, T., Chikayama, T., Rokusava, K., Nakase, A.: KLIC: A portable implementation of KL1. In: FGCS 1994. pp. 66–79. ICOT, Tokyo (Dec 1994)
10. Henderson, F., Somogyi, Z.: Compiling Mercury to high-level C code. In: CC 2002. Grenoble, France (2002)
11. Junior, J., Musse, S., Jung, C.: Crowd analysis using computer vision techniques. A survey. IEEE Signal Processing Magazine 27(5), 66–77 (Sep 2010)
12. Kim, I., Choi, H., Yi, K., Choi, J., Kong, S.: Intelligent visual surveillance—a survey. International Journal of Control, Automation, and Systems 8(5), 926–939 (2010)
13. Kuramochi, S.: KLIJava home page (1999), http://www.ueda.info.waseda.ac.jp/~satoshi/klijava/klijava-e.html
14. Machot, F., Kyamakya, K., Dieber, B., Rinner, B.: Real time complex event detection for resource-limited multimedia sensor networks. In: Workshop on Activity monitoring by multi-camera surveillance systems (AMMCSS). pp. 468–473 (2011)
15. Morozov, A.A.: Actor Prolog: an object-oriented language with the classical declarative semantics. In: Sagonas, K., Tarau, P. (eds.) IDL 1999. pp. 39–53. Paris, France (Sep 1999), http://www.cplire.ru/Lab144/paris.pdf
16. Morozov, A.A.: On semantic link between logic, object-oriented, functional, and constraint programming. In: MultiCPL 2002. Ithaca, NY, USA (Sep 2002), http://www.cplire.ru/Lab144/multicpl.pdf
17. Morozov, A.A.: Logic object-oriented model of asynchronous concurrent computations. Pattern Recognition and Image Analysis 13(4), 640–649 (2003), http://www.cplire.ru/Lab144/pria640.pdf
18. Morozov, A.A.: Operational approach to the modified reasoning, based on the concept of repeated proving and logical actors. In: Salvador Abreu, V.S.C. (ed.) CICLOPS 2007. pp. 1–15. Porto, Portugal (Sep 2007), http://www.cplire.ru/Lab144/ciclops07.pdf
19. Morozov, A.A.: Visual logic programming based on the SADT diagrams. In: Dahl, V., Niemela, I. (eds.) ICLP 2007. pp. 436–437. LNCS 4670, Springer, Heidelberg (2007)
20. Morozov, A.A.: Actor Prolog to Java translation (in Russian). In: IIP-9. pp. 696–698. Torus Press Moscow, Budva, Montenegro (2012)
21. Morozov, A.A.: A GitHub repository containing source codes of Actor Prolog built-in classes (including the Vision package) (2014), https://github.com/Morozov2012/actor-prolog-java-library
22. Morozov, A.A., Vaish, A., Polupanov, A.F., Antciperov, V.E., Lychkov, I.I., Alfimtsev, A.N., Deviatkov, V.V.: Development of concurrent object-oriented logic programming system to intelligent monitoring of anomalous human activities. In: Jr., A.C., Plantier, G., Schultz, T., Fred, A., Gamboa, H. (eds.) BIODEVICES 2014. pp. 53–62. SCITEPRESS (Mar 2014), http://www.cplire.ru/Lab144/biodevices2014.pdf
23. O'Hara, S.: VERSA—video event recognition for surveillance applications. M.S. thesis. University of Nebraska at Omaha. (2008)
24. Shet, V., Singh, M., Bahlmann, C., Ramesh, V., Neumann, J., Davis, L.: Predicate logic based image grammars for complex pattern recognition. International Journal of Computer Vision 93(2), 141–161 (Jun 2011)
25. Tarau, P.: The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. Theory and Practice of Logic Programming 12(1–2), 97–126 (2012)

Extending the Finite Domain Solver of GNU Prolog

Vincent Bloemen[1], Daniel Diaz[2], Machiel van der Bijl[3], and Salvador Abreu[4]

[1] University of Twente, Formal Methods and Tools group, The Netherlands
`v.bloemen@student.utwente.nl`
[2] University of Paris 1-Sorbonne - CRI, France
`daniel.diaz@univ-paris1.fr`
[3] Axini B.V., The Netherlands
`vdbijl@axini.com`
[4] Universidade de Évora and CENTRIA, Portugal
`spa@di.uevora.pt`

Abstract. This paper describes three significant extensions for the Finite Domain solver of GNU Prolog. First, the solver now supports negative integers. Second, the solver detects and prevents integer overflows from occurring. Third, the internal representation of sparse domains has been redesigned to overcome its current limitations. The preliminary performance evaluation shows a limited slowdown factor with respect to the initial solver. This factor is widely counterbalanced by the new possibilities and the robustness of the solver. Furthermore these results are preliminary and we propose some directions to limit this overhead.

1 Introduction

Constraint Programming [1,7,16] emerged, in the late 1980s, as a successful paradigm with which to tackle complex combinatorial problems in a declarative manner [11]. However, the internals of constraint solvers, particularly those over Finite Domains (FD) were wrapped in secrecy, only accessible to only a few highly specialized engineers. From the user point of view, a constraint solver was an opaque "black-box" providing a fixed set of (hopefully) highly optimized constraints for which it ensures the consistency.

One major advancement in the development of constraint solvers over FD is, without any doubt, the article from Van Hentenryck et al. [12]. This paper proposed a "glass-box" approach based on a single *primitive constraint* whose understanding is immediate. This was a real breakthrough with respect to the previous way of thinking about solvers. This primitive takes the form X `in` r, where X is an FD variable and r denotes a *range* (i.e. a set of values). An X `in` r constraint enforces X to belong to the range denoted by r *which can involve other FD variables*. An X `in` r constraint which depends on another variable Y becomes *store sensitive* and must be (re)activated each time Y is updated, to ensure the consistency. The X `in` r constraint can be seen as embedding the core propagation mechanism. Indeed, it is possible to define different propagation schemes for a given constraint, corresponding to different degrees of consistency.

It possible to define high-level constraints, such as equations or inequations, in terms of X `in` r primitive constraints. It is worth noticing that these constraints are therefore not built into the theory. From the theoretical point of view, it is only necessary to work at the primitive level as there is no need to give special treatment to high-level constraints. This approach yielded significant advances in solvers. From the implementation point of view, an important article is [5] which proposed a complete machinery (data structures, compiler, instruction set) to efficiently implement an FD solver based on X `in` r primitives. It also shows how some optimizations at the primitive level can, in turn, be of great benefit to all high-level constraints. The resulting system, called `clp(FD)`, proved the efficiency of the approach: the system was faster than CHIP, a highly optimized black-box solver which was a reference at the time. This work has clearly inspired most modern FD solvers (SICStus Prolog, bProlog, SWI Prolog's clpfd, Choco, Gecode, ...) but also solvers over other domains like booleans [4], intervals [10] or sets [9,2]. Returning to FD constraints, a key point is the ability to reason on the outcome of a constraint (success or failure). Again, the "RISC" approach restricted the theoretical work about *entailment* at the primitive level [3]. This allowed a new kind of constraints: *reified constraints*, in which a constraint becomes concretized. The "RISC" approach was also very convenient for *constraint retraction* [6].

When GNU Prolog was developed, it reused the FD solver from `clp(FD)`. The X `in` r constraint was generalized to allow the definition of new high-level constraints, e.g. other arithmetic, symbolic, reified and global constraints. Nevertheless, the internals of the solver were kept largely unchanged. The outcome is a fast FD solver, but also one with some limitations:

- First, the domain of FD variables is restricted to positive values (following the original paper [12]). This is not restrictive from a theoretical point of view: a problem can always be "translated" to natural numbers but, from a practical point of view, there are several drawbacks: the translation is error-prone, the resulting programs are difficult to read and can exhibit significant performance degradation.
- Second, the domain representation uses a fixed-size bit-vector to encode sparse domains. Even if this size can be controlled by the user, it is easily and frequently misused. In some cases, the user selects a very large value for simplicity, without being aware of the waste of memory nor the loss of efficiency this induces.
- Lastly, for efficiency reasons the GNU Prolog solver does not check for integer overflows. This is generally not a problem when the domains of FD variables are correctly specified, as all subsequent computation will not produce any overflow. However, if one forgets to declare all domains, or does not declare them at the beginning, an overflow can occur. This is the case of:

```
|?- X * Y #= Z.
No
```

Indeed, without any domain definition for X and Y the non-linear constraint `X * Y #= Z` will compute the upper bound of Z as $2^{28} \times 2^{28}$ which overflows 32 bits resulting in a negative value for the *max*, thus the failure ($max < min$). This behaviour is hard to understand and requires an explanation. We admit this is not the *Way of Prolog* and does not help to promote constraint programming to new users. We could raise an exception (e.g. `instantiation_error` or `representation_error`) but this would still be of little help to most users.

In this article we describe and report on initial results for the extension and modification of the GNU Prolog FD solver to overcome these three limitations. This is a preliminary work: special attention has been put on ensuring correctness and the implementation is not yet optimized. Nevertheless the results are encouraging, as we shall see, and there is ample room and directions to research on performance improvements.

The remainder of this article is organized as follows: Section 2 introduces some important aspects of the original FD solver required to understand the modifications. Section 3 is devoted to the inclusion of negative values in FD domains. Section 4 explains how integer overflow is handled in the new solver, while Section 5 explains the new representation for sparse domains. A performance evaluation may be found in Section 6. Section 7 provides some interesting directions to optimize the overall performance. A short conclusion ends the paper.

2 The GNU Prolog FD Solver

The GNU Prolog solver follows the "glass-box" approach introduced by Van Hentenryck et al. in [12], in which the authors propose the use of a single *primitive constraint* of the form X `in` r, where X is an FD variable and r denotes a *range* (ie. a set of values). An X `in` r constraint enforces X to belong to the range denoted by r which can be constant (e.g. the interval 1..10) but can also use the following *indexicals*:

- `dom(`Y`)` representing the whole current domain of Y.
- `min(`Y`)` representing the minimum value of the current domain of Y.
- `max(`Y`)` representing the maximum value of the current domain of Y.
- `val(`Y`)` representing the final value of the variable of Y (when its domain is reduced to a singleton). A constraint using this indexical is postponed until Y is instantiated.

An X `in` r constraint which uses an indexical on another variable Y becomes *store sensitive* and must be (re)activated each time Y is updated to ensure the consistency. Thanks to X `in` r primitive constraints it is possible to define high-level constraints such as equations or inequations. Obviously all solvers offer a wide variety of predefined (high-level) constraints to the programmer. Nevertheless, the experienced user can define his own constraints if needed.

The original FD solver of GNU Prolog is also based on indexicals. Its implementation is widely based on its predecessor, `clp(FD)` [5]. In the rest of this section we only describe some aspects of the original implementation which are important later on. The interested reader can refer to [8] for missing details.

2.1 The FD definition language

The original X `in` r is not expressive enough to define all needed constraints in practice. We thus defined the FD language: a specific language to define the constraints of the GNU Prolog solver. Figure 1 shows the definition of the constraint $A \times X = Y$ in the FD language:

```
ax_eq_y(int A, fdv X, fdv Y)                 /* here A != 0 */
{
  start X in min(Y) /> A .. max(Y) /< A       /* X = Y / A */
  start Y in min(X) *  A .. max(X) *  A       /* Y = X * A */
}
```

Fig. 1. Definition of the constraint $A \times X = Y$ in the FD language

The first line defines the constraint name (`ax_eq_y`) and its arguments together with their types (`A` is expected to be an integer, `X` and `Y` FD variables). The `start` instruction installs and activates an X `in` r primitive. The first primitive computes X from Y in the following way: each time a bound of Y is modified the primitive is triggered to reduce the domain of X accordingly. The operator `/>` (resp. `/<`) denote division rounded upwards (resp. downwards). Similarly, the second primitive updates (the bounds) of Y with repect to X. This is called *bound consistency* [1] : if a *hole* appears inside the domain of X (i.e. a value V different from both the min and the max of X has beed removed from the domain of X), the corresponding value $A \times V$ will not be removed from the domain of Y. If wanted, such a propagation (called *domain consistency*) could be specified using the `dom` indexical.

A compiler (called `fd2c`) translates an FD file to a C source file. The use of the C language as target is motivated by the fact that all the GNU Prolog system is written in C (so the integration is simple) but mainly by the fact that modern C compilers produce very optimized code (this is of prime importance if we consider that a primitive constraint can be awoken several thousand times in a resolution). When compiled such a definition gives rise to different C functions:

- the *main function*: a public function (`ax_eq_y`) which mainly creates an environment composed of the 3 arguments (A, X, Y) and invokes the installation functions for the involved X `in` r primitives.
- the *installation function*: a private function for each X `in` r primitive which is responsible for the installation of the primitive. This consists of installing

the dependencies (e.g. add a new dependency to Y, so that each time Y is modified the primitive is re executed to update X) and the execution function is invoked (this is the very first execution of the primitive).
– the *execution function*: a private function for each X `in` r primitive which computes the actual value of r and enforces $X \in r$. This function will be (re)executed each time an FD variable appearing in the definition of r is updated.

2.2 Internal domain representations

There are 2 main representations of a domain (range):

– *MinMax*: only the *min* and the *max* are stored. This representation is used for intervals (including `0..fd_max_integer`).
– *Sparse*: this representation is used as soon as a hole appears in the domain of the variable. In that case, in addition to the *min* and the *max*, a bit-vector is used to record each value of the range.

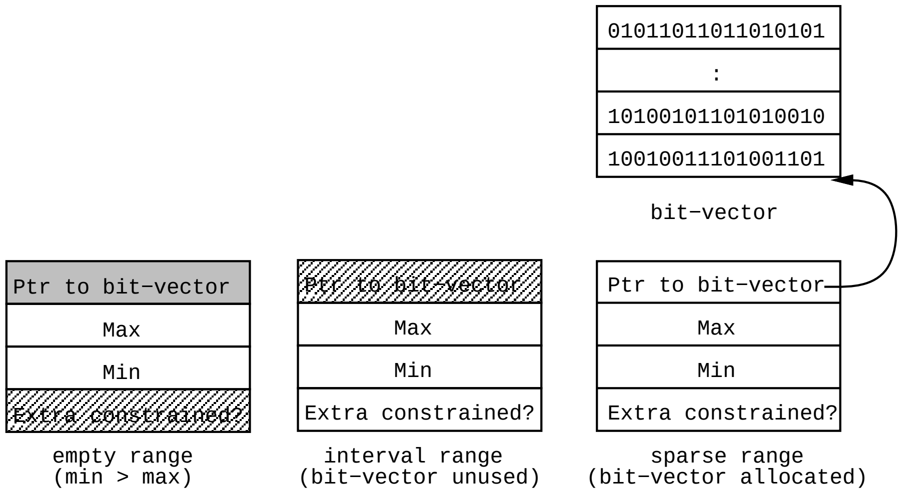


Fig. 2. Representations of a range

When an FD variable is created it uses a *MinMax* representation. As soon as a “hole” appears it is transparently switched to a *Sparse* representation which uses a bit-vector. For efficiency reasons all bit-vector have the same size inside `0..fd_vector_max`. By default `fd_vector_max` equals 127 and can be redefined via an environment variable or via a built-in predicate (this should be done before any constraint is told). When a range becomes *Sparse*, some values are possibly lost if `fd_vector_max` is less than the current *max* of the variable. To inform the user of this source of incompleteness, GNU Prolog maintains a flag to indicate that a range has been *extra constrained* by the solver (via an imaginary constraint X *in* 0..`fd_vector_max`). The flag *extra_cstr* associated to each range is updated by all operations, e.g. the intersection of two ranges is extra-constrained iff both ranges are extra constrained, thus the resulting flag is the logical *and* between the two flags. When a failure occurs on a variable whose domain is extra constrained a message is displayed to inform the user that some solutions can be lost since bit-vectors are too small. Finally an empty range is represented with $min > max$. This makes it possible to perform an intersection between R_1 and R_2 in *MinMax* mode simply with $Max(min(R_1), min(R_2))..Min(max(R_1), max(R_2))$ which returns $min > max$ if either R_1 or R_2 is empty. Figure 2 shows the different representations of a range.

3 Supporting Negative Values

In this section we describe how the inclusion of negative values in FD variables is realized. First we show why the current implementation does not support negative values. Then we show how to address the problems by mainly focusing on the implementation. This section only describes bound consistency; negative values are handled similarly in domain consistency due to the new sparse design, described in Section 5.

3.1 Current limitations

The current implementation does not support negative values, FD variables stay within the bounds `0..fd_max_integer`. Adding support for negative values seems obvious at a first glance, however some attention has to be paid. The modifications concern constraints whose current implementation implicitly utilize the fact that values are always positive, which is no longer valid. Other modifications concern constraints which are sign sensitive from the interval arithmetical point of view. This is the case for multiplication: if X is in $min..max$ then $-X$ is in $-max..-min$. Let us consider again the case of the constraint $A \times X = Y$ whose current definition is presented in Figure 1. Presuming that A can be negative the current definition will not update the domains of X and Y correctly: in that case X will be constrained to $\lceil \frac{min(Y)}{A} \rceil..\lfloor \frac{max(Y)}{A} \rfloor$ which produces an empty interval since $min(X) > max(X)$. To support negative values in FD variables, this instance, as well as other arithmetical constraints require updating to handle negative values properly.

3.2 Method and approach

One possible approach to deal with negative numbers is to construct a mapping for negative values to natural numbers so that the arithmetic constraints can continue to operate strictly on the positive domain. Another approach is to update the constraints to be fully functional for both positive and negative domains. The former is undesirable since the translation quickly becomes cumbersome and would carry a considerable performance impact. Aside from that, several operations such as taking the power or root are affected by the variable sign. As the latter approach is less error-prone and more robust, we chose to implement it and thus need to reformulate several arithmetic constraints.

First, the initial domain bounds of FD variables are updated to range in `fd_min_integer..fd_max_integer`. To remain backwards compatible, an environment variable is created that, if set, will use the original bounds for FD variables.

On updating the arithmetic constraints, all possible cases for each FD variable need to be considered, that is < 0, $= 0$ and > 0 for both the *min* and *max* of the variable. For instance, the $A \times X = Y$ constraint from Figure 1 is updated as follows:

```
ax_eq_y(int A, fdv X, fdv Y)                 /* A != 0 */
{
 start X in ite(A>0, min(Y), max(Y)) /> A    /* X = Y / A */
         .. ite(A>0, max(Y), min(Y)) /< A
 start Y in ite(A>0, min(X), max(X)) * A     /* Y = X * A */
         .. ite(A>0, max(X), min(X)) * A
}
```

where `ite` represents an if-then-else expression (corresponding to the C operator `?:`). This modification ensures that for all interpretations of A, X and Y the domains are updated correctly.

A more complex example is the constraint $X^A = Y$, where X and Y are FD variables and A is an integer > 2. In the current version, this constraint is given as follows:

```
x_power_a_eq_y(fdv X, int A, fdv Y)          /* A > 2 */
{
 start Y in Power(min(X), A)..Power(max(X), A)
 start X in Nth_Root_Up(min(Y), A)..Nth_Root_Dn(max(Y), A)
}
```

With the introduction of negative values, the constraint is specified as:

```
x_power_a_eq_y(fdv X, int A, fdv Y)          /* A > 2 */
{
 start X in ite(is_even(A),
                min_root(min(X), min(Y), max(Y), A),
                ite(min(Y) < 0,
                    -Nth_Root_Dn(-min(Y), A),
                    Nth_Root_Up(min(Y), A)))
         .. ite(is_even(A),
                max_root(max(X), min(Y), max(Y), A),
                    ite(max(Y) < 0,
                        -Nth_Root_Up(-max(Y), A),
                        Nth_Root_Dn(max(Y), A)))

 start Y in ite(min(X) < 0 && is_odd(A),
                Power(min(X), A),
                Power(closest_to_zero(min(X), max(X)), A))
         .. ite(min(X) < 0 && is_even(A),
                Power(Max(abs(min(X)), max(X)), A),
                Power(max(X), A))
}
```

here, a couple of C functions and macros are introduced:

- `Min` and `Max` are used to compute the minimum resp. maximum of two values.
- `is_even` and `is_odd` return wether the variable is even or odd.
- `min_root` and `max_root` calculate the minimum and maximum value of $\pm \sqrt[A]{Y}$ that lie in the bounds of `min(X)..max(X)`.
- `Power` and `Nth_Root` refer to C functions that calculate the n^{th} power and n^{th} root of a variable.
- `closest_to_zero(A,B)` returns the closest value to 0 in the interval `A..B`.

In this specification, Y can only include negative values if X contains negative values and A is an odd number (e.g. $-2^3 = -8$). Similarly, if Y is strictly positive, X can only take negative values if A is an even number (e.g. $-2^4 = 16$). In short, the above constraint needs to distinguish between even and odd powers of X, which was originally unnecessary. With this definition, the following query correctly reduces the domains of X and Y:

```
|?- fd_domain([X,Y],-50,150), X ** 3 #= Y.
X = _#0(-3..5)
Y = _#17(-27..125)
```

The support for negative values in FD variables is achieved by carefully re-designing the arithmetic constraints. An obvious side-effect of the modifications is that some overhead is introduced, even when considering strictly positive FD variables. The benchmark tests, see Section 6, will show the impact of the modifications compared to the original solver.

4 Handling Integer Overflows

4.1 Current limitations

The current implementation of GNU Prolog does not check for overflows. This means that without preliminary domain definitions for X, Y and Z, the non-linear constraint $X \times Y = Z$ will fail due to an overflow when computing the upper bound of the domain of Z : $2^{28} \times 2^{28}$. In 32-bit arithmetic, this overflow causes a negative result for the upper bound and the constraint then fails since $min(X) > max(X)$.

At present, the user needs to adapt the variable bounds beforehand to prevent this constraint from failing. To reduce the burden to the user and improve the robustness of the solver, we propose a better way of handling overflows.

4.2 Method and approach

There are two approaches to handle overflows. One is to report the problem via an ISO exception (e.g. `evaluation_error`), thereby informing the user that the domain definitions for the FD variables are too mild and should be made more restrictive. The other approach is to instrument the solver to detect overflows and cap the result. As placing less restrictions on the user and more robustness for the solver is desirable, the second approach is chosen.

The key idea behind preventing overflows is to detect when one would occur and provide means to restrict this from happening. For the solver this means that when a multiplication or power operation is applied in a constraint, an overflow prevention check should be considered. This can also be the case for other arithmetic operations.

Consider again the constraint $X \times Y = Z$. Because both $1 \times 2^{28} = 2^{28}$ and $2^{28} \times 1 = 2^{28}$, the maximum value that both X and Y can take is 2^{28}. Therefore the following (and current implementation) for finding the domain for Z causes an overflow:

```
start Z in min(X) * min(Y) .. max(X) * max(Y)
```

For this case and similar instances, the following function is designed to cap results of arithmetic, thereby preventing overflows:

```
static int inline mult(int a, int b)
{
  int64_t res = ((int64_t) a) * ((int64_t) b);
  if (res > max_integer)
    res = max_integer;
  else if (res < min_integer)
    res = min_integer;
  return (int) res;
}
```

Since integers only need 29-bits, the 64-bit result is enough to check if an overflow occurs and cap the result if needed. In the constraint definitions, the standard multiplication gets replaced with a `mult` call when it could cause an overflow. For the $X \times Y = Z$ constraint, this is as follows:[5]

```
start Z in mult(min(X), min(Y)) .. mult(max(X), max(Y))
```

[5] The constraint is further modified for negative values, along the same lines.

As a consequence, the $X \times Y = Z$ constraint now gives the following result:

```
| ?- X * Y #= Z.
X = _#3(-268435456..268435455)
Y = _#20(-268435456..268435455)
Z = _#37(-268435456..268435455)
```

where `-268435456 = fd_min_integer` and `268435455 = fd_max_integer`.

At first, we used `mult` for every applied multiplication in the constraint definitions. However, in some cases it is not necessary to check for overflows. For instance, consider the implementations for `ax_eq_y` and `x_power_a_eq_y` of Section 3.2. By first restricting the domain of X (in both cases), no overflow can occur when the domain of Y is calculated. Note that if the domain of Y is computed first, an overflow could happen. Note however, that such an optimization is not possible for some constraints, for instance $X \times Y = Z$, since the domains of X and Y do not necessarily get reduced.

In conclusion, even if several overflow problems could be resolved by rearranging the order of execution, in general it is necessary to take preventive measures.

5 New Domain Representation

5.1 Current limitations

In the current implementation, when a domain gets a *hole*, its representation is switched to the *Sparse* form, which stores domains using a static-sized bit-vector. The problem with this approach is that values which lie outside the range `0..fd_vector_max` are lost. An internal flag *extra_cstr* is set when this occurs to inform the user of lost values. Even though the user is able to globally set `fd_vector_max`, there are several problems with this representation:

- The user has to know the variable bounds in advance; an over-estimate of the domain size results in a waste of memory (and loss of efficiency).
- There is an upper-limit for `fd_vector_max` which is directly related to the available memory space in bits. Also note that doing operations on a large bit-vector can severely impact the performance.
- The current *Sparse* representation is unable to store negative values.

5.2 Method and approach

To deal with the limitations, a redesign is needed for the *Sparse* representation. Some research has been done in representing sparse domains [13,14]. Considering the requirements – remain efficient while taking away the limitations – there are several options for the redesign, while also considering alternatives and variations:

1. Use a list of *MinMax* chunks: Store only the minimum and maximum of consecutively set values. The values between two chunks are defined to be all unset. This is especially effective if the number of holes is small or large gaps exist in the domain.
2. Use a list of bit-vector chunks: Use a bit-vector together with an offset to store all (un)set actual values. The values between two chunks can either be defined as all set or all unset (possibly defined per chunk with a variable). This is in particular effective on small domains with many holes.
3. A combination of (1) and (2): Determine per chunk whether it should be a *MinMax* chunk or bit-vector chunk, so that the number of total chunks is minimal. This takes the advantages of both individual options but it does introduce extra overhead for determining which representation to choose and operations between two different chunk representations can become difficult.

Note that all suggested model takes away the limitations of the current design. Le Clément et al. [13] provide a more in-depth analysis on the different representations with respect to their time complexities. Note that differences arise for specific operations on domains: for instance, a value removal is done more efficiently in a bit-vector while iteration is more efficient on *MinMax* chunks.

We initially opted for the combination of the *MinMax* and bit-vector chunks because the extra overhead is presumed to not be a significant factor. For the moment, however, we implemented a list of *MinMax* chunks. Its performance compared to the original *Sparse* implementation shows a limited slowdown factor, as discussed in Section 6. Because of these results (a slowdown is expected anyway, due to the new possibilities), the addition of a bit-vector representation was postponed. We now discuss the new implementation using a list of *MinMax* chunks.

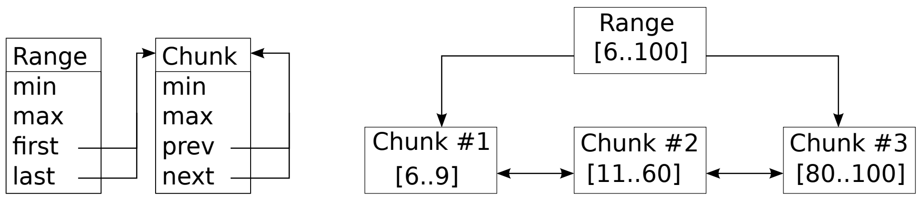


Fig. 3. Left: UML diagram of the new *Sparse* range, right: example for representing the set of values {`6..9,11..60,80..100`}.

The domain initially uses a *MinMax* representation (just a Range instance) which only stores min and max, with first and last being null pointers. When a hole appears, the domain switches to the *Sparse* representation by adding Chunk instances. The range keeps track of the first and last chunk of the list and continues to maintain the min and max of the whole domain (`range.min = range.first.min`). The list is a doubly-linked list for efficient insertion and removal of chunks, each chunk maintains its minimum and maximum values. This representation is depicted in Figure 3.

For every two consecutive chunks c_1 and c_2, we have $c_1.\mathsf{max} + 1 < c_2.\mathsf{min}$; chunks are sorted and always have at least one unset value between them. Furthermore, $c_i.\mathsf{min} \leq c_i.\mathsf{max}$.

Operations on *Sparse* ranges (e.g. intersection, union, ...) are efficiently done by iterating over the chunks and updating these in place whenever possible. An example of this is provided in Table 1 for intersecting two *Sparse* ranges. The implementation only considers one chunk of each range at a time and the cases are considered from top to bottom.

Case:	Action (in pseudo code):
`chunk_1.max` < `chunk_2.min`	- Remove `chunk_1` - `chunk_1` = `chunk_1.next` // advance `chunk_1`
`chunk_1.max` ≤ `chunk_2.max`	- Create new chunk and set before `chunk_1` with `min` = *Max*(`chunk_1.min, chunk_2.min`) and `max` = *Min*(`chunk_1.max, chunk_2.max`) - `chunk_1` = `chunk_1.next` // advance `chunk_1`
`chunk_1.min` > `chunk_2.max`	- `chunk_2` = `chunk_2.next` // advance `chunk_2`
`chunk_1.max` > `chunk_2.max`	- Create new chunk and set before `chunk_1` with `min` = *Max*(`chunk_1.min, chunk_2.min`) and `max` = *Min*(`chunk_1.max, chunk_2.max`) - `chunk_2` = `chunk_2.next` // advance `chunk_2`

Table 1. Implementation of the range intersection operation.

Because the solver may need to backtrack, domains need to be trailed. Modifications on domains can cause its chunks to disperse in memory, therefore all chunks of the domain are saved on the trail, upon modification. A classical timestamp technique is used to avoid trailing more than once per choice-point.

With this new implementation for the *Sparse* domain, it is now possible to store negative values and the domain bounds are no longer limited to a static arbitrary value, thereby rendering the *extra_cstr* flag useless.

6 Performance Analysis

In this section we compare the original FD constraint solver to a version that includes the new extensions. Table 2 presents the total execution times (in milliseconds) for runs of several benchmarks. *Neg + Ovfl* consists of the negative values extension and the overflow prevention (the *Ovfl* extension is implemented simultaneously with *Neg*). *Neg + Ovfl + Dom* includes all three extensions presented in this article. Times are measured on a 64-bit i7 Processor, 2.40GHz×8 with 8GB memory running Linux (Ubuntu 13.10).[6]

[6] The results can be reproduced with version 1.4.4 of GNU Prolog for the current version and the git branch `negative-domain` for the new version.

Program	Original	Neg + Ovfl		Neg + Ovfl + Dom	
	Time	Time	Speedup	Time	Speedup
`queens 29`	429	414	1.04	644	0.66
`digit8 ff` (×100)	787	1197	0.66	1082	0.73
`qg5 11` (×10)	610	593	1.03	813	0.75
`queens ff 100`	156	153	1.02	201	0.77
`partit 600`	200	266	0.75	254	0.79
`eq20` (×100)	189	249	0.76	228	0.83
`crypta` (×1000)	888	1016	0.87	1075	0.83
`langford 32`	551	549	1.00	646	0.85
`magsq 11`	810	802	1.01	923	0.88
`multipl` (×10)	567	577	0.98	604	0.94
`magic 200`	180	178	1.02	180	1.00
`donald` (×10)	167	158	1.06	166	1.00
`alpha` (×10)	409	407	1.00	396	1.03
`interval 256` (×10)	217	205	1.06	140	1.55
Geometric mean	364	389	**0.94**	413	**0.88**

Table 2. Performance Impact of Extensions (times in ms.)

The original implementation and the benchmark tests are solely designed for the positive domain. Therefore the domain bounds are restricted to positive values (using the environment variable discussed in Section 3.2), while making use of the updated constraint definitions. Multiple test runs show an estimated standard deviation of 3 milliseconds. The annotation (×10) indicates that the test time is formed from 10 consecutive executions (to reduce the effect of the deviation).

On average, the introduction of negative domains + overflow detection penalizes the benchmarks by 6%. This slowdown is an expected consequence of the increased complexity, and we are quite pleased that it turns out small. The worst case is for `digit8 ff` with a 34% performance loss (see [15] for a definition of "performance gain"). The reason for this is because the square root is often calculated, which is slower as both the positive and negative solutions are considered in the predicates. The best case scenario is for `donald`, which exhibits a 6% performance gain over the base version: the redesign for the predicates actually improved the solver's performance in several cases.

With the inclusion of the new *Sparse* domain alongside the other extensions, on average the benchmarks suffer a performance loss of 12%. The worst case test is `queens 29` with 34% and the best case, `interval 256`, has a 55% performance gain over the base version. The `queens 29` test creates a lot of holes on a small domain which is more efficient with a bit-vector than *MinMax* chunks. The `interval 256` test often iterates on domains: this is more efficient in the new *Sparse* domain because finding the n^{th} element is achieved in *O(nr. of holes)* time. The base version has to iterate over the bit-vector until the n^{th} element is found, making the time complexity *O(size of bit-vector)*.

Note that these benchmark tests do not utilize the enhanced capabilities of the new solver. For instance, test programs that use the negative domain cannot be tested in the original solver. It is therefore difficult to make a fair comparison.

7 Future Work

While the results show that the extensions only cause a limited slowdown factor, there is much room for improvements.

The measures taken to prevent overflows can be optimized further. In the new implementation, several unnecessary preventive checks are still being done: for instance, for the constraint $X + Y = Z$ no overflow detection is needed when computing Z, since adding two 29-bit values cannot cause overflow in 32-bit arithmetic, yet it's being checked for. Furthermore, when the run-time domain bounds imply that no overflows can occur; for instance if X and Y are in 0..10 there is no need to check for overflow in the constraint $X \times Y = Z$, since domains are reduced monotonically. As seen in section 3.2, supporting negative numbers for $X^A = Y$ implies testing the parity of A. At present this is done every time the constraint is reactivated, however, with a slightly more complex compilation scheme, there will be two versions of the execution function (see 2.1): one specialized for even As and another for odd. The installation function would be responsible to select the adequate execution function, depending on the actual value of A at run-time. This will entail enriching the FD language to be able to express user-specified installation procedures.

It will definitely be interesting to combine our new *Sparse* domain representation with bit-vectors, whenever applicable. We will experiment in this direction. Similarly, instead of using a (doubly-linked) list for maintaining chunks, a tree-structure is likely to be more efficient. Ohnishi et al. [14] describe how a balanced tree structure is realized on interval chunks. Incorporation of this structure should improve the time complexity on insertion and deletion from $O(n)$ to $O(\log n)$ (for n as the number of chunks) in worst case scenarios.

The added expressiveness allows us to tackle more complex problems, which were previously hard or impossible to model. These will also have to be benchmarked against other systems.

8 Conclusion

We presented a set of extensions to the GNU Prolog FD solver which allow it to more gracefully handle real-world problems. Central to these is a domain representation that, in order to gain generality, forgoes the compactness found in the existing solver: we moved from static vectors to dynamic data structures. The solver is now also capable of handling negative values and measures were taken to improve its robustness and correctness. The result is a system which can more easily model complex problems.

The performance evaluation of the initial, suboptimal, implementation shows encouraging results: the slowdown is quite acceptable, in the order of 12%. Furthermore, we have proposed ways to further reduce the impact of these design options, and thus hope to reclaim the lost performance.

References

1. Krzysztof R. Apt. *Principles of constraint programming.* Cambridge University Press, 2003.
2. Federico Bergenti, Alessandro Dal Palù, and Gianfranco Rossi. Integrating Finite Domain and Set Constraints into a Set-based Constraint Language. *Fundam. Inform.*, 96(3):227–252, 2009.
3. Björn Carlson, Mats Carlsson, and Daniel Diaz. Entailment of Finite Domain Constraints. In Pascal Van Hentenryck, editor, *ICLP*, pages 339–353. MIT Press, 1994.
4. Philippe Codognet and Daniel Diaz. clp(B): Combining Simplicity and Efficiency in Boolean Constraint Solving. In Manuel V. Hermenegildo and Jaan Penjam, editors, *PLILP*, volume 844 of *Lecture Notes in Computer Science*, pages 244–260. Springer, 1994.
5. Philippe Codognet and Daniel Diaz. Compiling Constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
6. Philippe Codognet, Daniel Diaz, and Francesca Rossi. Constraint Retraction in FD. In Vijay Chandru and V. Vinay, editors, *FSTTCS*, volume 1180 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 1996.
7. Rina Dechter. *Constraint processing.* Elsevier Morgan Kaufmann, 2003.
8. Daniel Diaz, Salvador Abreu, and Philippe Codognet. On the implementation of GNU Prolog. *TPLP*, 12(1-2):253–282, 2012.
9. Carmen Gervet. Conjunto: Constraint Logic Programming with Finite Set Domains. In Maurice Bruynooghe, editor, *ILPS*, pages 339–358. MIT Press, 1994.
10. Frédéric Goualard, Frédéric Benhamou, and Laurent Granvilliers. An Extension of the WAM for Hybrid Interval Solvers. *Journal of Functional and Logic Programming*, 1999(Special Issue 1), 1999.
11. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.
12. Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, Implementation, and Evaluation of the Constraint Language cc(FD). In Andreas Podelski, editor, *Constraint Programming*, volume 910 of *Lecture Notes in Computer Science*, pages 293–316. Springer, 1994.
13. Vianney Le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, September 2013.
14. Shuji Ohnishi, Hiroaki Tasaka, and Naoyuki Tamura. Efficient Representation of Discrete Sets for Constraint Programming. In Francesca Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 920–924. Springer, 2003.
15. David A Patterson and John L Hennessy. *Computer Organization and Design: the Hardware/Software Interface.* Morgan Kaufmann, 2013.
16. F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.

A λProlog Based Animation of Twelf Specifications

Mary Southern and Gopalan Nadathur

University of Minnesota, Minneapolis MN 55455, USA

Abstract. Specifications in the Twelf system are based on a logic programming interpretation of the Edinburgh Logical Framework or LF. We consider an approach to animating such specifications using a λProlog implementation. This approach is based on a lossy translation of the dependently typed LF expressions into the simply typed lambda calculus (STLC) terms of λProlog and a subsequent encoding of lost dependency information in predicates that are defined by suitable clauses. To use this idea in an implementation of logic programming *a la* Twelf, it is also necessary to translate the results found for λProlog queries back into LF expressions. We describe such an inverse translation and show that it has the necessary properties to facilitate an emulation of Twelf behavior through our translation of LF specifications into λProlog programs. A characteristic of Twelf is that it permits queries to consist of types which have unspecified parts represented by meta-variables for which values are to be found through computation. We show that this capability can be supported within our translation based approach to animating Twelf specifications.

1 Introduction

The Edinburgh Logical Framework or LF [4] is a dependently typed lambda calculus that has proven useful in specifying formal systems such as logics and programming languages (see, e.g., [5]). The key to its successful application in this setting is twofold. First, the abstraction operator that is part of the syntax of LF provides a means for succinctly encoding formal objects whose structures embody binding notions. Second, LF types can be indexed by terms and, as such, they can be used to represent relations between objects that are encoded by terms. More precisely, types can be viewed as formulas and type checking as a means for determining if a given term represents a proof of that formula. Proof search can be introduced into this context by interpreting a type as a request to determine if there is a term of that type. Further, parts of a type can be left unspecified, thinking of it then as a request to fill in these parts in such a way that the resulting type is inhabited. Interpreting types in this way amounts to giving LF a logic programming interpretation. The Twelf system [9, 10] is a realization of LF that is based on such an interpretation.

An alternative approach to specifying formal systems is to use a predicate logic. Objects treated by the formal systems can be represented by the terms of this logic and relations between them can be expressed through predicates over these terms. If the terms include a notion of abstraction, e.g., if they encompass simply typed lambda terms, then they provide a convenient means for representing binding notions. By restricting the formulas that are used to model relations suitably, it is possible to constrain proof search behavior so that the formulas can be given a rule-based interpretation. The logic of higher-order hereditary Harrop formulas (*hohh*) has been designed with these ideas in mind and many experiments have shown this logic to be a useful specification device (see, e.g., [7]). This logic has also been given a computational interpretation in the language λProlog [8], for which efficient implementations such as the Prolog/Mali [1] and the Teyjus [11] systems have been developed.

The two different approaches to specification that are described above have a relationship that has been explored formally. In early work, Felty and Miller showed that LF derivations could be encoded in *hohh* derivations by describing a translation from the former to the latter [3]. This translation demonstrated the expressive power of *hohh*, but did not show the correspondence in proof search behavior. To rectify this situation, Snow et. al. described a transformation of LF specifications into *hohh* formulas that allowed the construction of derivations to be related [12]. This work also showed how to make the translation more efficient by utilizing information available from a static checking of LF types, and it refined the resulting *hohh* specifications towards making their structure more closely resemble that of the LF specifications they originated from.

The primary motivation for the work of Snow et. al. was a desire to use Teyjus as a backend for an alternative implementation of logic programming in Twelf. However, it falls short of achieving this goal in two ways that we address in this paper. First, although it relates derivations from LF specifications to ones from their translations, it does not make explicit the process of extracting an LF "result" term from a successful *hohh* derivation; such an extraction is necessary if Teyjus is to serve as a genuine, invisible backend. To close this gap, we describe an inverse translation and show that it has the necessary properties to allow Twelf behavior to be emulated through computations from λProlog programs. Second, Snow et. al. dealt only with closed types, i.e., they did not treat the idea of filling in missing parts of types in the course of looking for an inhabitant. To overcome this deficiency, we include meta-variables in specifications and treat them in the back-and-forth translations as well as in derivations; the last aspect, that is also the most critical one in our analysis, requires us to build substitutions and unification explicitly into our formalization of derivations.

The remainder of this paper is structured as follows. Sections 2 and 3 respectively present LF and the *hohh* logic together with their computational interpretations. Section 4 describes a translation from LF specifications into *hohh* ones together with an inverse translation for extracting solution terms from *hohh* derivations. We then propose an approach for developing a proof of correctness for this translation. Section 5 improves the basic translation and Section 6 uses it to illustrate our proposed approach to realizing logic programming in Twelf. Section 7 concludes the paper.

$$\frac{X : A \in \Delta}{\Gamma \vdash_\Sigma X : A}\ \textit{meta-var}$$

$$\frac{\Sigma\ \textit{sig} \quad c : A \in \Sigma}{\Gamma \vdash_\Sigma c : A^\beta}\ \textit{const-obj} \qquad \frac{\Gamma \vdash_\Sigma A : \textit{Type} \quad \Gamma, x : A \vdash_\Sigma M : B}{\Gamma \vdash_\Sigma (\lambda x{:}A.M) : (\Pi x{:}A^\beta.B)}\ \textit{abs-obj}$$

$$\frac{\vdash_\Sigma \Gamma\ \textit{ctx} \quad x : A \in \Gamma}{\Gamma \vdash_\Sigma x : A^\beta}\ \textit{var-obj} \qquad \frac{\Gamma \vdash_\Sigma M : \Pi x{:}A.B \quad \Gamma \vdash_\Sigma N : A}{\Gamma \vdash_\Sigma (M\ N) : (B[N/x])^\beta}\ \textit{app-obj}$$

Fig. 1. Rules for typing LF objects

2 Logic programming in LF

Three categories of expressions constitute LF: kinds, type families or types which are classified by kinds, and objects which are classified by types. Below, x denotes an object variable, X an object meta-variable, c an object constant, and a a type constant. Letting K range over kinds, A and B over types, and M and N over objects, the syntax of these expressions is given as follows:

$$\begin{array}{lll} K & ::= & \textit{Type} \mid \Pi x{:}A.K \\ A, B & ::= & a \mid \Pi x{:}A.B \mid A\ M \\ M, N & ::= & c \mid x \mid X \mid \lambda x{:}A.M \mid M\ N \end{array}$$

Both Π and λ are binders which also assign types to the (object) variables they bind over expressions. Notice the dependency present in LF expressions: a bound object variable may appear in a type family or kind. In the remainder of this paper we use U and V ambiguously for types and objects and P similarly for types and kinds. The shorthand $A \to P$ is used for $\Pi x{:}A.P$ if P is a type family or kind that is not dependent on the bound variable, i.e. if x does not appear free in P. Terms differing only in bound variable names are identified. We write $U[M_1/x_1, \ldots, M_n/x_n]$ to denote the capture avoiding substitution of $M_1, \ldots, M_n$ for the free occurrences of $x_1, ..., x_n$ respectively in U.

LF kinds, types and objects are formed relative to a signature Σ that identifies constants together with their kinds or types. In determining if an expression is well-formed, we additionally need to consider contexts, denoted by Γ, that assign types to variables. The syntax for signatures and contexts is as follows:

$$\Sigma \quad ::= \quad \cdot \mid \Sigma, a : K \mid \Sigma, c : A \qquad\qquad \Gamma \quad ::= \quad \cdot \mid \Gamma, x : A$$

In contrast to usual LF presentations, we have allowed expressions to contain object meta-variables. We assume an infinite supply of such variables for each type and that an implicit meta-variable context Δ assigns types to these variables. These meta-variables act as placeholders, representing the part of an expression one wishes to leave unspecified.

Complementing the syntax rules, LF has typing rules that limit the set of acceptable or well-formed expressions. These rules define the following mutually recursive judgments with the associated declarative content:

$\Sigma\ \textit{sig}$	Σ is a valid signature

$\vdash_\Sigma \Gamma\ ctx$	Γ is a valid context relative to the (valid) signature Σ
$\Gamma \vdash_\Sigma K\ kind$	K is a valid kind in signature Σ and context Γ
$\Gamma \vdash_\Sigma A : K$	A is a type of kind K in a signature Σ and context Γ
$\Gamma \vdash_\Sigma M : A$	M is an object of type A in signature Σ and context Γ

In our discussion of logic programming, we rely on a specific knowledge of the rules for only the last of these judgments which we present in Figure 1; an intuition for the other rules should follow from the ones presented and their explicit presentation can be found, e.g., in [4]. By these rules we can see that if a well-formed expression contains a meta- variable X of type A, then replacing the occurrences of X with a well- formed object of type A will produce an expression which is also well-formed.

The rules in Figure 1 make use of an equality notion for LF expressions that is based on β-conversion, i.e., the reflexive and transitive closure of a relation equating two expressions which differ only in that a subexpression of the form $((\lambda x{:}A.M)\ N)$ in one is replaced by $M[N/x]$ in the other. We shall write U^β for the β-normal form of an expression, i.e., for an expression that is equal to U and that does not contain any subexpressions of the form $((\lambda x{:}A.M)\ N)$. Such forms are not guaranteed to exist for all LF expressions. However, they do exist for well-formed LF expressions [4], a property that is ensured to hold for each relevant LF expression by the premises of every rule whose conclusion requires the β-normal form of that expression.

Equality for LF expressions also includes η-conversion, i.e., the congruence generated by the relation that equates $\lambda x{:}A.(M\ x)$ and M if x does not appear free in M. The β-normal forms for the different categories of expressions have the following structure

$Kind$	$\Pi x_1{:}A_1.\ldots\Pi x_n{:}A_n.Type$
$Type$	$\Pi y_1{:}B_1.\ldots\Pi y_n{:}B_m.a\ M_1\ \ldots\ M_n$
$Object$	$\lambda x_1{:}A_1.\ldots\lambda x_n{:}A_n.u\ M_1\ \ldots\ M_n$

where u is an object constant or variable and where the subterms and subtypes appearing in the expression recursively have the same form. We refer to the part corresponding to $a\ M_1\ \ldots\ M_n$ in a type in this form as its *target* type and to $B_1, \ldots, B_m$ as its *argument* types. Let w be a variable or constant which appears in the well-formed term U and let the number of Πs that appear in the prefix of its type or kind in beta normal form be n. We say w is *fully applied* if every occurrence of w in U has the form $w\ M_1 \ldots M_n$. A type of the form $a\ M_1 \ldots M_n$ where a is fully applied is a *base type*. We also say that U is *canonical* if it is in normal form and every occurrence of a variable or constant in it is fully applied. It is a known fact that every well-formed LF expression is equal to one in canonical form by virtue of $\beta\eta$-conversion [4]. For the remainder of this paper we will assume all terms are in β-normal form.

A specification in LF comprises a signature that, as we have seen, identifies a collection of object and type constants. The Curry-Howard isomorphism [6] allows types to be interpreted dually as formulas. The dependent nature of the

```
nat : type.                    list : type.
z : nat.                       nil : list.
s : nat -> nat.                cons : nat -> list -> list.

append : list -> list -> list -> type.
app-nil : append nil L L.
app-cons : append L1 L2 L3 -> append (cons X L1) L2 (cons X L3).
```

Fig. 2. A Twelf signature specifying lists and the append relation

LF type system allows type constants to take objects as arguments. Such constants then correspond to the names of predicates over suitably typed objects. Moreover, the same isomorphism allows object constants, which provide a means for constructing expressions of particular types, to be viewed as the names of parameterized rules for constructing proofs of the relations represented by the types.

Figure 2 presents a concrete signature to illustrate these ideas. In showing this and other similar signatures, we use the Twelf syntax for LF expressions. In this syntax, $\Pi x{:}A.U$ is written as $\{x : A\}\ U$ and $\lambda x{:}A.M$ is written as $[x : A]\ M$. Further, bindings and the corresponding type annotations on variables are made implicit in situations where the types can be uniquely inferred; the variables that are implicitly bound are denoted in Prolog style by tokens that begin with uppercase letters. The initial part of the signature in Figure 2 defines type and object constants that provide a representation of the natural numbers and lists of natural numbers. The signature then identifies a type constant `append` that takes three lists as arguments. Under the viewpoint just explained, this constant can be interpreted as a predicate that relates three lists. Objects of this type can be constructed by using the constants `app-nil` and `app-cons` that are also presented in the signature. Viewed differently, these constants name rules that can be used to construct a proof of the append relation between three lists. Notice that `app-cons` requires as an argument an object of `append` type. This object plays the role of a premise for the rule that `app-cons` identifies.

The logic programming use of LF that underlies Twelf consists of presenting a type A in the setting of a signature Σ. Such a type corresponds to the request to find an object M such that the judgment $\vdash_\Sigma M : A$ is derivable. Alternately, a query in Twelf can be seen as the desire to determine the derivability of a formula, the inhabiting term that is found being its proof. The type that is presented as a query may also contain meta-variables, denoted by tokens that begin with uppercase letters. In this case, the request is to find substitutions for these variables while simultaneously showing that the instance type is inhabited.

An example of a query relative to the signature in Figure 2 is the following.

```
append (cons z nil) nil L
```

An answer to this query is the substitution `(cons z nil)` for `L`, together with the object `(app-cons (cons z nil) nil (cons z nil) (app-nil nil))` that inhabits that type. Another query in this setting is

```
{x:nat} append (cons x nil) (cons z (cons x nil)) (L x).
```

$$\frac{}{\Xi; \Gamma \longrightarrow \top}\ \top R \qquad \frac{\Xi; \Gamma \cup \{D\} \longrightarrow G}{\Xi; \Gamma \longrightarrow D \supset G}\ \supset R \qquad \frac{c \notin \Xi \quad \Xi \cup \{c\}; \Gamma \longrightarrow G[c/x]}{\Xi; \Gamma \longrightarrow \forall x.G}\ \forall R$$

$$\frac{\Xi; \Gamma \longrightarrow G_1[\overrightarrow{t_1/x_1}] \qquad \ldots \qquad \Xi; \Gamma \longrightarrow G_n[\overrightarrow{t_1/x_1}, \ldots, \overrightarrow{t_n/x_n}]}{\Xi; \Gamma \longrightarrow A}\ \textit{backchain}$$

$$\text{where } \forall \overrightarrow{x_1}.(G_1 \supset \ldots \supset \forall \overrightarrow{x_n}.(G_n \supset A') \ldots) \in \Gamma,$$
$$\overrightarrow{t_1}, \ldots, \overrightarrow{t_n} \text{ are } \Xi\text{-terms and } A'[\overrightarrow{t_1/x_1}, \ldots, \overrightarrow{t_n/x_n}] = A$$

Fig. 3. Derivation rules for the *hohh* logic

in which `L` is a “higher-order” meta-variable of type `nat -> list`. The substitution that would be computed by Twelf for the variable `L` in this query is

```
[y:nat] (cons y (cons z (cons y nil))),
```

and the corresponding inhabitant or proof term is

```
[y:nat] app-cons nil (cons z (cons y nil))
                     (cons z (cons y nil)) y
                     (app-nil (cons z (cons y nil)))
```

Notice that the variable `x` that is explicitly bound in the query has a *different* interpretation from the meta-variable `L`. In particular, it receives a “universal” reading: the query represents a request to find a value for `L` that yields an inhabited type regardless of what the value of `x` is.

Although neither of our example queries exhibited this behavior, the range of an answer substitution may itself contain variables and there may be some residual constraints on these variables presented in the form of a collection of equations between object expressions called “disagreement pairs.” The interpretation of such an answer is that a complete solution can be obtained from the provided substitution by instantiating the remaining variables with closed object expressions that render identical the two sides of each disagreement pair.

3 Logic programming based on *hohh*

An alternative approach to specifying formal systems is to use a logic in which relationships between terms are encoded in predicates. The idea of animating a specification then corresponds to constructing a proof for a given “goal” formula in the chosen logic. To yield a sensible notion of computation, specifications must also be able to convey information about how a search for a proof should be conducted. Towards this end, we use here the logic of higher-order hereditary Harrop formulas, referred to in short as the *hohh* logic. This logic underlies the programming language λProlog [8].

The *hohh* logic is based on Church’s Simple Theory of Types [2]. The expressions of this logic are those of a simply typed λ-calculus (STLC). Types are constructed from the atomic type o for propositions and a finite set of other atomic types by using the function type constructor $\rightarrow$. We assume we have been

```
nat : type.           list : type.
z : nat.              nil : list.
s : nat -> nat.       cons : nat -> list -> list.
                      append : list -> list -> list -> o.

∀L. append nil L L.
∀X∀L1∀L2∀L3. append L1 L2 L3 ⊃ append (cons X L1) L2 (cons X L3).
```

Fig. 4. An *hohh* specification of lists and the append relation

given a set of variables and a set of constants, each member of these sets being identified together with a type. More complex terms are constructed from these atomic symbols by using application and λ-abstraction in a way that respects the constraints of typing. As in LF, terms differing only in bound variable names are identified. The notion of equality between terms is further enriched by β- and η-conversion. When we orient these rules and think of them as reductions, we are assured in the simply typed setting of the existence of a unique normal form for every well-formed term under these reductions. Thus, equality between two terms becomes the same as the identity of their normal forms. For simplicity, in the remainder of this paper we will assume that all terms have been converted to normal form. We write $t[s_1/x_1, \ldots, s_n/x_n]$ to denote the capture avoiding substitution of the terms $s_1, \ldots, s_n$ for free occurrences of $x_1, ..., x_n$ in t.

Logic is introduced into this setting by identifying a sub-collection of the set of constants as logical constants and giving them a special meaning. The logical constants that we shall use here are the following:

$\top$ of type o
$\supset$ of type $o \to o \to o$
Π of type $(\tau \to o) \to o$ for each type τ

We intend $\top$ to denote the always true proposition and $\supset$, which we will write in infix form, to denote implication. The symbol Π corresponds to the generalized universal quantifier: the usual notation $\forall x.F$ for universal quantification serves as a shorthand for $\Pi(\lambda x.F)$.

To construct a specification within the *hohh* logic, a user must identify a collection of types and a further set of constants, called non-logical constants, together with their types. A collection of such associations forms a signature. There is a proviso on the types of non-logical constants: their argument types must not contain o. Non-logical constants that have o as their target or result type correspond to predicate symbols. If c is such a constant with the type $\tau_1 \to \ldots \to \tau_n \to o$ and $t_1, \ldots, t_n$ are terms of type $\tau_1, \ldots, \tau_n$, respectively, then the term $(c\ t_1 \ldots\ t_n)$ of type o constitutes an *atomic formula*. We shall use the syntax variable A to denote such formulas. More complex terms of type o are constructed from atomic formulas by using the logical constants. Such terms are also referred to as *formulas*.

The *hohh* logic is based on two special classes of formulas identified by the following syntax rules:

$$G \quad ::= \quad \top \mid A \mid D \supset G \mid \forall x.G \qquad\qquad D \quad ::= \quad A \mid G \supset D \mid \forall x.D$$

$$\begin{aligned} \phi(A) &:= \textit{lf-obj} \text{ when } A \text{ is a base type} \\ \phi(\Pi x{:}A.P) &:= \phi(A) \to \phi(P) \\ \phi(\mathit{Type}) &:= \textit{lf-type} \end{aligned} \qquad \begin{aligned} \langle u \rangle &:= u \\ \langle x \rangle &:= x \\ \langle X \rangle &:= X \\ \langle M_1\ M_2 \rangle &:= \langle M_1 \rangle\ \langle M_2 \rangle \\ \langle \lambda x{:}A.M \rangle &:= \lambda^{\phi(A)} x.\langle M \rangle \end{aligned}$$

Fig. 5. Flattening of types and encoding of terms

We will refer to a D-formula also as a program clause. Notice that, in elaborated form, such a formula has the structure $\forall \overrightarrow{x_1}.(G_1 \supset \ldots \supset \forall \overrightarrow{x_n}.(G_n \supset A)\ldots)$; we write $\forall \overrightarrow{x_i}$ here to denote a sequence of universal quantifications.

The computational interpretation of the *hohh* logic consists of thinking of a collection of D-formulas as a program and a G-formula as a goal or query that is to be solved against a given program $\mathcal{P}$ in the context of a given signature Ξ. We represent the judgment that the query G has a solution in such a setting by the "sequent" $\Xi;\mathcal{P} \longrightarrow G$. The rules for deriving such a judgment are shown in Figure 3. Using these rules to search for a derivation leads to a process in which we first simplify a goal in a manner determined by the logical constants that appear in it and then employ program clauses in a familiar backchaining mode to solve the atomic goals that are produced. A property of the *hohh* logic that should be noted is that both the program and the signature can change in the course of a computation.

We illustrate the use of these ideas in practice by considering, once again, the encoding of lists of natural numbers and the append relation on them. Figure 4 provides both the signature and the program clauses that are needed for this purpose. This specification is similar to one that might be provided in Prolog, except for the use of a curried notation for applications and the fact that the language is now typed. We "execute" these specifications by providing a goal formula. As with Twelf, we will allow goal formulas to contain free or meta-variables for which we intend instantiations to be found through proof search. A concrete example of such a goal relative to the specification in Figure 4 is `(append (cons z nil) nil L)`. This goal is solvable with the substitution `(cons z nil)` for `L`. Another example of a query in this setting is ∀`x.(append (cons x nil) (cons z (cons x nil)) (L x))` and an answer to this goal is the substitution λ`y.(cons y (cons z (cons y nil)))` for `L`.

4 Translating Twelf specifications into predicate form

We now turn to the task of animating Twelf specifications using a λProlog implementation. Towards this end, we describe a meaning preserving translation from LF signatures into *hohh* specifications. Our translation extends the one in [12] by allowing for meta-variables in LF expressions. We also present an inverse translation for bringing solutions back from λProlog to the Twelf setting.

$$\{\!\{\Pi x{:}A.B\}\!\} := \lambda M.\ \forall x.\ (\{\!\{A\}\!\}\ x) \supset (\{\!\{B\}\!\}\ (M\ x))$$
$$\{\!\{A\}\!\} := \lambda M.\ \mathit{hastype}\ M\ \langle A\rangle \text{ where } A \text{ is a base type}$$

Fig. 6. Encoding of LF types using the `hastype` predicate

The first step in our translation is to map dependently typed lambda expressions into simply typed ones. We shall represent both types and objects in LF by STLC terms (which are also *hohh* terms), differentiating the two categories by using the (simple) type *lf-obj* for the encodings of LF objects and *lf-type* for those of LF types. To play this out in detail, we first associate an *hohh* type with each LF type and kind that is given by the $\phi(\cdot)$ mapping shown in Figure 5. Then, corresponding to each object and type-level LF constant $u : P$, we identify an *hohh* constant with the same name but with type $\phi(P)$. Finally, we transform LF objects and kinds into *hohh* terms using the $\langle\cdot\rangle$ mapping in Figure 5.

We would like to consider an inverse to the transformation that we have described above. We have some extra information available in constructing such an inverse: the constants that appear in the *hohh* terms of interest have their correlates which have been given specific types in the originating LF signature. Even so, the lossy nature of the translation makes the inversion questionable. There are two kinds of problems. First, because (the chosen) simple typing is not sufficiently constraining, we may have well-formed STLC terms for which there is no corresponding LF expression. As a concrete example, consider the following LF signature:

```
    i : type     j : type     a : i -> j     c : i
```

In the encoding we will have the following two constants with associated types:

```
    a : lf-obj -> lf-obj                c : lf-obj
```

This means that we can construct the simply typed term `(a (a c))` which cannot be the image of any LF expression that is well-formed under the given signature. The second problem is that when an *hohh* term involves an abstraction, the choice of LF type to use for for the abstracted variable is ambiguous. As a concrete example, consider the *hohh* term $\lambda x.x$ that has the type `lf-obj -> lf-obj`. This term could map to the LF objects `[x:nat] x` and `[x:list] x`, amongst many other choices.

Our solution to these problems is twofold. First, we will assume that we know the type of the LF object that the inversion is to produce; this information will always be available when the *hohh* terms arise in the course of simulating LF typing derivations using *hohh* derivations. Second, we will define inversion as a partial function: when we use it to calculate an LF expression from an answer substitution returned by an *hohh* computation, we will have an additional obligation to show that the inverse must exist.

The rules in Figure 7 define the inverse transformation. The judgments $inv^{\downarrow}(t; A; \Theta) = M$ and $inv^{\uparrow}(t; A; \Theta) = M$ are to be derivable when t is an

$$\frac{X : A \in \Delta}{inv^{\uparrow}(X; A; \Theta) = X}\ \textit{inv-var} \qquad \frac{inv^{\downarrow}(M; B; \Theta, x : A) = M'}{inv^{\downarrow}(\lambda x.M; \Pi x{:}A.B; \Theta) = \lambda x{:}A.M'}\ \textit{inv-abs}$$

$$\frac{inv^{\uparrow}(M_1; \Pi x{:}B.A; \Theta) = M_1' \qquad inv^{\downarrow}(M_2; B; \Theta) = M_2'}{inv^{\uparrow}(M_1\ M_2; A[M_2'/x]; \Theta) = M_1'\ M_2'}\ \textit{inv-app}$$

$$\frac{u : A \in \Theta}{inv^{\uparrow}(u; A; \Theta) = u}\ \textit{inv-const} \qquad \frac{inv^{\uparrow}(M; A; \Theta) = M'}{inv^{\downarrow}(M; A; \Theta) = M'}\ \textit{inv-syn}$$

Fig. 7. An inverse encoding

hohh term in β-normal form that inverts to the LF object M that has type A in a setting where variables and constants are typed according to Θ. The difference between the two judgments is that the first expects A as an input whereas the second additionally synthesizes the type. The process starts with checking against an LF type—this type will be available from the original LF query—and it is easily shown that if $inv^{\downarrow}(t; A; \Sigma \cup \Gamma) = M$, then $\Gamma \vdash_{\Sigma} M : A$. Notice that we will only ever check an abstraction term against an LF type, ensuring that the type chosen for the bound variable will be unique. We say a substitution θ is invertible in a given context and signature if each term in its range is invertible in that setting, using the type associated with the domain variable by Δ.

The translation of LF expressions into *hohh* terms loses all relational information encoded by dependencies in types. For example it transforms the constants encoding the append relation in Figure 2 into the following *hohh* signature:

```
append : lf-obj -> lf-obj -> lf-obj -> lf-type.
app-nil : lf-obj -> lf-obj.
app-cons : lf-obj -> lf-obj ->
           lf-obj -> lf-obj -> lf-obj -> lf-obj.
```

It is no longer possible to construe this as a specification of the append relation between lists. To recover the lost information, we employ a second pass that uses predicates to encode relational content. This pass employs the *hohh* predicate *hastype* with type *lf-obj* $\rightarrow$ *lf-type* $\rightarrow$ *o* and generates clauses that are such that *hastype* X T is derivable from them exactly when X is the encoding of an LF term M of a base LF type whose encoding is T. More specifically, this pass processes each item of the form $U : P$ in the LF signature and produces from it the clause $\{\!\{P\}\!\}\ \langle U \rangle$ using the rules in Figure 6 that define $\{\!\{\cdot\}\!\}$.

To illustrate the second pass, when used with the signature in Figure 2, we see that it will produce the following clauses:

```
hastype z nat.
∀x.hastype x nat ⊃ hastype (s x) nat.
hastype nil list.
∀x.(hastype x nat ⊃
       ∀l.(hastype l list ⊃ hastype (cons x l) list)).

∀l.hastype l list ⊃ hastype (app-nil l) list.
```

```
∀x.(hastype x nat ⊃ ∀l1.(hastype l1 list ⊃
        ∀l2.(hastype l2 list ⊃ ∀l3.(hastype l3 list ⊃
          ∀a.(hastype a (append l1 l2 l3)⊃
            hastype (app-cons x l1 l2 l3 a)
                     (append (cons x l1) l2 (cons x l3))))))).
```

Contrasting these clauses with the ones of the λProlog program in Figure 4, we see that it is capable not only of producing answers to `append` queries but also a "proof-term" that traces the derivation of such queries.

The correctness of our translation is captured by the following theorem (whose proof is currently incomplete). We had said earlier that when looking at terms that are produced by *hohh* derivations from LF translations, we would have an assurance that these terms are invertible. This is a property that flows, in fact, from the structure of the *hastype* clauses: as a *hohh* derivation is constructed, all the substitution terms that are generated are checked to be of the right type using the *hastype* predicate, and so we will not be able to construct a term which is not invertible.

Theorem 1. *Let Σ be an LF signature and let A be an LF type that possibly contains meta-variables.*

1. *If Twelf solves the query $M : A$ with the ground answer substitution σ, then there is an invertible answer substitution θ for the goal $\{\!\{A\}\!\}\ \langle M\rangle$ wrt $\{\!\{\Sigma\}\!\}$ such that the inverse θ' of θ generalizes σ (i.e. there exists a σ' such that $\sigma' \circ \theta' = \sigma$).*
2. *If θ is an invertible answer substitution for $\{\!\{A\}\!\}\ \langle M\rangle$, then its inverse is an answer substitution for $M : A$.*

Our approach to proving this theorem is to consider the operational semantics of the two systems and to show that derivations in each system can be factored into sequences of steps that can be simulated by the other system. Moreover, this simulation ensures the necessary relationships hold between the answer substitutions that are gradually developed by the derivations in the respective systems.

5 Optimizing the translation

The translation presented in the preceding section does not lend itself well to proof search because it generates a large amount of redundant typing checking. There are many instances when this redundancy can be recognized by a direct analysis of a given Twelf specification: in particular, we can use a structural analysis of an LF expression to determine that a term being substituted for a variable must be of the correct type and hence it is unnecessary to check this explicitly. In this section we develop this idea and present an improved translation. We also discuss another optimization that reflect the types in the Twelf signature more directly into types in *hohh*. The combination of these optimizations produce clauses that are more compact and that resemble those that might be written in λProlog directly.

$$\frac{dom(\Gamma); \cdot; x \sqsubset_o A_i \text{ for some } A_i \text{ in } \overrightarrow{A}}{\Gamma; x \sqsubset_t c\overrightarrow{A}}\ \mathrm{APP_t}$$

$$\frac{\Gamma, y : A; x \sqsubset_t B}{\Gamma; x \sqsubset_t \Pi y{:}A.B}\ \mathrm{PI_t} \qquad \frac{\Gamma_1; x \sqsubset_t B \qquad \Gamma_1, y : B, \Gamma_2; y \sqsubset_t A}{\Gamma_1, y : B, \Gamma_2; x \sqsubset_t A}\ \mathrm{CTX_t}$$

$$\frac{y_i \in \delta \text{ for each } y_i \text{ in } \overrightarrow{y} \qquad \text{each variable in } \overrightarrow{y} \text{ is distinct}}{\Delta; \delta; x \sqsubset_o x\ \overrightarrow{y}}\ \mathrm{INIT_o}$$

$$\frac{y \notin \Delta \text{ and } \Delta; \delta; x \sqsubset_o M_i \text{ for some } M_i \text{ in } \overrightarrow{M}}{\Delta; \delta; x \sqsubset_o y\ \overrightarrow{M}}\ \mathrm{APP_o} \qquad \frac{\Delta; \delta, y; x \sqsubset_o M}{\Delta; \delta; x \sqsubset_o \lambda y{:}A.M}\ \mathrm{ABS_o}$$

Fig. 8. Strictly occurring variables in types and objects

We are interested in translating an LF type of the form $\Pi x_1{:}A_1. \ldots \Pi x_n{:}A_n.B$ into an *hohh* clause that can be used to determine if a type B' can be viewed as an instance $B[M_1/x_1, \ldots, M_n/x_n]$ of the target type B. This task also requires us to show that $M_1, \ldots, M_n$ are inhabitants of the types $A_1, \ldots, A_n$; in the naive translation, this job is done by the *hastype* formulas pertaining to x_i and A_i that appear in the body of the *hohh* clause produced for the overall type. However, a particular x_i may occur in B in a manner which already makes it clear that the term M_i which replaces it in any instance of B must possess such a property. What we want to do, then, is characterize such occurrences of x_i such that we can avoid having to include an inhabitation check in the *hohh* clause.

We define a strictness condition for variable occurrences and, hence, for variables that possesses this kind of property. By using this condition, we can simplify the translation of a type into an *hohh* clause without losing accuracy. In addition to efficiency, such a translation also produces a result that bears a much closer resemblance to the LF type from which it originates.

The critical idea behind this criterion is that the path down to the occurrence of x is *rigid*, i.e., it cannot be modified by substitution and x is not applied to arguments in a way that could change the structure of the expression substituted for it. We know that the structure will be unchanged by application of arguments by requiring the occurrence of x to be applied only to distinct λ-bound variables. Thus we know that any term substituted for x has the correct type without needing to explicitly check it. Specifically, we say that the bound variable x_i occurs strictly in the type $\Pi x_1{:}A_1. \ldots \Pi x_n{:}A_n.B$ if it is the case that

$$x_1 : A_1, \ldots, x_{i-1} : A_{i-1}; x_i \sqsubset_t \Pi x_{i+1}{:}A_{i+1}. \ldots \Pi x_n{:}A_n.B$$

holds. We have been able to extend the strictness condition as described in [12] recursively while preserving its utility in recognizing redundancy in type checking. We consider occurrences of bound variables to be strict in the overall type if they are strict in the types of other bound variables that occur strictly in the target type. The relation defined in Figure 8 formalizes this idea.

When $\Gamma; x \sqsubset_t A$ is derivable it means that the variable x appears strictly in the type A in the context Γ. As we work down through the structure of a type we will eventually look at a specific term M and a derivation of $\Delta; \delta; x \sqsubset_o M$ means that x appears strictly in the term M. Here, Δ and δ are both lists of

$$
\begin{aligned}
\phi(a\ M_1 \ldots M_n) &:= \textit{a-type} \\
\phi(\Pi x{:}A.P) &:= \phi(A) \to \phi(P) \\
\phi(\mathit{Type}) &:= \textit{lf-type}
\end{aligned}
\qquad
\begin{aligned}
\langle u \rangle &:= u \\
\langle x \rangle &:= x \\
\langle X \rangle &:= X \\
\langle M_1\ M_2 \rangle &:= \langle M_1 \rangle\ \langle M_2 \rangle \\
\langle \lambda x{:}A.M \rangle &:= \lambda^{\phi(A)} x.\langle M \rangle
\end{aligned}
$$

$$
\begin{aligned}
[\![\Pi x{:}A.B]\!]^+_\Gamma &:= \begin{cases} \lambda M.\ \forall x.\ \top \supset [\![B]\!]^+_{\Gamma,x}(M\ x) & \text{if } \Gamma; x \sqsubset_t B \\ \lambda M.\ \forall x.\ [\![A]\!]^-(x) \supset [\![B]\!]^+_{\Gamma,x}(M\ x) & \text{otherwise} \end{cases} \\
[\![u\ \overrightarrow{N}]\!]^+_\Gamma &:= \lambda M.\ u\ \overrightarrow{\langle N \rangle}\ M \\
[\![\Pi x{:}A.B]\!]^- &:= \lambda M.\ \forall x.\ [\![A]\!]^+_\cdot(x) \supset [\![B]\!]^-(M\ x) \\
[\![u\ \overrightarrow{N}]\!]^- &:= \lambda M.\ u\ \overrightarrow{\langle N \rangle}\ M
\end{aligned}
$$

Fig. 9. Optimized translation of Twelf signatures to λProlog programs

variables where δ contains the λ-bound variables currently in scope, while Δ contains the Π-quantified variables collected while walking through the type A.

Another, more direct, optimization is to reflect the LF types into types in the simply typed lambda calculus. Along with this optimization we can also use specialized predicates, rather than just `hastype`. For each LF type $u : K$ we will create a new atomic type `u-type` in *hohh*, as well as a new predicate `u` which has the type $\phi(K)$ `-> u-type -> o`. We then use these to encode the signature in a more natural way. See Figure 9 for the new translation.

There are now two modes in which translation operates, the negative, $[\![\cdot]\!]^-$, which is essentially the same as before in that it does not check for strictness of bound variables, and the positive, $[\![\cdot]\!]^+$, which will only generate *hastype* formulas for variables which do not appear strictly. We do this to insure that the eliminations occur in situations in which it makes sense to think of the implication encoding an inhabitation check. We will write $\forall x.[\![B]\!]^+_{\Gamma,x}(M\ x)$ for $\forall x.\top \supset [\![B]\!]^+_{\Gamma,x}(M\ x)$ in future to simplify the generated signatures. These optimizations not only clean up the generated signature, but they also improve performance as we have limited the number of clauses which match the head of any given goal formula.

6 An illustration of the translation approach

We illustrate the use of the ideas described in the earlier sections by considering the `append` relation specified in Twelf by the signature in Figure 2. The Twelf query that we shall consider is the following that we previously saw in Section 2:

```
{x:nat} append (cons x nil) (cons z (cons x nil)) (L x).
```

This query asks for a substitution for `L` that yields an inhabited type and an object that is a corresponding inhabitant.

```
nat : nat-type -> o.
list : list-type -> o.
append : list-type -> list-type -> list-type -> append-type -> o.

nat z.
∀x. nat x ⊃ nat (s x).
list nil.
∀x.(nat x ⊃ ∀l. list l ⊃ list (cons x l)).

∀l. append nil l l (app-cons l).
∀x∀l1∀l2∀l3∀a. append l1 l2 l3 a ⊃
    append (cons x l1) l2 (cons x l3) (app-cons x l1 l2 l3 a).
```

Fig. 10. The Twelf specification of `append` translated into λProlog

Applying the optimized translation to the signature in Figure 2 yields the λProlog program shown in Figure 10. Further, the Twelf query of interest translates into the *hohh* goal formula

```
    ∀x. append (cons x nil) (cons z (cons x nil)) (L x) M.
```

The answer substitution for this goal in λProlog is

```
  L = y\ cons y (cons z (cons y nil)),
  M = y\ app-cons nil (cons z (cons y nil))
                 (cons z (cons y nil)) y
                 (app-nil (cons z (cons y nil)))
```

Applying the inverse translation described in Section 4 to this answer substitution yields the value for `L` and the proof term for the Twelf query that we saw in Section 2.

7 Conclusion

We have considered here the possibility of implementing the logic programming treatment of LF specifications that is embodied in Twelf by using the Teyjus implementation of λProlog as a backend. Central to the approach we have examined is a meaning-preserving translation of Twelf specifications into λProlog programs. The basic structure of such a translation has previously been described by Snow et. al. [12]. However, to use our approach in an actual implementation of Twelf, it is necessary to complement the translation with a method for turning solutions found in the λProlog setting into expressions in LF syntax that constitute answers to the original queries. Towards this end, we have described an inverse encoding that maps *hohh* terms back to LF objects that are well-formed with respect to the starting signature. In their translation, Snow et. al. only considered LF expressions that are closed. To capture the full scope of logic programming in Twelf, we have allowed LF types that constitute queries to contain meta-variables and we have provided a treatment for such variables in both the back-and-forth translations and in derivations. Finally, we have formulated a correctness theorem for our approach to implementing Twelf and we have outlined a method for proving this theorem that relates a unification based operational semantics for Twelf and the *hohh* logic. Our ongoing work is directed at completing the proof of the correctness theorem and at obtaining an empirical assessment of our proposed approach by experimenting with an implementation of Twelf that is based on it.

Acknowledgements

This work has been partially supported by the NSF Grant CCF-0917140. Opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. P. Brisset and O. Ridoux. The compilation of λProlog and its execution with MALI. Publication Interne 687, IRISA, 1992.
2. Alonzo Church. A formulation of the simple theory of types. *J. of Symbolic Logic*, 5:56–68, 1940.
3. Amy Felty and Dale Miller. Encoding a dependent-type λ-calculus in a logic programming language. In Mark Stickel, editor, *Proceedings of the 1990 Conference on Automated Deduction*, volume 449 of *LNAI*, pages 221–235. Springer, 1990.
4. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
5. Robert Harper and Daniel R. Licata. Mechanizing metatheory in a logical framework. *Journal of Functional Programming*, 17(4–5):613–673, July 2007.
6. William A. Howard. The formulae-as-type notion of construction, 1969. In J. P. Seldin and R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, New York, 1980.
7. Dale Miller and Gopalan Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
8. Gopalan Nadathur and Dale Miller. An Overview of λProlog. In *Fifth International Logic Programming Conference*, pages 810–827, Seattle, August 1988. MIT Press.
9. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon D. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
10. Frank Pfenning and Carsten Schürmann. *Twelf User's Guide*, 1.4 edition, December 2002.
11. Xiaochu Qi, Andrew Gacek, Steven Holte, Gopalan Nadathur, and Zach Snow. The Teyjus system – version 2, March 2008. http://teyjus.cs.umn.edu/.
12. Zachary Snow, David Baelde, and Gopalan Nadathur. A meta-programming approach to realizing dependently typed logic programming. In *ACM SIGPLAN Conference on Principles and Practice of Declarative Programming (PPDP)*, pages 187–198, 2010.

Towards Pre-Indexed Terms[⋆]

J. F. Morales[1] and M. Hermenegildo[1,2]

[1] IMDEA Software Research Insitute, Madrid, Spain
[2] School of Computer Science, Technical University of Madrid, Spain

Abstract. Indexing of terms and clauses is a well-known technique used in Prolog implementations (as well as automated theorem provers) to speed up search. In this paper we show how the same mechanism can be used to implement efficient reversible mappings between different term representations, which we call *pre-indexing*s. Based on user-provided term descriptions, these mappings allow us to use more efficient data encodings internally, such as prefix trees. We show that for some classes of programs, we can drastically improve the efficiency by applying such mappings at selected program points.

1 Introduction

Terms are the most important data type for languages and systems based on first-order logic, such as (constraint) logic programming or resolution-based automated theorem provers. Terms are inductively defined as variables, atoms, numbers, and compound terms (or structures) comprised by a functor and a sequence of terms.[3] Two main representations for Prolog terms have been proposed. Early Prolog systems, such as the Marseille and DEC-10 implementations, used *structure sharing* [2], while the WAM [13,1] –and consequently most modern Prolog implementations– uses *structure copying*. In structure sharing, terms are represented as a pair of pointers, one for the structure skeleton, which is shared among several instances, and another for the binding environment, which determines a particular instantiation. In contrast, structure copying makes a copy of the structure for each newly created term. The encoding of terms in memory resembles tree-like data structures.

In order to speed up resolution, sophisticated term indexing has been implemented both in Prolog [1,8] and automated theorem provers [7]. By using specialized data structures (such as, e.g., tries), indexing achieves sub-linear complexity in clause selection. Similar techniques are used to efficiently store predicate solutions in tabling [11]. This efficient machinery for indexing is often

[⋆] Research supported in part by projects EU FP7 318337 *ENTRA*, Spanish MINECO TIN2012-39391 *StrongSoft* and TIN2008-05624 *DOVES*, and Comunidad de Madrid TIC/1465 *PROMETIDOS-CM*.

[3] Additionally, many Prolog systems implement an extension mechanism for variable domains using *attributed variables*.

attractive for storing and manipulating program data, such as dynamic predicates. Indexed dynamic predicates offer the benefits of efficient key-value data structures while hiding the implementation details from the user program.

Modulo some issues like variable sharing, there is thus a duality in programming style between *explicitly* encoding data as terms or encoding data *implicitly* as tuples in dynamic predicates. However, although both alternatives have some *declarative* flavor, it is also frequent to find code where, for performance reasons, the data is represented in the end in a quite unnatural way. E.g., the set $\{1, 2, 3, \ldots, n\}$ can be represented naturally as the term `[1,2,3,...,n]` (equivalent to a linked list). However, depending on the lifetime and operations to be performed on the data, binary trees, some other map-like structure, or dynamic predicates may be preferable. These changes in representation often propagate through the whole program.

The goal of this paper is to study the merits of term indexing during term creation rather than at clause selection time. We exploit the fact that data has frequently a fixed skeleton structure, and introduce a mapping in order to index and share that part. This mapping is derived from program declarations specifying term encoding (called *rtype*s, for r*epresentation types*) and annotations defining the program points where *pre-indexing* of terms is performed. This is done on top of structure copying, so that no large changes are required in a typical Prolog runtime system. Moreover, the approach does not require large changes in program structure, which makes *rtype*s easily interchangeable.

We have implemented a prototype as a Ciao package that deals with *rtype* declarations as well as with some additional syntactic sugar that we provide for marking pre-indexing points. We leave as future work the automatic selection of encoding decisions based on profiling and more detailed cost models.

2 Background

We follow the definitions and naming conventions for *term indexing* of [4,7]. Given a set of terms $\mathcal{L}$ (the *indexed terms*), a binary relation R over terms (the *retrieval condition*), and a term t (the *query term*), we want to identify the subset $\mathcal{M} \subseteq \mathcal{L}$ consisting of all the terms l such that $R(l, t)$ holds (i.e., such that l is R-compatible with t). We are interested in the following retrieval conditions R (where σ is a substitution):

– $\mathit{unif}(l, t) \Leftrightarrow \exists\sigma\ l\sigma = t\sigma$ (unification)
– $\mathit{inst}(l, t) \Leftrightarrow \exists\sigma\ l = t\sigma$ (instance check)
– $\mathit{gen}(l, t) \Leftrightarrow \exists\sigma\ l\sigma = t$ (generalization check)
– $\mathit{variant}(l, t) \Leftrightarrow \exists\sigma\ l\sigma = t$ and σ is a renaming substitution (variant check)

Example 1. Given $L = \{h(f(A)), h(f(B, C)), h(g(D))\}$, $t = h(f(1))$, and $R = \mathit{unif}$, then $M = \{h(f(A))\}$.

The objective of *term indexing* is to implement fast retrieval of candidate terms. This is done by processing the indexed set $\mathcal{L}$ into specialized data structures (*index construction*) and modifying this index when terms are inserted or deleted from $\mathcal{L}$ (*index maintenance*).

When the retrieval condition makes use of the function symbols in the query and indexed terms, it is called *function symbol based indexing.*

In Prolog, indexing finds the set of program clauses such that their heads unify with a given literal in the goal. In tabled logic programming, this is also interesting for detecting if a new goal is a variant or subsumed by a previously evaluated subgoal [6,10].

Limitations of indexing. Depending on the part of the terms that is indexed and the supporting data structure, the worst case cost of indexing is proportional to the size of the term. When computing hash keys, the whole term needs to be traversed (e.g., computing the key for `h(f(A))` requires walking over `h` and `f`). This may be prohibitively costly, not only in the maintenance of the indices, but also in the lookup. As a compromise many systems rely only on first argument, first level indexing (with constant hash table lookup, relying on linear search for the selected clauses). However, when the application needs stronger, multi-level indexing, lookup costs are repeated many times for each clause selection operation.

3 Pre-indexing

The goal of pre-indexing is to move lookup costs to term building time. The idea that we propose herein is to use a bijective mapping between the standard and the pre-indexed representations of terms, at selected program points. The fact that terms can be partially instantiated brings in a practical problem, since bounding a variable may affect many precomputed indices (e.g., precomputed indices for `H=h(X)`, `G=g(X)` may need a change after `X=1`). Our proposed solution to this problem is to restrict the mapping to terms of a specific form, based on (herein, user-provided) *instantiation types.*

Definition 1 (Instantiation type). *We say that t is an instance of an* instantiation type τ *(defined as a unary predicate), written as check*$(\tau(t))$*, if there exists a term l such that $\tau(l)$ is in the model of τ and gen*(l, t) *(or inst*(t, l)*).*

For conciseness, we will describe the restricted form of instantiation types used herein using a specialized syntax: [4]

```
  :- rtype lst ---> [] ; [any|lst]
```

In these rules `any` represents any term or variable while `nv` represents any `nonvar` term. The rule above thus corresponds to the predicate:

```
  lst([]).
  lst([_|Xs]) :- lst(Xs).
```

[4] Despite the syntax being similar to that described in [9], note that the semantics is not equivalent.

Example 2. According to the definition above for `lst`, the terms `[1,2,3]` and `[_,2]` belong to `lst` while `[1|_]` does not. If `nv` were used instead of `any` in the definition above then `[_,2]` would also not belong to `lst`.

Type-based pre-indexing. The idea behind pre-indexing is to maintain specialized indexing structures for each *rtype* (which in this work is done based on user annotations). Conceptually, the indexing structure will keep track of all the *rtype* inhabitants dynamically, assigning a unique identifier (the pre-index key) to each of them. E.g., for `lst` we could assign $\{$`[]`$\mapsto k_0$, `[_]`$\mapsto k_1$, `[_,_]`$\mapsto k_2, \ldots\}$.

Translation between pre-indexed and non-pre-indexed forms is defined in terms of a *pre-indexing casting*. Given $check(\tau(t))$, $\exists\ l \in |\tau|$ (set of "weakest" terms for which τ holds) such that $gen(l, t)$.

Definition 2 (Pre-indexing cast). *A* pre-indexing cast *of type τ is a bijective mapping between terms, denoted by $\#\tau$, with the following properties:*

- *For every term x and substitution σ so that $check(\tau(x))$, then $\#\tau(x\sigma) = \#\tau(x)\sigma$ (σ-commutative), and*
- *the first-level functor of $\#\tau(x)$* encodes *the structure of the arguments (so that it uniquely identifies the* rtype *inhabitant).*

Informally, the first property ensures that pre-indexing casts can be selectively introduced in a program without altering the (substitution) semantics. Moreover, the meaning of many built-ins is also preserved after pre-indexing, as expressed in the following theorem.

Theorem 1 (Built-in homomorphism). *Given $check(\tau(x))$ and $check(\tau(y))$, then $unif(x, y) \Leftrightarrow unif(\#\tau(x), \#\tau(y))$ (equivalently for gen, inst, variant, and other built-ins like `==/2`, `ground/1`).*

Proof. $unif(x, y) \Leftrightarrow$ [def. of unif] $\exists\sigma\ x\sigma = y\sigma$. Since $\#\tau$ is bijective, then $\#\tau(x\sigma) = \#\tau(y\sigma) \Leftrightarrow$ [σ-commutative] $\#\tau(x)\sigma = \#\tau(y)\sigma$. Given the def. of *unif*, it follows that unif$(\#\tau(x), \#\tau(y))$. The proofs for other built-ins are similar.

In this work we do not require the semantics of built-ins like `@<` (i.e., *term ordering*) to be preserved, but if desired this can be achieved by selecting carefully the order of keys in the pre-indexed term. Similarly, functor arity in principle will not be preserved since ground arguments that are part of the *rtype* structure are allowed to be removed.

3.1 Building pre-indexed terms

We are interested in building terms directly into their pre-indexed form. To achieve this we take inspiration from WAM compilation. Complex terms in variable-term unifications are decomposed into simple variable-structure unifications $X = f(A_1, \ldots, A_n)$ where all the A_i are variables. In WAM bytecode, this is further decomposed into a `put_str` f/n (or `get_str` f/n) instruction followed by a sequence of `unify_arg` A_i. These instructions can be expressed as follows:

```
put_str(X,F/N,S0,S1),    % | F/N |
unify_arg(A1,S1,S2)      % | F/N | A1 |
...
unify_arg(An,Sn,S)       % | F/N | A1 | ... | An |
```

where the S_i represent each intermediate heap state, which is illustrated in the comments on the right.

Assume that each argument A_i can be split into its indexed part A_ik and its value part A_iv (which may omit information present in the key). *Pre-indexing* builds terms that encode A_ik into the main functor:

```
g_put_str(X,F/N,S0,S1),    % | F/N |
g_unify_arg(A1,S1,S2)      % | F/N<A1k> | A1v |
...
g_unify_arg(An,Sn,S)       % | F/N<A1k,...,Ank> | A1v | ... | Anv |
```

The *rtype* constructor annotations (that we will see in Section 3.2) indicate how the functor and arguments are indexed.

Cost analysis. Building and unifying pre-indexed terms have impact both on performance and memory usage. First, regarding time, although pre-indexing operations can be slower, clause selection becomes faster, as it avoids repetitive lookups on the fixed structure of terms. In the best case, $O(n)$ lookups (where n is the size of the term) become $O(1)$. Other operations like unification are sped-up (e.g., earlier failure if keys are different). Second, pre-indexing has an impact on memory usage. Exploiting the data structure allows more compact representations, e.g., `bitpair(bool,bool)` can be assigned an integer as key (without storage costs). In other cases, the supporting *index structures* may effectively share the common part of terms (at the cost of maintaining those structures).

3.2 Pre-indexing Methods

Pre-indexing is enabled in an *rtype* by annotating each constructor with modifiers that specify the *indexing method*. Currently we support compact trie-like representations and packaged integer encodings.

Trie representation is specified with the `index(Args)` modifier, which indicates the order in which arguments are walked in the decision-tree. The process is similar to term creation in the heap, but instead of moving a heap pointer, we combine it with walking through a trie of nodes. Keys are retrieved from the term part that corresponds to the *rtype* structure.

For example, let us consider the input set of terms $[a(x), c(z)]$, $[a(x), d(w)]$, $[b(y), c(z)]$, $[b(y), d(w)]$, where a, b, c, d are function symbols and x, y, z, w are

5

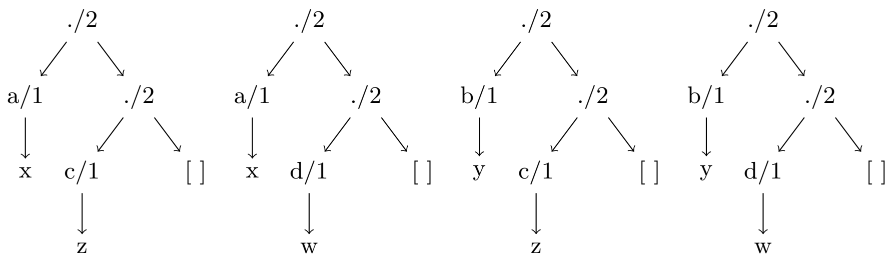


Fig. 1. Example terms for pre-indexing

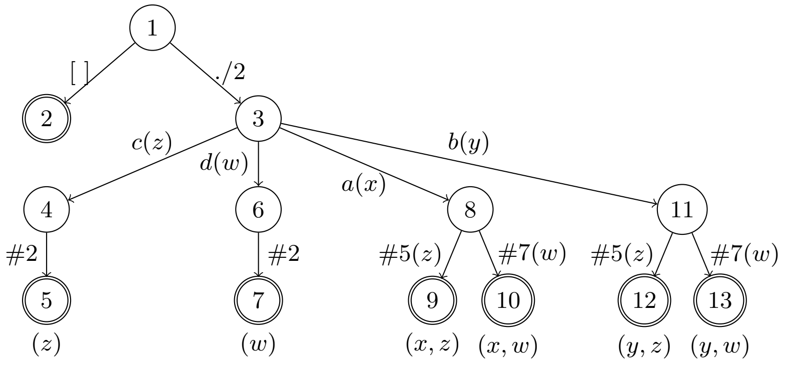


Fig. 2. Index for example terms (*rtype* `lst ---> [] ; [nv|lst]:::index([0,1,2])`)

variable symbols. The heap representation is shown in Fig. 1.[5] We will compare different *rtype* definitions for representing these terms.

As mentioned before, `nv` represents the *rtype* for any `nonvar` term (where its first level functor is part of the type). The declaration:

```
:- rtype lst ---> [] ; [nv|lst]:::index([0,1,2]).
```

specifies that the lookup order for `[_|_]` is a) the constructor name (*./2*), b) the first argument (not pre-indexed), and c) the second argument (pre-indexed). The resulting trie is in Fig. 2. In the figure, each node number represents a position in the trie. Singly circled nodes are temporary nodes, doubly circled nodes are final nodes. Final nodes encode terms. The initial node (*#1*) is unique for each *rtype*. Labels between nodes indicate the lookup input. They can be constructor names (e.g., *./2*), `nv` terms (e.g., *b(y)*), or other pre-indexed `lst` (e.g., *#2* for *[]*, or *#5(z)* for *[c(z)]*). The arguments are placeholders for the non-indexed information. That is, a term *[a(g),c(h)]* would be encoded as *#9(g,h)*.

Trie indexing also supports *anchoring* on non-root nodes. Consider this declaration:

[5] Remember that `[1,2] = .(1,.(2,[]))`.

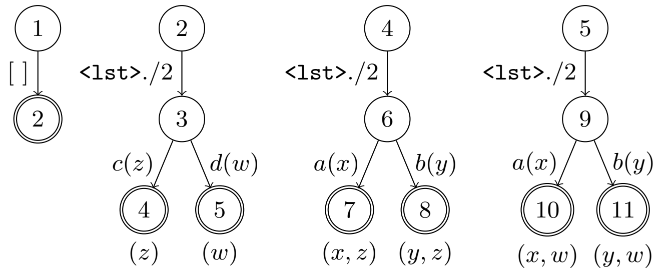


Fig. 3. Index for example terms (*rtype* `lst ---> [] ; [nv|lst]:::index([2,0,1])`)

```
:- rtype lst ---> [] ; [nv|lst]:::index([2,0,1]).
```

Figure 3 shows the resulting trie. The lookup now starts from the second argument, then the constructor name, and finally the first argument. The main difference w.r.t. the previous indexing method is that the beginning node is another pre-indexed term. This may lead to more optimal memory layouts and need fewer lookup operations. Note that constructor names in the edges from initial nodes need to be prefixed with the name of the *rtype*. This is necessary to avoid ambiguities, since the initial node is no longer unique.

Garbage Collection and Indexing Methods. Indexing structures require special treatment for garbage collection.[6] In principle, it would not be necessary to keep in a trie nodes for terms that are no longer reachable (e.g., from the heap, WAM registers, or dynamic predicates), except for caching to speed-up node creation. Node removal may make use of lookup order. That is, if a key at a temporary level n corresponds to an atom that is no longer reachable, then all nodes above n can be safely discarded.

Anchoring on non-root nodes allows the simulation of interesting memory layouts. For example, a simple way to encode objects in Prolog is by introducing a *new object* operation that creates new fresh atoms, and storing object attributes with a dynamic `objattr(ObjId, AttrName, AttrValue)` predicate. Anchoring on `ObjId` allows fast deletion (at the implementation level) of all attributes of a specific object when it becomes unreachable.

4 Applications and Experimental Evaluation

To show the feasibility of the approach, we have implemented the pre-indexing transformations as source-to-source transformations within the Ciao system. This is done within a Ciao package which defines the syntax and processes the *rtype* declarations as well as the marking of pre-indexing points.

[6] Automatic garbage collection of indexing structures is not supported in the current implementation.

```
compress(Cs, Result) :-              % Compress Cs
    build_dict(256),                 % Build the dictionary
    compress_(Cs, #lst([]), Result).

compress_([], W, [I]) :-             % Empty, output code for W
    dict(W,I).
compress_([C|Cs], W, Result) :-      % Compress C
    WC = #lst([C|^W]),
    ( dict(WC,_) ->                  % WC is in dictionary
        W2 = WC,
        Result = Result0
    ; dict(W,I),                     % WC not in dictionary
      Result = [I|Result0],          % Output the code for W
      insert(WC),                    % Add WC to the dictionary
      W2 = #lst([C])
    ),
    compress_(Cs, W2, Result0).
```

Fig. 4. LZW Compression: Main code.

As examples, we show algorithmically efficient implementations of the Lempel-Ziv-Welch (LZW) lossless data compression algorithm and the Floyd-Warshall algorithm for finding the shortest paths in a weighted graph, as well as some considerations regarding supporting module system implementation. In the following code, `forall/2` is defined as `\+ (Cond, \+ Goal)`.

4.1 Lempel-Ziv-Welch compression

Lempel-Ziv-Welch (LZW) [14] is a lossless data compression algorithm. It encodes an input string by building an indexed dictionary D of words and writing a list of dictionary indices, as follows:

1- $D := \{w \mid w \text{ has length } 1\}$ (all strings of length one).
2- Remove from input the longest prefix that matches some word W in D, and emit its dictionary index.
3- Read new character C, $D := D \cup \mathsf{concat}(W, C)$, go to step 2; otherwise, stop.

A simple Prolog implementation is shown in Fig. 4 and Fig. 5. Our implementation uses a *dynamic* predicate `dict/2` to store words and corresponding numeric indices (for output). Step 1 is implemented in the `build_dict/1` predicate. Steps 2 and 3 are implemented in the `compress_/3` predicate. For encoding words we use lists. We are only interested in adding new characters and word matching. For that, list construction and unification are good enough. We keep words in reverse order so that appending a character is done in constant time. For constant-time matching, we use an *rtype* for pre-indexing lists. The implementation is straighforward. Note that we add a character to a word in `WC =`

```
 1  % Mapping between words and dictionary index
 2  :- data dict/2.
 3
 4  % NOTE: #lst can be changed or removed, ^ escapes cast
 5  % Anchors to 2nd arg in constructor
 6  :- rtype lst ---> [] ; [int|lst]:::index([2,0,1]).
 7
 8  build_dict(Size) :-                   % Initial dictionary
 9      assertz(dictsize(Size)),
10      Size1 is Size - 1,
11      forall(between(0, Size1, I),      % Single code entry for I
12         assertz(dict(#lst([I]), I))).
13
14  insert(W) :-                          % Add W to the dictionary
15      retract(dictsize(Size)), Size1 is Size + 1, assertz(dictsize(Size1)),
16      assertz(dict(W, Size)).
```

Fig. 5. LZW Compression: Auxiliary code and *rtype* definition for words.

	data size		indexing (time)		
	original	result	none	clause	term
data1	1326	732	0.074	0.025	0.015
data2	83101	20340	49.350	1.231	0.458
data3	149117	18859	93.178	2.566	0.524

Table 1. Performance of LZW compression (in seconds) by indexing method.

`#lst([C|^W])` (Line 8). The annotation indicates that words are pre-indexed using the `lst` *rtype* and that `W` is already pre-indexed (indicated by the escape `^` prefix). Thus we can effectively obtain optimal algorithmic complexity.

Performance evaluation. We have encoded three files of different format and size (two HTML files and a Ciao bytecode object) and measured the performance of alternative indexing and pre-indexing options. The experimental results for the algorithm implementation are shown in Table 1.[7] The columns under *indexing* show the execution time in seconds for different indexing methods: *none* indicates that no indexing is used (except for the default first argument, first level indexing); *clause* performs multi-level indexing on `dict/2`; *term* uses pre-indexed terms.

Clearly, disabling indexing performs badly as the number of entries in the dictionary grows, since it requires one linear (w.r.t. the dictionary size) lookup operation for each input code. Clause indexing reduces lookup complexity and shows a much improved performance. Still, the cost has a linear factor w.r.t. the

[7] Despite the simplicity of the implementation, we obtain compression rates similar to `gzip`.

word size. Term pre-indexing is the faster implementation, since the linear factor has disappeared (each word is uniquely represented by a trie node).

4.2 Floyd-Warshall

```
floyd_warshall :-
    % Initialize distance between all vertices to infinity
    forall((vertex(I), vertex(J)), assertz(dist(I,J,1000000))),
    % Set the distance from V to V to 0
    forall(vertex(V), set_dist(V,V,0)),
    forall(weight(U,V,W), set_dist(U,V,W)),
    forall((vertex(K), vertex(I), vertex(J)),
        (dist(I,K,D1),
         dist(K,J,D2),
         D12 is D1 + D2,
         mindist(I,J,D12))).

mindist(I,J,D) :- dist(I,J,OldD), ( D < OldD -> set_dist(I,J,D) ; true ).

set_dist(U,V,W) :- retract(dist(U,V,_)), assertz(dist(U,V,W)).
```

Fig. 6. Floyd-Warshall Code

The Floyd-Warshall algorithm computes the shortest paths problem in a weighted graph in $O(n^3)$ time, where n is the number of vertices. Let $G = (V, E)$ be a weighted directed graph, $V = v_1, \ldots, v_n$ the set of vertices, $E \subseteq V^2$, and $w_{i,j}$ the weight associated to edge (v_i, v_j) (where $w_{i,j} = \infty$ if $(v_i, v_j) \notin E$ and $w_{i,i} = 0$). The algorithm is based on incrementally updating an estimate on the shortest path between each pair of vertices until the result is optimal. Figure 6 shows a simple Prolog implementation. The code uses a dynamic predicate `dist/3` to store the computed minimal distance between each pair of vertices. For each vertex k, the distance between each (i, j) is updated with the minimum distance calculated so far.

Performance evaluation. The performance of our Floyd-Warshall implementation for different sizes of graphs is shown in Fig. 7. We consider three indexing methods for the `dist/3` predicate: *def* uses the default first order argument indexing, *t12* computes the vertex pair key using two-level indices, *p12* uses a packed integer representation (obtaining a single integer representation for the pair of vertices, which is used as key), and *p12a* combines *p12* with a specialized array to store the `dist/3` clauses. The execution times are consistent with the expected algoritmic complexity, except for *def*. The linear relative factor with the rest of methods indicates that the complexity without proper indexing is $O(n^4)$.

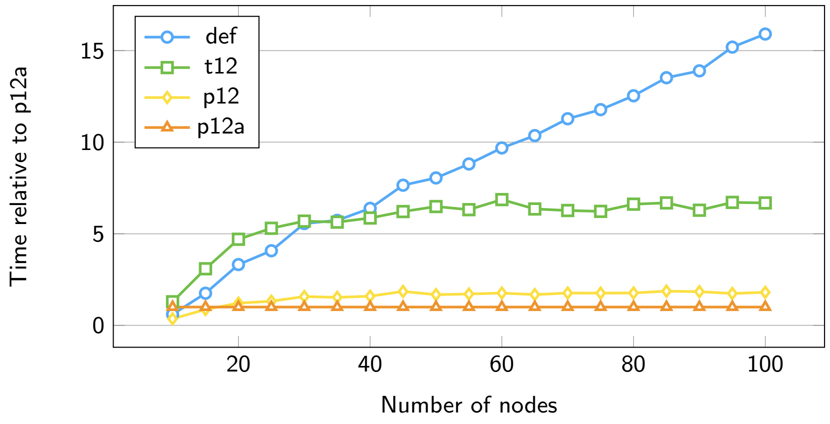


Fig. 7. Execution time for Floyd-Warshall

On the other hand, the plots also show that specialized computation of keys and data storage (*p12* and *p12a*) outperforms more generic encoding solutions (*t12*).

4.3 Module System Implementations

Module systems add the notion of modules (as separate namespaces) to predicates or terms, together with visibility and encapsulation rules. This adds a significantly complex layer on top of the program database (whether implemented in C or in Prolog meta-logic as hidden tables, as in Ciao [5]). Nevertheless, almost no changes are required in the underlying emulator machinery or program semantics. Modular terms and goals can be perfectly represented as `M:T` terms and a program transformation can systematically introduce `M` from the context. However, this would include a noticeable overhead. To solve this issue, Ciao reserves special atom names for module-qualified terms (currently, only predicates).

We can see this optimization as a particular case of pre-indexing, where the last step in module resolution (which maps to the internal representation) is a pre-indexing cast for an `mpred` *rtype*:

```
:- rtype mpred ---> nv:nv ::: index([1,0,2]).
```

For example, given a module `M = lists` and goal `G = append(X,Y,Z)`, the pre-indexed term `MG = #mpred(M:G)` can be represented as `'lists:append'(X,Y,Z)`,[8] where the first functor encodes both the module and the predicate name. To enable meta-programming, when `MG` is provided, both `M` and `G` can be recovered.

Internally, another rewrite step replaces predicate symbols by actual pointers in the bytecode, which removes yet another indirection step. This indicates that

[8] Note that the identifier does not need any symbolic description in practice.

it would be simple to reuse pre-indexing machinery for module system implementations, e.g., to enhance modules with hierarchies or provide better tools for meta-programming. In principle, pre-indexing would bring the advantages of efficient low-level code with the flexibility of Prolog-level meta representation of modules. Moreover, anchoring on `M` mimicks a memory layout where predicate tables are stored as key-value tables inside module data structures.

5 Related Work

There has been much previous work on improving indexing for Prolog and logic programming. Certain applications involving large data sets need any- and multi-argument indexing. In [3] an alternative to static generation of multi-argument indexing is presented. The approach presented uses dynamic schemes for demand-driven indexing of Prolog clauses. In [12] a new extension to Prolog indexing is proposed. User-defined indexing allows the programmer to index both instantiated and constrained variables. It is used for range queries and spatial queries, and allows orders of magnitude speedups on non-trivial datasets.

Also related is ground-hashing for tabling, studied in [15]. This technique avoids storing the same ground term more than once in the table area, based on computation of hash codes. The approach proposed adds an extra cell to every compound term to memoize the hash code and avoid the extra linear time factor.

Our work relates indexing techniques (which deal with fast lookup of terms in collections) with term representation and encoding (which clearly benefits from specialization). Both problems are related with optimal data structure implementation. Prolog code is very often used for prototyping and then translated to (low-level) imperative languages (such as C or C++) if scalability problems arise. This is however a symptom that the emulator and runtime are using suboptimal data structures which add unnecessary complexity factors. Many specialized data structures exist in the literature, with no clear winner in all cases. If they can be directly implemented in Prolog, they are often less efficient than their low-level counterparts (e.g., due to data immutability). Without proper abstraction they obscure the program to the point where a low-level implementation may not be more complex. On the other hand, adding them to the underlying Prolog machines is not trivial. Even supporting more than one term representation may have prohibitive costs (e.g., efficient implementations require a low number of tags, small code that fits in the instruction cache, etc.). Our work aims at reusing the indexing machinery when possible and specializing indexing for particular programs.

6 Conclusions and Future Work

Traditionally, Prolog systems index terms during clause selection (in the best case, reducing a linear search to constant time). Despite that, index lookup is proportional to the size of the term. In this paper we have proposed a mixed approach where indexing is precomputed during term creation. To do that, we define a notion of instantiation types and annotated constructors that specify the indexing mode. The advantage of this approach is that lookups become sub-linear. We have shown experimentally that this approach improves clause indexing and that it has other applications, for example for module system implementation.

These results suggest that it may be interesting to explore lower-level indexing primitives beyond clause indexing. This work is also connected with structure sharing. In general, pre-indexing annotations allow the optimization of simple Prolog programs with scalability problems due to data representation.

As future work, there are some open lines. First, we plan to polish the current implementation, which is mostly based on program rewriting and lacks garbage collection of indexing tables. We expect major performance gains by optimizing some operations at the WAM or C level. Second, we want to extend our repertoire of indexing methods and supporting data structures. Finally, *rtype* declarations and annotations could be discovered and introduced automatically via program analysis or profiling (with heuristics based on cost models).

References

1. Ait-Kaci, H.: Warren's Abstract Machine, A Tutorial Reconstruction. MIT Press (1991)
2. Boyer, R., More, J.: The sharing of structure in theorem-proving programs. Machine Intelligence 7 pp. 101–116 (1972)
3. Costa, V.S., Sagonas, K.F., Lopes, R.: Demand-driven indexing of prolog clauses. In: Dahl, V., Niemelä, I. (eds.) ICLP. Lecture Notes in Computer Science, vol. 4670, pp. 395–409. Springer (2007)
4. Graf, P.: Term Indexing, Lecture Notes in Computer Science, vol. 1053. Springer (1996)
5. Hermenegildo, M.V., Bueno, F., Carro, M., López, P., Mera, E., Morales, J., Puebla, G.: An Overview of Ciao and its Design Philosophy. Theory and Practice of Logic Programming 12(1–2), 219–252 (January 2012), http://arxiv.org/abs/1102.5497
6. Johnson, E., Ramakrishnan, C., Ramakrishnan, I., Rao, P.: A space efficient engine for subsumption-based tabled evaluation of logic programs. In: Middeldorp, A., Sato, T. (eds.) Functional and Logic Programming, Lecture Notes in Computer Science, vol. 1722, pp. 284–299. Springer Berlin / Heidelberg (1999)
7. Ramakrishnan, I.V., Sekar, R.C., Voronkov, A.: Term indexing. In: Robinson, J.A., Voronkov, A. (eds.) Handbook of Automated Reasoning, pp. 1853–1964. Elsevier and MIT Press (2001)
8. Santos-Costa, V., Sagonas, K., Lopes, R.: Demand-Driven Indexing of Prolog Clauses . In: International Conference on Logic Programming. LNCS, vol. 4670, pp. 395–409. Springer Verlag (2007)
9. Schrijvers, T., Costa, V.S., Wielemaker, J., Demoen, B.: Towards Typed Prolog. In: Pontelli, E., de la Banda, M.M.G. (eds.) International Conference on Logic Programming. pp. 693–697. No. 5366 in LNCS, Springer Verlag (December 2008)
10. Swift, T., Warren, D.S.: Tabling with answer subsumption: Implementation, applications and performance. In: Janhunen, T., Niemelä, I. (eds.) JELIA. Lecture Notes in Computer Science, vol. 6341, pp. 300–312. Springer (2010)

11. Swift, T., Warren, D.S.: Xsb: Extending prolog with tabled logic programming. TPLP 12(1-2), 157–187 (2012)
12. Vaz, D., Costa, V.S., Ferreira, M.: User defined indexing. In: Hill, P.M., Warren, D.S. (eds.) ICLP. Lecture Notes in Computer Science, vol. 5649, pp. 372–386. Springer (2009)
13. Warren, D.H.D.: An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA 94025 (1983)
14. Welch, T.A.: A technique for high-performance data compression. IEEE Computer 17(6), 8–19 (1984)
15. Zhou, N.F., Have, C.T.: Efficient tabling of structured data with enhanced hash-consing. TPLP 12(4-5), 547–563 (2012)

A System for Embedding Global Constraints into SAT

Md Solimul Chowdhury and Jia-Huai You

Department of Computing Science
University of Alberta

Abstract. Propositional satisfiability (SAT) and Constraint Programming (CP) are two dominant tools for solving combinatoral search problems. Both of these has their own strengths and weaknesses. In this report, we describe a tight integration of SAT with CP, called SAT(gc), which embeds global constraints into SAT. A system named SATCP is implemented by integrating the DPLL based SAT solver zchaff and the generic constraint solver gecode. Experiments are carried out for benchmarks from puzzle domains and planning domains to reveal insights in compact representation, solving effectiveness, and novel usability of the new framework. We highlight some issues with the current implementation of SATCP, with possible directions to resolve those issues.

Keywords: SAT, CSP, Global Constraints, Integration, Embedding.

1 Introduction

Constraint Programming (CP) [15] is a programming paradigm, developed for studying and solving constraint problems. It has been applied to solving many practical problems from domains of scheduling, planning, and verification [20]. For practical applications, languages for CP have been developed to facilitate the definitions of constraints in terms of primitive constraints and built-in constraints. One kind of these built-in constraints are called *global constraints* [19]. The use of global constraints not only facilitate problem representation, but also have very efficient implementation based on special data structures and dedicated constraint propagation mechanisms (see, for example, [3]).

Another way of solving combinatorial search problems is Boolean Satisfiability (SAT), in which a problem is represented by a collection of Boolean clauses, called a *formula*. To solve a SAT formula, we need to determine whether there is a truth value assignment that satisfies all the clauses.

In recent years, cross fertilization of these two areas has become a topic of interest. It is argued that complex real world applications may require effective features of both [4]. A number of approaches have been pursued in this direction. For example, in SAT Modulo Theory (SMT) [13], theory solvers of various kinds are incorporated into a SAT solver, where part of the problem is encoded in an embedded theory and solved by a dedicated theory solver. To deal with numeric constraints, the SAT community has moved to a different direction - *pseudo Boolean constraints*, where constraints are expressed by linear inequalities over sum of weighted Boolean functions (see, e.g., [5]). The paper [14] presents a tight integration of SAT and CP, where CSP propagators are emulated as learned SAT clauses. In [9], a framework for integrating CSP style constraint solving in Answer Set Programming (ASP) has been developed, employing an evaluation strategy similar to the lazy SMT approach. In [8], the authors propose a translational approach to constraint answer set programs and show its effectiveness. The usefulness of combining ASP and CP to industrial sized problems is demonstrated in [1].

We pursue a tight integration of SAT and CSP which implies tight interleaves between SAT and CSP solver. Tight integration poses a number of challenging issues like, how to represent a SAT problem in the presence of global constraints, and how deductions, conflict analysis and backtracking with learning can be performed in the presence of global constraints. In our work, we develop a framework to incorporate CSP style constraint solving into SAT solving with the anticipation that this tight integration will enhance the usability of SAT solver and increase its efficiency for some application domains. The report presented here is an extended version of [7], providing more details on implementation, some system level problems and possible solutions.

The rest of this report is organized as follows. The next section presents an embedding of global constraints into SAT, called SAT(gc). We describe a prototype system of SAT(gc), named SATCP, in Section 3. In Section 4, we describe the experiments we carry out, along with details on the benchmarks used, their encoding in SAT(gc), the experimental results and analysis. We then discuss some implementation issues of SATCP in Section 5. Section 6 compares SAT(gc) with related frameworks, with Section 7 providing conclusions and pointing to future directions.

2 The SAT(gc) Framework

In this section, we describe the framework for tightly embedding global constraints in SAT solving, which we refer to as SAT(gc).

Here, first we will provide the language and notation specification of the SAT(gc) framework. Then we will describe the algorithm for SAT(gc) solver, which deals with the following two major issues of the integration - how to perform deduction in the presence of global constraints in a SAT(gc) formula, and how to perform conflict directed backtracking and learning in the presence of global constraints.

2.1 Language and Notation

In SAT, a formula is a finite set of clauses in propositional logic, where a clause is a disjunction of literals and a literal is either a proposition or its negation. Propositions are also called *variables*. To distinguish, let us call these variables *normal variables*. In the language of SAT(gc), we have two additional types of variables/literals. The first is called a *global constraint literal*, or just a *gc-literal*, which represents a call to a global constraint. E.g., we can write a clause

$$allDiff(x_0 : \{v_1, v_2\}, x_1 : \{v_2, v_3\}) \vee \neg p$$

where the first disjunct is a call to the global constraint allDifferent in which x_0 and x_1 are CSP variables each of which is followed by its domain. In the sequel, we will use a named variable in the place of a gc-variable, with the correspondence between it and the (call to the) global constraint as part of a SAT(gc) instance.

A gc-literal is true if and only if the corresponding global constraint is solvable. Then it means that there exists one or more solutions for that gc-literal. Such a solution can be represented by a conjunction of propositional variables, each of which is a proposition representing that a given CSP variable takes a particular value from its domain. These new type of variables are called *value variables*. For each CSP variable x and each value a in its domain, we write $x = a$ for the corresponding value variable. Semantically, $x = a$ is true iff x is assigned with value a. Since a value variable is just a proposition, it can appear in clauses of a SAT(gc) instance.

As a CSP variable cannot be assigned to more than one value from its domain, we impose the *exclusive value axioms (EVAs)*: for each CSP variable x and distinct domain values a and b, we have a clause $\neg(x = a) \vee \neg(x = b)$. In the sequel, we assume that EVAs are part of a SAT(gc) instance, so that *unit propagation* enforces these axioms automatically.

With the language of SAT(gc) defined above, given a SAT(gc) instance, a gc-variable in it is semantically equivalent to a disjunction of conjunctions of value variables, augmented by the exclusive value axioms, with each conjunction representing a solution of the corresponding global constraint (if such a disjunction is empty, it represents *false*). That is, a SAT(gc) instance is semantically equivalent to a propositional formula. Given a SAT(gc) instance Π, let us denote by $\sigma(\Pi)$ this propositional formula. We now can state precisely what the satisfiability problem in the current context is: *Given a formula Π in the language of* SAT*(gc), determine whether there exists a variable assignment such that $\sigma(\Pi)$ evaluates to true.*

2.2 Representation in SAT(gc)

Let us consider some examples. In the first, suppose given a 4 by 4 board where each cell contains a number from a given domain D. We can express a disjunctive constraint, "at least one row has the sum of its numbers equal to a given number, say k", as follows

$$sum(x_{11} : D, \ldots, x_{14} : D, =, k) \vee \ldots \vee sum(x_{41} : D \ldots, x_{44} : D, =, k)$$

In SAT(gc) this can be written by a clause of four gc-variables as

$$v_{g_1} \vee v_{g_2} \vee v_{g_3} \vee v_{g_4}$$

with the correspondence between the gc-variables and global constraints recorded as part of input instance. If, in addition, we want to express that there should be exactly one of sum constraints that holds, we can write

$$\neg v_{g_i} \vee \neg v_{g_j} \qquad 1 \leq i, j \leq 4, i \neq j$$

As another example, suppose we want to represent a conditional constraint: given a graph and four colors, $\{r, b, y, p\}$ (for red, blue, yellow, and purple), if a node a is colored with red, denoted by variable a_r, then the nodes with an edge from node a, denoted by $edge_{a,n_i}$ for node n_i, must be colored with distinct colors different from red. This can be modeled by

$$\neg a_r \vee \neg edge_{a,n_1} \vee \neg edge_{a,n_2} \vee ... \vee \neg edge_{a,n_m} \vee v_g$$

where v_g denotes the global constraint, $allDiff(x_{n_1} : \{b, y, p\}, ..., x_{n_m} : \{b, y, p\})$.

Apparently, SAT(gc) allows some complex interactions of constraints, e.g., conditional disjunctive constraints. Note that unlike typical CSP implementations(see, e.g., [2]), SAT(gc) encoding is capable of expressing conditional constraints and disjunctive constraints.

2.3 SAT(gc) Solver

We formulate a SAT(gc) solver in Algorithm 1, which is an extension of the iterative DPLL algorithm given in [23].

Given an instance Π in SAT(gc), the solver first performs preprocessing by calling the function *gc_preprocess*() (Line 1, Algorithm 1). It applies the standard preprocessing operations; however, it will not make any assignments on gc-variables. If *gc_preprocess*() does not solve the problem, then following a predefined decision heuristic the solver proceeds to branch on an unassigned variable (Line 5, Algorithm 1) to satisfy at least one clause in Π. Each decision variable is associated with a *decision level*, which starts from 1 and gets incremented on the subsequent decision level by 1. Then, the procedure *gc_deduce*() is invoked (Line 7, Algorithm 1), and any new assignment generated by the procedure gets the same decision level of the current decision variable.

Procedure *gc_deduce*() In standard Boolean Constraint Propagation (BCP), there is only one inference rule, the *unit clause rule* (UCR). With the possibility of value literals to be assigned, either as part of a solution to a global constraint or as a result of decision or deduction, we need two additional propagation rules.

– **Domain Propagation (DP):** When a CSP variable x is committed to a value a, all the occurrences of x in other global constraints must also commit to the same value. Thus, for any global constraint g and any CSP variable x in it, whenever x is committed to a, $Dom(x)$ is reduced to $\{a\}$. Similarly, when a value variable is assigned to false, the corresponding value is removed from the domain of the CSP variable occurring in any global constraint.

```
1  status = gc_preprocess()
2  if status = KNOWN then
3      return status
4  while true do
5      gc_decide_next_branch()
6      while true do
7          status = gc_deduce()
8          if status == INCONSISTENT then
9              blevel = current_decision_level
10             gc_backtrack(blevel)
11         else if status == CONFLICT then
12             blevel = gc_analyze_conflict()
13             if blevel == 0 then
14                 return UNSATISFIABLE
15             else
16                 gc_backtrack(blevel)
17         else if status == SATISFIABLE then
18             return SATISFIABLE
19         else
20             break
```

Algorithm 1: An Iterative Algorithm for SAT(gc)

– **Global Constraint Rule (GCR):** If the domain of a CSP variable of a global constraint v_g is empty, v_g is not solvable, which is therefore assigned to false. If a global constraint v_g is assigned to true, the constraint solver is called. If a solution is returned, the value variables corresponding to the generated solution are assigned to true; if no solution is returned, v_g is assigned to false.

Now BCP consists of three rules, UCR, DP, and GCR, which are performed repeatedly until no further assignment is possible. Note that, since all of these rules are monotonic, the order of their applications is unimportant.

Since a global constraint v_g in Π is semantically equivalent to the disjunction of its solutions (in the form of value variables), when v_g is assigned to false in the current partial assignment, the negation of the disjunction should be implied. Algorithm 1 does not do this explicitly. Instead, it checks the consistency in order to prevent an incorrect assignment.[1] In case of *gc_deduce*() returning *INCONSISTENT*, the search backtracks to the current decision level (Line 10, Algorithm 1). Otherwise, SAT(gc) checks if a conflict has occurred. If yes, SAT(gc) invokes its conflict analyzer *gc_analyze_conflict*() (Line 12, Algorithm

[1] The general process of this checking can be highly complex, as it may involve the generation of all solutions of a global constraint. However, in many practical situations, e.g., for all the benchmarks we have experienced, this checking is not necessary.

1), which performs conflict analysis, possibly learns a clause, and returns a backtrack level/point.[2]

Conflict Analysis in SAT(gc) Let us first review some terms of DPLL based conflict analysis. The descriptions are based on the procedural process of performing (standard) BCP that implements what is called FirstUIP [22].

– *Antecedent clause (of a literal)*: the antecedent clause of a literal l is the clause which has forced an implication on l.
– *Conflicting clause*: the first failed clause, i.e., the first clause during BCP in which every literal evaluates to false under the current partial assignment.
– *Conflicting variable*: The variable which was assigned last in the conflicting clause.
– *Asserting clause*: the clause that has all of its literals evaluate to false under the current partial assignment and has exactly one literal with the current decision level.
– *Resolution*: The goal is to discover an asserting clause. From the antecedent clause *ante* of the conflicting variable and the conflicting clause *cl*, resolution between the two combines *cl* and *ante* while dropping the resolved literals. This has to be done repeatedly until *cl* becomes an asserting clause.
– *Asserting level*: the second highest decision level in an asserting clause. Note that by definition, an asserting clause has at least two literals.

Below we describe the conflict analyzer $gc\_analyze\_conflict()$ of SAT(gc) . For details on $gc\_analyze\_conflict()$ see [7].

Like the conflict analyzer of [23], $gc\_analyze\_conflict()$ attempts to find the asserting clause by using an iterative resolution process. In each iteration of the resolution process,

- it begins with a conflicting clause *cl* and obtains the last failed literal *lit* in *cl*. *lit* can be either a gc-literal, a value literal or a normal literal. The following cases handles each of the possibilities:
  (a) *lit* is a gc-literal: Last call to the constraint solver for that *lit* failed. There are two subcases.
    (1) *lit* is intrinsically unsolvable: No previous DP operation was performed on the CSP variables in the scope of *lit* and no call to *lit* has succeeded before; only the other literals in *cl* may satisfy the clause. We drop *lit* from *cl*. There are three subcases.
      (i) *cl* becomes empty: The given SAT(gc) instance is not satisfiable.
      (ii) *cl* becomes unit: Then *cl* cannot be an asserting clause (by definition an asserting clause has at least two literals in it). So, we perform chronological backtracking.

[2] A backtrack level leads to backtracking to the decision variable of that level, i.e., undoing all the assignments up to the decision variable of that level, while a backtrack point is a point of an assignment, which may or may not be a point of decision.

(iii) Otherwise: Continue with resolution.

(2) *lit* is not intrinsically unsolvable : We perform chronological backtracking. If *lit* is the *decision variable* of the current decision level, the previous decision level is returned as the backtracking level; Otherwise *lit* is *forced* in the current decision level, in which case the current decision level is returned as the backtracking level.

(b) *lit* is a value literal: Value literal may or may not have an antecedent clause, depending on how its truth value is generated.

(1) *lit* has no antecedent clause: *lit* is a value literal assigned by a DP, SAT(gc) backtracks to the point where the corresponding global constraint (which triggered the DP operation) is invoked for trying to generate an alternative solution for the same global constraint.

(2) *lit* has an antecedent clause : Continue with resolution.

(c) *lit* is a normal literal: Continue with resolution.

In $gc\_analyze\_conflict()$, after the cases (a) and (b), resolution is performed over *cl* and *ante* which results in a new *cl*. Notice that, the resulting clause *cl* also has all of its literals evaluated to false, and is thus a conflicting clause. We then again check the last assigned literal *lit* in *cl*. If *lit* does not have any antecedent clause and *lit* is not a decision variable, then it becomes the case of (b). Otherwise, this resolution process is repeated until *cl* becomes an asserting clause, or either one of the above two cases (a) or (b) occurs. If an asserting clause is found, then the procedure $gc\_analyze\_conflict()$ learns the asserting clause *cl* and returns the asserting level as the backtracking level.

After $gc\_analyze\_conflict()$ returns the backtracking level, if it is 0 then SAT(gc) returns UNSATISFIABLE (Line 14, Algorithm 1). Otherwise, it calls $gc\_backtrack(blevel)$.

Backtracking in SAT(gc) The procedure $gc\_backtrack(blevel)$ distinguishes different types of conflict cases, depending on how the backtracking level is obtained:

(a) *blevel* obtained from an asserting clause: $gc\_backtrack(blevel)$ backtracks to *blevel* and unassigns all the assignments up to the decision variable of $blevel + 1$. After backtracking the learned clause *cl* becomes a unit clause and the execution proceeds from that point in a new search space within the level *blevel*.

(b) Otherwise: we perform chronological backtracking. (b) has three sub cases.

(1) *blevel* is a backtrack point in the current decision level: We backtrack and unassigns assignments up to that backtrack point in the current decision level.

(2) Conflict due to a gc-literal failure: We backtrack to *blevel* and unassign assignments up to the decision variable of *blevel*.

(3) Inconsistency is detected during deduction: We perform backtracking similarly as in (b-(2)).

Example 1. Suppose, as a part of a SAT(gc) instance Π, we have

$(c1)\ \neg r \vee d \quad (c2)\ r \vee v_g \quad (c3)\ t \vee s \vee \neg(x_1 = a) \vee p \quad (c4)\ t \vee s \vee \neg(x_1 = a) \vee \neg p$

where v_g is a gc-variable for the global constraint, $allDiff(x_1 : \{a\}, x_2 : \{a, b\})$.

Let the current decision level be dl, and suppose at a previous decision level dl' $\neg s$ and $\neg t$ were assigned, and at level dl $\neg r$ is decided to be true. Then, v_g is unit propagated from clause $c2$. The call for v_g returns the CSP solution $\{x_1 = a, x_2 = b\}$, hence the value variables $x_1 = a$ and $x_1 = b$ are assigned to true; but then a conflict occurs on $c4$. So, $gc\_analyze\_conflict()$ is called.

In the procedure $gc\_analyze\_conflict()$, $c4$ (conflicting clause) and $c3$ (antecedent clause of the conflicting variable p) are resolved so that cl becomes $t \vee s \vee \neg(x_1 = a)$. Then, it is found that the last assigned literal in cl is $\neg(x_1 = a)$, which is generated by the solution of v_g. So, it returns the assignment point of v_g as the backtrack point. The procedure $gc\_backtrack(blevel)$ unassigns all assignments up to v_g. Then, the constraint solver is called again, but this time v_g generates no alternative solution. So, v_g is assigned to false. As a result, a conflict occurs on clause $c2$. The procedure $gc\_analyze\_conflict()$ is again called.

It is found that the conflicting variable is a forced gc-variable v_g and a solution was previously generated for it. So, $gc\_analyze\_conflict()$ returns the current decision level as the backtracking level, and $gc\_backtrack(blevel)$ backtracks to the assignment $\neg r$, and flips it to r. This flipping immediately satisfies clause $c2$ and the literal d is unit propagated from $c1$. The search continues from there.

3 Implementation

We have implemented a prototype system of SAT(gc), which is called SATCP, where a SAT solver named ZCHAFF [3] is used as the DPLL engine and a constraint solver named GECODE [4] is used as the constraint solving engine.

3.1 Preprocessing, Deduction, Conflict Analysis and Backtracking

To comply with the SAT(gc) framework, in SATCP gc-varialbe are not assigned during the preprocessing step. Existing preprocessing function of ZCHAFF is modified accordingly. Intuitively, a solution of a gc-variable tends to make a good amount of implications by DP operation. This intuition has lead us to implement a variable selection heuristic, which puts higher priority on gc-variables of a SAT(gc) formula Π. The order in which gc-variables are considered for decision is determined by the order of appearance of gc-variables in Π . We have slightly modified the deduce function of ZCHAFF to implement GCR and DP. From the application perspective, a gc-variable occurs only in a unit clause positively. So GCR is implemented only for positive phased gc-literals. Before invoking GECODE for solving a gc-literal, according to the partial assignment, DP

[3] http://www.princeton.edu/ chaff/zchaff.html
[4] http://www.gecode.org/

operations are performed on the domain of its CSP variables. For conflict analysis and backtracking, we relied on the existing conflict analysis and backtracking function of ZCHAFF, with slight modification. In SATCP, when a conflict occurs because of gc-literal assignment, we raise a flag by assigning respective value to a designated flag variable. The conflict analyzer uses this flag to identify types of conflict that have occurred. As gc-literals are only assigned as decision literals, gc-literals can not be forced. So, in case of any gc-literal conflict in SATCP, the conflict analyzer function returns *blevel* and the backtracking function unassigns all the assignments up to the decision variable of *blevel*. [5] For more details, please see [6].

3.2 GC-Variable and Search Engine

CSP problems are modeled in GECODE by creating a subclass of a built-in class named Space and specifying the model inside that subclass.The solution for a CSP model are searched by creating search engines [16].

From the CP perspective, every global constraint in a given SAT(gc) formula is an independent CSP problem. So, before starting executing SATCP, for every gc-variable v_{g_i} in that formula, we create a subclass of the class Space, which models the global constraint v_{g_i} as an independent CSP model. At the very beginning of the execution of SATCP, for each of the CSP models it creates a search engine globally.

SATCP uses the Branch and Bound (BAB) search engine, which is a built-in search engine of GECODE [16]. BAB search engine has a public method, named *next*(), which returns the next alternative solution for the CSP model to which it is attached. When there is no more alternative solution exists for the attached model, it returns *null* and expires. BAB engine also allows the implementation of a virtual function namely - *constraint*(), which allows us to post additional constraints before searching for the next alternative solution. This property of BAB engines is particularly favorable for our implementation. Before calling GECODE to generate an alternative solution for a gc-variable v_g, by using that *constraint*() function, SATCP posts a constraint named *dom*, which sets the domain of the CSP variables of v_g according to the current partial assignment. For more details on BAB search engine please see [16].

4 Experiments

In this section we present experiments performed with SATCP [6] using three benchmark problems. We present these benchmarks, their SAT(gc) encodings, experimental results.

[5] For gc-failure-conflict, *blevel* is the previous decision level of current decision level and for value-variable-conflict, *blevel* is the current decision level.

[6] Source code of SATCP and benchmark instance generators can be found at https://github.com/solimul/SATCP

4.1 The Latin Square Problem

The Latin Square (LS) can be described as follows: *A Latin square of order n is an n by n array of n numbers (symbols) in which every row and columns must contain distinct numbers (symbols).*

We encode the constraint "no two numbers are assigned to the same cell" by negative binary clauses and constraint "every number must appear exactly once in a row and in a column" is encoded by using *n allDiff* global constraints.

We have run the LS problem on ZCHAFF (with SAT encoding) and on our prototype implementation system, SATCP (with SAT(gc) encoding) for instances of different sizes up to 10 by 10 LS. For the 10 by 10 instance of LS, ZCHAFF does not return any solution in 15 minutes, but SATCP returns it within 169.71 seconds[7]. For details, see [7].

To encode LS problem in SAT, total number of clauses required are $O(n^4)$ [11]. The SAT(gc) instance has $O(n^3)$ clauses. Thus, by the use of global constraints, one can write more compact representations in SAT(gc) than in SAT. This plus the efficient propagators for $allDiff$ global constraint as implemented in GECODE seems to be the main reasons for the better performance of our implementation of SATCP.

4.2 The Normal Magic Square Problem

A Normal Magic Square (NMS) can be defined as follows: *A magic square of order n is an arrangement of n^2 numbers, usually distinct integers, in a square, such that the sum of n numbers in rows, columns, and in both diagonals are equal to a constant number.*

The constant sum over rows, columns and diagonals is called the *magic sum*, which is known to be $n(n^2+1)/2$.

We present two different encoding for the NMS problem. We refer the first one as *monolithic* encoding, where the whole NMS problem is encoded as a single gc-variable. The monolithic gc-variable is composed of a $allDiff$ global constraint over n^2 cells to model the distinct constraint of the square and $2n+2$ *sum* global constraints, each over n different cells of the square. We refer the second one as *decomposed* encoding, where the distinct constraint is encoded by negative binary clauses and the sum constraints are encoded by $2n+2$ *sum* global constraints. In this encoding, each of the *sum* global constraints is encoded as single gc-variable. For both of the cases, the *sum* constraints encodes the sum constraints across the rows, columns and diagonals of the square.

Notice that in monolithic encoding all the variables and constraints are put inside the same constraint store, while in decomposed encoding the *sum* constraints are put into separate constraint store.

[7] Size of 10 by 10 SAT(gc) instance : 1000 variables, 1210 clauses (1200 CNF clauses and 10 clauses containing 10 $allDiff$ gc-variables)

We have solved NMS problem with monolithic constraint up to the size 7 by 7 by SATCP in under 4 seconds[8]. For the decomposed encoding, SATCP solved of order 3 [9] in 3.17 sec. But for the instances of higher order, SATCP failed to generate a solution within 15 minutes. For details, see [7]. In section 5.2. we present an analysis of the result with decomposed encoding of NMS.

The experimental results with the monolithic encoding of NMS running on SATCP confirms the results of [21], as with this encoding SATCP runs much faster than CLASP, which uses aggregates to encode numerical constraints. For example, CLASP solves the NMS problem of order 7 in 450.58 seconds. It also demonstrates that, the propagators of global constraints from CSP are more efficient than the aggregates from the logic programming framework.

4.3 The Planning Problem of Block Stacking with Numerical Constraints

The planning problem of block stacking with numerical constraints can be described as follows:

> In a table, there are n ($n > 1$) stacks of blocks, each having m_i number of blocks, where $1 \leq i \leq n$. Let $block_{ij}$ be the j^{th} ($1 \leq j \leq m_i$) block of i^{th} ($1 \leq i \leq n$) stack. In the initial configuration in every stack i the first block $block_{i1}$ is placed on the table. If $m_i > 1$, then $block_{ij}$ is placed on $block_{i(j-1)}$. Every block $block_{ij}$ has a weight w_{ij}. We have to generate a plan of actions for building a new stack of blocks by taking exactly one block from each of the initial n stacks in such a way that
> - The total weight of the selected blocks should be equal to a certain total weight W_{max}. That is, if block $j^1, j^2 \dots j^n$ are selected respectively from stacks $1, 2 \dots n$, then $w_{1j^1} + w_{2j^2} + \dots + w_{ij^i} + \dots + w_{nj^n} = W_{max}$ (Constraint1).
> - Block selected from the i^{th} stack must be placed over block selected from the $(i-1)^{th}$ stack (Constraint2).

Constraint1 is encoded by using a *sum* global constraint as follows:

$$sum(stack_1, stack_2, \dots, stack_n, W_{max})$$

Where $Dom(stack_i) = \{w_{i1}, w_{i2}, \dots w_{ij}, \dots w_{im_i}\}$. The assignment $stack_i = w_{ij}$ asserts that the j^{th} block form the i^{th} stack is selected for building the goal stack.

From a STRIPS specification, we generate a planning instance that models constraint2 [10]. We also introduce $n * m$ (where $m = \bigwedge_{i=1}^{n} m_i$) propositional

[8] Size of 7 by 7 monolithic SAT(gc) instance of NMS instance: 2401 variables, 50 clauses (49 CNF clauses and a clause for the monolithic gc-variable)

[9] Size of 3 by 3 decomposed SAT(gc) instance of NMS: 81 variables, 664 clauses (657 CNF clauses and 8 clauses for 8 *linear* gc-variables.

[10] For our experiment, we have used action based CNF encoding, generated by an automated SAT based planner named, SATPLAN.

variables, which are the value variables corresponding to the domain values of $stack_i$. Then for every pair of blocks, $block_{ij}$ and $block_{i'j'}$ (where $i' = i + 1$), we add their corresponding value literals to their stacking action (goal action) clauses (at goal layer).

We have solved five instances with different number of blocks and stacks for the planning problem of block stacking with numerical constraint by SATCP. We solve those problems under 131 seconds. For details, see [7].

5 Implementation Issues with SATCP

Here we briefly describe the implementation issues of SATCP.

5.1 Alternative Solution Generation for once failed gc-litarl

Whenever a gc-variable v_{g_i} is assigned, SATCP executes the *next*() method of the search engine attached to the model of v_{g_i} to get the next alternative solution for that v_{g_i}. When the attached search engine does not find any alternative solution for v_{g_i}, it returns *null* and expires. If that v_{g_i} is assigned again, SATCP creates a new search engine at the local scope which searches for alternative solutions for v_{g_i}. Here, one point is worth mentioning. Every time an alternative solution needs to be generated for such a v_{g_i} (once failed), a local search engine needs to be created. So, for getting the j^{th} alternative solution for such v_{g_i}, $j-1$ solutions need to be generated. This is one reason, for which SATCP provides no solution within a reasonable amount of time for the decomposed encoding of NMS for the orders greater than 3.

This undesirable effect is caused by the inability of GECODE to reinstantiate the globally created search engine associated with a gc-literal (once failed), in case that gc-literal gets re-assigned by the SAT component. Apparently, to tackle this problem, GECODE needs to be modified to make the initially created global search engine reusable.

5.2 Inhibition of Constraint Propagation

In the decomposed encoding of NMS, we have used separate *sum* global constraints for modeling the sum constraints across each of the rows, columns and diagonals. Thus, at the GECODE end, we model each of these global constraints as separate CSPs. When SATCP executes on a decomposed magic square instance, these global constraints are put into separate constraint stores and are treated as independent and unrelated constraints. But, notice that these *sum* global constraints are related, as they are sharing CSP variables with each other. Assigning a CSP variable x to a value by any of the global constraints effects all CSP variables of the global constraints those have x on their scope. Though we are performing DP operation, after an assignment on a CSP variable x is made by a global constraint, the propagator for other related global constraint (those have x on their scope) are not getting notified of that assignment, as the *sum* global constraints are put on different constraint store. Therefore, the result of DP operation cannot be propagated to other related CSP variables. This is another reason for which SATCP fails to provide any solution for decomposed encoding of NMS for the orders greater than 3.

Once a CSP variable is assigned to a value, SATCP is inherently incapable of propagating constraint on related global constraints, as each global constraints are put into separate constraint stores. To prevent this inhibition of constraint propagation, we need to put all the related global constraints in the same constraint store, with a call to a gc-literal returning solution only for itself. Modifying GECODE seems to be a way to tackle this problem.

6 Related Work

To compare SAT(gc) with SMT, both adopt a DPLL based SAT solver as the overall solver. The SMT solver uses a theory solver to determine the satisfiability of a portion of a T-formula. On the other hand, the SAT(gc) solver uses a constraint solver to compute a solution of a global constraint for which the constraint solver is invoked. In SMT, whenever an inconsistent assignment is found by the T-solver, it informs the DPLL solver about the inconsistency and the T-solver sends information back to the DPLL solver as a theory lemma, so that the DPLL solver can learn a clause and backtrack to a previous point. On the other hand, in SAT(gc) no such conflicting information is sent back from the constraint solver. The DPLL component of SAT(gc) identifies the conflicts/inconsistencies related to the global constraint at hand and does the necessary domain setup for the respective CSP variables, clause learning and backtracking. The constraint solver is used as a black box, to solve the global constraints for which it is called.

In [9], following the lazy SMT approach, a framework called CDNL-ASPMCSP has been developed for integrating CSP style constraint solving in Answer Set Programming (ASP). The ASP solver passes the portion of its partial assignment associated with constraints to a CP solver,which checks the satisfiability of these constraint atoms. In case of conflicts, conflict analysis is performed as follows: it constructs a non-trivial reason from the structural properties of the underlying CSP problem in the ASP program at hand and use this reason to find a learned clause and backtracking level.

SAT(gc) is monotonic, in contrast to CDNL-ASPMCSP, where the semantic issue of allowing global constraints in non-monotonic rules is nontrivial. SAT(gc) is more eager than CDNL-ASPMCSP, in that whenever a Boolean literal associated with a constraint is assigned, it calls the constraint solver immediately. If any solution is returned, the current partial assignment is also eagerly extended by adding relevant value literals. In contrast to CDNL-ASPMCSP, whenever a conflict involves a gc-literal or value literal, SAT(gc) performs chronological backtracking. The more eager approach of SAT(gc) identifies immediate occurrence of conflict due to a gc-literal or a value literal (if any exists) and enables SAT(gc) to perform chronological backtracking, as the backtracking points are obvious.

In [14] a tight integration of SAT and CP is presented. In their work, upon invocation, instead of returning reduced domains, the attached CSP solver returns some propagation rules, which emulate the functionality of those propagators. These rules are then converted into SAT clauses. The converted clauses are added one by one as learned clauses into the SAT clause database. This approach requires heavy modification of the attached solvers. In our approach we can incorporate an off-the-shelf CSP solver more easily; when SAT is absent SAT(gc) behaves like CSP. But for the solver described in [14], there is no guarantee.

An encoding scheme is presented in [17] to encode a finite linear CSP into SAT. The idea is to first convert a linear comparison into a primitive linear comparison by using the bounds of the comparison and then encode the primitive linear comparison into a SAT formula. In contrast to SAT(gc), which integrates SAT and CSP in a tight fashion, [17] presents a very specialized encoding scheme to solve finite linear CSPs by using SAT solvers.

In [10], the authors present a translation scheme from FlatZinc [12] CSP model into CNF, which is then solved by a SAT solver. [10] bundles MiniZinc and a SAT solver into one tool, named FznTini. Unlike SAT(gc), FznTini is not an extension of SAT, but it is a very specialized tool for solving FlatZinc CSP models by SAT solvers. Moreover, the translation scheme of FznTini does not directly support any global constraints.

In [18] the authors present a compact order encoding of CSP instances to SAT, which uses numeric systems of base ≥ 2 to represent an integer number, and the compact sized encoding results in fewer number of propagations than the previously proposed encoding in [17]. However, [18] does not address the encoding of other types of constraints, including global constraints. Also, unlike SAT(gc), it is not a an extension of SAT.

7 Conclusion and Future Work

The current implementation of SATCP has two major weaknesses. Firstly, SATCP needs to perform repeated invocations to the constraint solver to generate alternative solutions for once failed gc-literal. Secondly, the assignments implied by the SAT solvers are not propagated into related global constraints as separate but related global constraints are needed to be kept into separate constraint stores. For both of these cases, apparently, the constraint solver needs to be modified. Note that these weaknesses, though present in the current version of SATCP, are not intrinsic problems of SAT(gc). We fully expect to address these problems in future versions of SATCP. Another interesting question is potential applications. A possible direction is the problem domain where planning and scheduling are tightly dependent on each other. Our benchmark from the planning domain shows a first step towards this direction.

References

1. Marcello Balduccini. Industrial-size scheduling with ASP+CP. In *Proc. LPNMR'11*, pages 284–296, 2011.

2. P. Baptiste and C.L. Pape. Disjunctive constraints for manufacturing scheduling: principles and extensions. *Interntional Journal of Computer Integrated Manufacturing*, 9(4):306–310, 1996.
3. Christian Bessiere, George Katsirelos, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. Propagating conjunctions of alldifferent constraints. In *Proc. AAAI'10*, 2010.
4. L. Bordeaux, Y. Hamadi, and L. Zhang. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys*, 38(4), 2006.
5. D. Chai and A. Kuehlmann. A fast pseudo-boolean constraint solver. *IEEE Transactions on Computer-Aided Design of Intergrated Circuits and Systems*, 24(3):305–317, 2005.
6. Md Solimul Chowdhury. SAT with Global Constriants. Master's thesis, Universtiy of Alberta, 2011.
7. Md Solimul Chowdhury and Jia-Huai You. SAT with Global Constraints. In *ICTAI*, pages 73–80, 2012.
8. Christian Drescher and Toby Walsh. A translational approach to constraint answer set solving. *TPLP*, 10(4-6):465–480, 2010.
9. M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In *Proc. of the 25th International Conference on Logic Programming*, pages 235–249, 2009.
10. Jinbo Huang. Universal booleanization of constraint models. In *CP*, pages 144–158, 2008.
11. I. Lynce. *Propositional Satisfiability: Techniques, Algorithms and Applications.* PhD thesis, Instituto Superior Tcnico, Universidade Tcnica de Lisboa, 2005.
12. Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard cp modelling language. In *CP*, pages 529–543, 2007.
13. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.
14. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
15. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc., 2006.
16. C. Schulte, G. Tack, and M. K. Lagerkvist. *Modeling and Programming with Gecode.* 2008.
17. Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.
18. Tomoya Tanjo, Naoyuki Tamura, and Mutsunori Banbara. A compact and efficient SAT-encoding of finite domain CSP. In *SAT*, pages 375–376, 2011.
19. W.-J. van Hoeve and I. Katriel. Global constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 7. Elsevier, 2006.
20. Mark Wallace. Practical applications of constraint programming. *Constraints*, 1(1/2):139–168, 1996.
21. Yisong Wang, J. You, F. Lin, L. Yuan, and M. Zhang. Weight constraint programs with evaluable functions. *Annals of Mathematics and Artificial Intelligence*, 60(3-4):341–380, 2010.
22. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. The 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285, 2001.
23. L. Zhang and S. Malik. The quest for efficient Boolean Satisfiability solvers. In *Proc. CAV'02*, LNCS 2404, pages 17–36. Springer, 2002.

SWI-Prolog version 7 extensions

Jan Wielemaker

Web and Media group, VU University Amsterdam,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands,
`J.Wielemaker@vu.nl`

Abstract. SWI-Prolog version 7 extends the Prolog language as a general purpose programming language that can be used as 'glue' between components written in different languages. Taking this role rather than that of a domain specific language (DSL) *inside* other IT components has always been the design objective of SWI-Prolog as illustrated by XPCE (its object oriented communication to the OS and graphics), the HTTP server library and the many interfaces to external systems and file formats. In recent years, we started extending the language itself, notably to accommodate expressing syntactic constructs of other languages such a HTML and JavaScript. This resulted in an extended notion of operators and quasi quotations. SWI-Prolog version 7 takes this one step further by extending the primitive data types of Prolog. This article describes and motivates these extensions.

1 Introduction

Prolog is often considered a *DSL*, a *Domain Specific Language*. This puts the language in a position similar to e.g., SQL, acting as a component in a larger application which takes care of most of the application and notably the interfaces to the outside world (be it a graphical interface targeting at humans or a machine-to-machine interface). This point of view is illustrated by vendors selling their system as e.g., *Prolog + Logic Server* (Amzi!), the interest in Prolog-in-some-language implementations as well as by the many questions about embedding interfaces that appear on mailing lists and forums such as stackoverflow.[1]

SWI-Prolog always had the opposite viewpoint, proposing Prolog as a 'glue' (or scripting language) suitable for the overall implementation of applications. As a consequence, SWI-Prolog has always provided extensive libraries to communicate to other IT infrastructure, such as graphics (XPCE), databases, networking, programming languages and document formats. We believe this is a productive approach for the following reasons:

- Many external entities can easily be wrapped in a Prolog API that provides a neat relational query interface.
- Given a uniform relational model to access the world makes reasoning about this world simple. Unlike classical relational query languages such as SQL though, Prolog rules can be *composed* from more elementary rules.

[1] `http://stackoverflow.com/`

– Prolog is one of the few languages that can integrate application logic without suffering from the *Object-Relational impedance mismatch*.[2] [5]
– Prolog naturally blends with constraint systems, either written in Prolog or external ones.
– Many applications have lots of little bits of logic that is way more concisely and readably expressed in terms of Prolog rules than in imperative if-then-else rules.
– Prolog's ability to write application-specific DSLs is highly valuable for developing larger applications.
– Prolog's reflective capabilities simplify many application specific rewriting, validation and optimisation requirements.
– Prolog's dynamic compilation provides a productive development environment.
– Prolog's simple execution order allows for writing simple sequences of actions.

For a long time, we have facilitated this architecture within the limitations of classical Prolog, although we lifted several limits that are common to Prolog implementations. For example, SWI-Prolog offers atoms that can hold arbitrary long Unicode strings, including the code point '0', which allows applications to represent text as well as 'binary blobs' as atoms. Atom garbage collections ensures that such applications can process unbounded amounts of data without running out of memory. It offers unbounded arity of compound terms to accommodate arrays and it offers multi-threading to allow for operation in threaded server environments. SWI-Prolog's support of data types and syntax was considered satisfactory for application development. Over the past (say) five years, our opinion started to shift for the following reasons:

– With the uptake of SWI-Prolog as a web-server platform, more languages came into the picture, notably HTML, JavaScript and JSON. While HTML can be represented relatively straightforward by Prolog terms, this is not feasible for JavaScript. Representing JSON requires wrapping terms in compounds to achieve an unambiguous representation. For example, JavaScript `null` is represented as @(null) to avoid confusing it with the string `"null"`. SWI-Prolog version 7 allows for an alternative JSON representation where Prolog strings are mapped to JSON strings and atoms are used for the JSON constants `null`, `true` and `false`.
– The primary application domain at SWI-Prolog's home base, the computer science institute at the VU University Amsterdam, in particular the 'web and media' and 'knowledge representation and reasoning' groups, is RDF and web applications. This domain fits naturally with Prolog and especially with SWI-Prolog. Nevertheless, we experienced great difficulty motivating our PhD students to try this platform, often because it looked too 'alien' to them.

In [7] we proposed extensions to the Prolog syntax to accommodate languages such as JavaScript and R using 'extended operators', which allows using $\{\ldots\}$ and $[\ldots]$ as operators. In [8] we brought the notion of quasi quotations to Prolog, providing an elegant way for dealing with external languages such as HTML, JavaScript, SQL and SPARQL as well as long strings. In this article we concentrate on extending Prolog's data types with two goals in mind:

[2] `http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch`

– Facilitate the interaction with other IT systems by incorporating their data types. In particular, we wish to represent data from today's dynamic data exchange languages such as JSON naturally and unambiguously.
– Provide access to structured data elements using widely accepted (functional) syntax.

This paper is organised as follows. In section 2 we identify the missing pieces. In section 3 we describe how these are realised in SWI-Prolog version 7, followed by an evaluation of the impact on compatibility, a preliminary evaluation of the new features, and our conclusions.

2 The missing pieces of the puzzle

2.1 Representing text

ISO Prolog defines two solutions for representing text: atoms (e.g., 'ab') and lists of characters, where the characters are either represented as code points, i.e., integers, such as [97,98] or atoms of one character ([a,b]). Representing text using atoms is often considered inadequate for several reasons:

– It hides the conceptual difference between text and program symbols. Where content of text often matters because it is used in I/O, program symbols are merely identifiers that match with the same symbol elsewhere in the program. Program symbols can often be consistently replaced, for example to obfuscate or compact a program.
– Atoms are globally unique identifiers. They are stored in a shared table. Volatile strings represented as atoms come at a significant price due to the required cooperation between threads for creating atoms. Reclaiming temporary atoms using *Atom garbage collection* is a costly process that requires significant synchronisation.
– Many Prolog systems (not SWI-Prolog) put severe restrictions on the length of atoms, the characters that can be used in atoms or the maximum number of atoms.

Representing text as a list of character codes or 1-character atoms also comes at a price:

– It is not possible to distinguish (at run time) a list of integers or atoms from a string. Sometimes this information can be derived from (implicit) typing. In other cases the list must be embedded in a compound term to distinguish the two types. For example, `s("hello world")` could be used to indicate that we are dealing with a string.
  Lacking run time information, debuggers and the top level can only use heuristics to decide whether to print a list of integers as such or as a string (see **portray_text/1**). While experienced Prolog programmers have learned to cope with this, we still consider this an unfortunate situation.
– Lists are expensive structures, taking 2 cells per character (3 for SWI-Prolog, which threads lists as arbitrary compound terms, represented as the 'functor' (**./2**) and an array of arguments). This stresses memory consumption on the stacks while pushing them on the stack and dealing with them during garbage collection is unnecessarily expensive.

2.2 Representing structured data

Structured data is represented in Prolog using compound terms, which identify the arguments by position. While that is perfectly natural for e.g., `point`(*X,Y*), it becomes cumbersome if there is no (low) natural number of arguments or if there is no commonly accepted order of the arguments. The Prolog community has invented many workarounds for this problem:

– Use lists of *Name*=*Value* or `Name`(*Value*) terms. While readable, this representation wastes space while accessing elements is inefficient.
– Use compound terms and some form of symbolic access. Alternatives seen here are SWI-Prolog's library(record), which generates access predicates from a declaration, the Ciao solution [4], which provides access using functional notation using *Term*$*field*, the ECLiPSe solution mapping terms `name{key1:value1,key2:value2,...}` to a term `name(value2,_,value1,_,...)` using expansion called by **read_term/3** based on a 'struct' declaration.[3]
– Using binary trees (e.g., the classical DEC10 library(assoc)). This provides fast access, but uses a lot of space while the structures are hard to write and read.

2.3 Ambiguous data

We have already seen one example of ambiguous data: the list [97,98] can be the string `"ab"` or a list with two integers. Using characters does not solve this. Defining a string as a list of elements of a new type 'character' still does not help as it fails to distinguish the empty list (`[]`) from the empty string (`""`). Normally, ambiguity is resolved in one of two ways: the data is passed between predicates that interpret the ambiguous terms in a predefined way (e.g., `atom_codes(A, [])` interprets the `[]` as an empty string) or the data is wrapped in a compound, e.g., `s([97,98])`. The first requires an interpretation context, which may not be present. The latter (known as *non-defaulty* representation) is well suited for internal processing, but hard to read and write and requires removing and wrapping the data frequently.

3 SWI-Prolog 7

With SWI-Prolog version 7, we decided to solve the above problems, accepting that version 7 would not be completely backward compatible with version 6 and the ISO standard. As we will see in section 4 though, the compatibility of SWI-Prolog version 7 to its predecessors can be considered fair. Most of the changes are also available in some other Prolog.

[3] `http://www.eclipseclp.org/doc/userman/umsroot022.html`

3.1 Double quoted strings

Strings, and their syntax have been under heavy debate in the Prolog community, but did not make it into the ISO standard. It is out of the scope of this paper to provide a historical overview of this debate. Richard O'Keefe changed his opinion on strings during the debate on the SWI-Prolog mailinglist. His current opinion can be found in "An Elementary Prolog Library", section 10[4]

SWI-Prolog 7 reads `"..."` as an object of type *string*. Strings are atomic objects that are distinct from atoms. They can represent arbitrary long sequences of Unicode text. Unlike atoms, they are not combined in a central table, live on the term-stack (global stack or heap) and their life time is the same as for compound terms (i.e., they are discarded on backtracking or garbage collection).

Strings as a distinct data type are present in various Prolog systems, e.g., SWI-Prolog, Amzi! and YAP. ECLiPSe [6] and hProlog[5] are the only system we are aware of that uses the common double-quoted syntax for strings. The set of predicates operating on strings has been synchronised with ECLiPSe.

3.2 Modified representation of lists

Representing lists the conventional way using **./2** as cons-cell and the atom '[]' as list terminator both (independently) poses difficulties, while these difficulties can be avoided easily. These difficulties are:

- Using **./2** prevents using this commonly used symbol as an operator because `a.B` cannot be distinguished from `[a|B]`. Changing the functor used for lists has little impact on compatibility because it is (almost) always written as [...]. It does imply that we can use this symbol to introduce widely used syntax to Prolog, as described in section 3.3.
- Using `'[]'` as list terminator prevents dynamic distinction between atoms and lists. As a result, we cannot use polymorphism that involve both atoms and lists. For example, we cannot use *multi lists* (arbitrary deeply nested lists) of atoms. Multi lists of atoms are in some situations a good representation of a flat list that is assembled from sub sequences. The alternative, using difference lists or Definite Clause Grammars (DCGs) is often less natural and sometimes demands for 'opening' proper lists (i.e., copying the list while replacing the terminating empty list with a variable) that have to be added to the sequence. The ambiguity of atom and list is particularly painful when mapping external data representations that do not suffer from this ambiguity.
  At the same time, avoiding `'[]'` as a list terminator avoids the ambiguity of text representations described in section 2.3, i.e., `'[]'` unambiguously represents two characters and [] unambiguously represents the empty string.

[4] `http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm#strs`

[5] `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW366.abs.html`

We changed the cons-cell functor name from '.' to `'[|]'`, inspired by Mercury.[6] We turned the empty list ([]) into a new data type, i.e., [] has the properties demonstrated by the queries below. This extension is also part of CxProlog using `--flags nil_is_special=true`.[7]

```
?- atom([]).                  % [] is not an atom
false.
?- atomic([]).                % [] is atomic
true.
?- is_list('[]').             % '[]' is not a list
false.
?- [] == '[]'.                % our goal
false.
?- [] == [].                  % obviously
true.
?- [] == [/*empty list*/].    % also
true.
?- '[]' == '[   ]'.           % two different atoms
false.
```

3.3 Introducing dicts: named key-value associations

Dicts are a new data type in SWI-Prolog version 7, which represents a key-value association. The keys in a dict are unique and are either atoms or integers. Dictionaries are represented by a canonical term, which implies that two dicts that represent the same set of key-value associations compare equal using **==/2**. Dicts are natively supported by **read/1** and **write/1**. The basic syntax of a dict is described below. Similar to compound terms, there cannot be a space between the *Tag* and the {...} term. The *Tag* is either an atom or a variable, notably _{...} is used as 'anonymous' dict.

Tag{Key1:Value1, Key2:Value2, ...}

Below are some examples, where the second example illustrates that the order is not maintained and the third illustrates an anonymous dict.

```
?- A = point{x:1, y:2}.
A = point{x:1, y:2}.

?- A = point{y:2, x:1}.
A = point{x:1, y:2}.

?- A = _{first_name:"Mel", last_name:"Smith"}.
A = _G1476{first_name:"Mel", last_name:"Smith"}.
```

[6] `http://www.mercurylang.org/information/doc-latest/transition_guide.pdf`
[7] `http://ctp.di.fct.unl.pt/~amd/cxprolog/MANUAL.txt`

Note that our dict notation looks similar, but is fundamentally different from ECLiPSe structs. The ECLiPSe struct notation {...} is a sparse notation for a declared normal compound term. The ECLiPSe notation can only be used for *input*, it is not possible to dynamically add new keys and the resulting term can be used anywhere where a compound term is allowed. For example, ECLiPSe allows us to write predicates with named arguments like this:

```
person{last_name:"Smith"}.
```

In contrast, our dicts are dynamic, but can only appear as a data term, i.e., not as the head of a predicate. This allows for using them to represent dynamic data from e.g., JSON objects or a set of XML attributes.

Dicts also differ from LIFE PSI-Terms [1], which are basically feature vectors that can unify as long as there are no conflicting key-value pairs in both PSI terms. Dicts are ground if all values are ground and it is thus impossible to add keys using unification. PSI terms can be simulated by associating a dict to an attributed variable. The code to unify two dicts with non-conflicting key-value pairs is given below, where **>:</2** succeeds after all values associated with common keys are unified. For example, `T{a:1,b:B} >:< d{a:A,b:2,c:C}` succeeds with *T*=d, *B*=2, leaving *C* unbound.

```
psi_unify(Dict1, Dict2, Dict) :-
        Dict1 >:< Dict2,
        put_dict(Dict1, Dict2, Dict).
```

Functional notation on dicts Dicts aim first of all at the representation of dynamic structured data. As they are similar to the traditional library(assoc), a predicate `get_dict`(*+Key,+Dict,-Value*) is obvious. However, code processing dicts will become long sequences of **get_dict/3** and **put_dict/4** calls in which the actual logic is buried. This at least is the response we get from users using library(record) and library(assoc) and is also illustrated by the aforementioned solutions in Ciao and ECLiPSe. We believe that a functional notation is the most natural way out of this.

The previously described replacement of '.'[8] with '`[|]`' as list cons-term provides the ideal way out of this because (1), the a.b notation is widely used for this purpose and (2) **./2** terms are extremely rare in Prolog. Therefore, SWI-Prolog transforms **./2** terms appearing in goals into a sequence of calls to **./3**,[9] followed by the original goal. Below are two examples, where the left code is the source and the right is the transformed version.

[8] Note that '.' and a plain . are the same, also in SWI-Prolog version 7. We use '.' in running text to avoid confusion with the end of a sentence.

[9] We called the helper predicate to evaluate **./2** terms **./3** to make the link immediate. It could also have been named e.g., **evaluate/3**.

```
                              .(D, last_name, LN),
   writeln(D.last_name)       writeln(LN)
```

```
last_name(D, D.last_name).   last_name(D, LN) :-
                                 .(D, last_name, LN).
```

Functions on dicts In the previous section we described the functional notation used to access keys on dicts. In addition to that, we allow for user defined functions on dicts. Such functions are invoked using the notation *Dict.Compound*, e.g., `Point.offset(X,Y)` may evaluate to a new *Point* dict at offset (*X*,*Y*) from the original. User defined functions are realised by means of the dict *tag*, which associates the dict with a Prolog module (inspired by attribute names of attributed variables). The offset function is defined as:

```
:- module(point, []).

Pt.offset(OX,OY) := point{x:X,y:Y} :-
    X is Pt.x + OX,
    Y is Pt.y + OY.
```

The above function definition is rewritten using term_expansion rules into the code below. The predicate **./3**, handling functional notation based on **./2**, translates *Dict.Compound* into `call`(*Compound, Dict, Result*). For example, the expression `point{x:1,y:2}.offset(2,4)` is translated into `call(offset(2,4),point{x:1,y:2},Result)`, which in term calls the predicate below.

```
offset(OX, OY, Pt, point{x:X, y:Y}) :-
        '.'(Pt, x, X0),
        X is X0+OX,
        '.'(Pt, y, Y0),
        Y is Y0+OY.
```

As *Dict.Atom* accesses a key and *Dict.Compound* calls a user-defined function, we have no way to express functions without arguments. In [7] we already concluded that *name()* is a valuable syntactic building block for DSL construction and therefore we decided to add support for zero-argument compound terms, such as *name()*. Zero-argument compounds are supported by the following predicates:

compound_name_arity(*Compound, Name, Arity***)**
compound_name_arguments(*Compound, Name, Arguments***)**
These predicates operate on compounds with any number of arguments, including zero.

functor(*Callable, Name, Arity*)
Callable **=..** *List*

These predicates operate on atoms or compounds. They raise an error if the first argument is a zero-arity compound.

4 Compatibility

SWI-Prolog version 7 is not fully backward compatible with older versions and drifts further away from the ISO standard. We believe that a programming language must evolve over time to match changing demands and expectations. In other words, there should be a balance between compatibility with older versions of the language and fulfilling evolving demands and expectations. As far as we understand, the ISO standardisation process only allows for strict *extensions* of a language, guaranteeing full backward compatibility with ISO standard. The ISO model allows for none of the described changes because all of them have at least corner cases where they break compatibility. Such a restrictive view does not allow for gradual language *evolution* and forces a *revolution*, replacing the language with a new one. We believe that the evolutionary route is more promising.

Nevertheless, SWI-Prolog version 7 introduces significant changes. We have evaluated the practical consequences of these changes as soon as we had a prototype implementation by porting our locally developed software as well as large systems for which we have the source code. A particularly interesting code base is Alpino [3], a parser for the Dutch language. Alpino has been developed for over 15 years, counts approximately 500K lines of Prolog code and contains many double quoted strings. Porting took two days, including implementing **list_strings/0** described below.

We received several evaluations from users about porting their program from version 6 to version 7, two of which concern code basis between 500K and 1M lines. This section summarise our key findings.

Double quoted strings This is the main source of incompatibilities. However, migrating programs proves to be fairly easy. First, note that string literals in DCGs can be mapped to lists of character codes by the DCG compiler, which implies that code such as `det --> "the"` remains valid. The double-quoted notation is commonly used to represent e.g., the format argument for **format/3**. This is, also conceptually, correct usage of strings and does not require any modification. We developed a program analyzer (**list_strings/0**) which examines the compiled program for instances of the new string objects. The analyzer has a user extensible list of predicates that accept string arguments, which causes a goal `format("Hello World~n")` to be considered safe. In practise, this reveals compatibility issues accurately. There are three syntactic measures to adapt your code:

– Rewrite the code. For example, change `[X] = "a"` into `X = 0'a`.
– If a particular module relies heavily on representing strings as lists of character code, consider adding the following directive to the module. Note that this flag only applies to the module in which it appears.

```
:- set_prolog_flag(double_quotes, codes).
```

– Use a back quoted string (e.g., `` `text` ``). Note that using `` `text` `` ties the code to SWI-Prolog version 7. There is no portable syntax that produces a list of characters. Such a list can be obtained using portable code using one of the constructs below.
  - `phrase("text", List)`
  - `atom_codes("text", List)`

Using the new [] as list terminator This change has very few consequences to Prolog programs. We encountered four issues, caused by calls such as `atom_concat`(*[], . . .*). Typically, these result from using [] as 'nil' or 'null', i.e., *no value*, but using them as a real value.

Using `[|]` *as cons-cell* We encountered a few cases where **./2** terms were handled explicitly or where the functor-name of a term was requested and **.** was expected. We also found lists written as e1.e2.e3.[] and [H|T] written as H.T. Such problems can typically be found by examining the compiled code for **./2** terms.

Together with [], the only major problem encountered is JPL, the SWI-Prolog Java interface. This interface represents Prolog terms as Java objects, assuming there are variables, atoms, numbers and compound terms. I.e., lists are treated as **./2** terms ending in the atom '[]'. We think JPL should be extended with a list representation. This will also cure the frequent stack overflows caused by long lists represented as deeply nested **./2** terms that are traversed by a recursive Java method.

Dicts, functions and zero-arity compounds The dict syntax infrequently clashes with a prefix operator followed by {. . . }. Cases we found include the use of `@{null}` to represent 'null' values, `::{...}` used (rarely) in Logtalk and `\+{...}` which can appear in DCGs to express negation of a native Prolog goal. The **./2** functional notation causes no issues after **./2**-terms that should have used list notation were fixed. The introduction of zero-arity compounds has no consequences for applications that do not use these terms. The fact that such terms can exist exposed many issues in the development tools.

5 Evaluation

The discussions on SWI-Prolog version 7 took place mostly in October to December 2013. The implementation of the above is now considered almost stable. Migrating towards effective and consistent use of these new features is not easy because most existing code and libraries use atoms to represent text and one of the described alternatives to represent key-value sets.

Our evaluation consists of two parts. First, we investigate uptake inside SWI-Prolog's libraries and by users. The second part is a performance analysis.

Dicts Currently, dicts can be used in the following areas:

– As an alternative option representation. All built-in predicates as well as library(option) accept dicts as an alternative specification for options.
– The JSON support library provides alternative predicates to parse input into JSON represented using dicts. The JSON write predicates accept both the old representation and the new dict representation.

We have been using dicts internally for new code. The mailing list had a brief discussion on dicts.[10] There two contributed SWI-Prolog add-ons[11] that target lists: *dict_schema* and *sort_dict* address type checking and sorting lists of dictionaries by key. Other remarks:

– *Michael Hendricks*: "We're using V7. Most of our new code uses dicts extensively. I've found them especially compelling for working with JSON and for replacing option lists."
– *Wouter Beek*: Uses dictionaries for WebQR[12] for dealing with JSON messages. Claims that the code becomes more homogeneous, readable and shorter. Also uses the *real* add-on, claiming that the dot-notation, functions without argument and double quoted strings gives the mapping a more 'native' flavour and makes the code more readable.

Strings Many applications would both conceptually and performance-wise benefit from using strings. As long as most libraries return their textual data using atoms, we cannot expect serious uptake. Strings (and zero-argument compounds) are currently used by the R-interface package *real* [2].

5.1 Dict performance

The dict representation uses exactly twice the memory of a compound term, given the current implementation which uses an array of consecutive (key,value) pairs sorted by the key. Future implementation may share the key locations between dicts with the same keys. We compared the performance using a tight loop. The full implementation using functional notation is given in figure 1.

We give only the recursive clause for the other test cases in figure 2. Thus, **t0/2** provides the base case without extracting data, **t1/2** is the same as **tf/2** in figure 1, but without using functional notation and thus avoiding **./3**. The predicate **t2/2** uses **arg/3** to extract a field from the structured data represented as a compound term, **t3/2** uses a plain list of *Name*=*Value* and **t4/2** uses library(assoc), the SWI-Prolog implemented of which uses an AVL tree. Table 1 shows the performance of the four loops for 1,000,000 iterations, averaged over 10 runs on an Intel i7-3770 running Ubuntu 13.10 and SWI-Prolog 7.1.12.

[10] `http://swi-prolog.996271.n3.nabble.com/Some-thoughts-on-dicts-in-SWIPL-7-tt14327.html`
[11] `http://www.swi-prolog.org/pack/list`
[12] `https://github.com/wouterbeek/WebQR`

```
tf(Size, N) :-
        data(Size, D),
        tf2(N, Size, D).

tf2(0, _, _) :- !.
tf2(N, Q, D) :-
        Q1 is (N mod Q)+1,
        a(D.Q1),
        N2 is N - 1,
        tf2(N2, Q, D).

data(Size, D) :-
        numlist(1, Size, L),
        maplist(pair, L, Pairs),
        dict_pairs(D, _, Pairs).

pair(X, X-X).
```

Fig. 1. Benchmark program to evaluate dict lookup performance

Table 1 shows that the overhead of using dicts compared to compound terms is low (t1 vs. t2). The overhead of the functional notation is caused by type checking and checking for accessing a field vs. accessing a function on the dict in the predicate **./3**. This overhead could be removed if we had type inference. The last two predicates (t3, t4) show the performance of two classical Prolog solutions.

```
t0(N,Q,D) :- Q1 is (N mod Q)+1,                      a(x), ...
t1(N,Q,D) :- Q1 is (N mod Q)+1, get_dict(b,D,A),    a(A), ...
t2(N,Q,D) :- Q1 is (N mod Q)+1, arg(Q1,D,A),        a(A), ...
t3(N,Q,D) :- Q1 is (N mod Q)+1, memberchk(Q1=A,D), a(A), ...
t4(N,Q,D) :- Q1 is (N mod Q)+1, get_assoc(Q1,D,A), a(A), ...
```

Fig. 2. Alternative body (second clause of **tf2/3** in figure 1). The predicate a/1 is the dummy consumer of the data, defines as `a(_)`.

5.2 String performance

We evaluated the performance of text processing with the task to create the texts "test"*N* for *N* in 1..1,000,000. The results of these tests are presented in table 2. The core of the 4 loops is shown in figure 3. The predicate **tl2/1** has been added on request by one of the reviewers who claimed that **tl1/1** is unfair because it requires pushing the

		GC		
Test	CPUTime	Times	AvgSize	GCTime
t0(1000,1000000)	0.111	0	0	0.000
tf(1000,1000000)	0.271	259	18,229	0.014
t1(1000,1000000)	0.218	129	18,229	0.007
t2(1000,1000000)	0.165	129	10,221	0.006
t3(1000,1000000)	20.420	305	50,213	0.031
t4(1000,1000000)	3.968	1,299	50,213	0.149
t0(3,1000000)	0.113	0	0	0.000
tf(3,1000000)	0.232	259	2,277	0.011
t1(3,1000000)	0.181	129	2,277	0.005
t2(3,1000000)	0.166	129	2,245	0.005
t3(3,1000000)	0.277	189	2,357	0.009
t4(3,1000000)	0.859	346	2,397	0.014

Table 1. Performance test for accessing structured data using various representations. The first argument is the number of key-value pairs in the data and the second is the number of iterations in each test. **CPUTime** is the CPU time in seconds. The **GC** columns represent heap garbage collection statistics, showing the number of garbage collections, the average size of the heap *after* GC and the time spent on GC in seconds.

list `‘list‘` onto the stacks on each iteration and the SWI-Prolog implementation of **append/2** performs a type-check on the first argument, making it relatively slow. We claim that **tl1/1** is a more natural translation of the other tests. The test `c`(*Threads,Goal*) runs *Threads* copies of *Goal*, each in their own threads, while the main thread waits for the completion of all threads. The tests were executed on a (quad core) Intel i7-3770 machine running Ubuntu 13.10 and SWI-Prolog 7.1.15.

```
t0(N)    :- dummy([test,N],_), N2 is N-1, t0(N2).
ta(N)    :- atomic_list_concat([test,N],_), N2 is N-1, ta(N2).
ts(N)    :- atomics_to_string(["test",N],_), N2 is N-1, ts(N2).
tl1(N)   :- number_codes(N,S), append([‘test‘,S],_),
            N2 is N-1, tl(N2).
tl2(N)   :- tl2(N,‘test‘).
tl2(N,P) :- number_codes(N,S), append(P,S,_),
            N2 is N-1, tl(N2).
dummy(_,_).
```

Fig. 3. Benchmarks for comparing concatenation of text in various formats.

We realise that the above performance analysis is artificial and limited. The tests only analyse *construction*, not processing text in the various representations. Notably atoms

	Time			**GC**			**Atom GC**		
Test	Process	Thread	Wall	Times	AvgWorkSet	GCTime	Times	Reclaimed	AGCTime
t0(1000000)	0.058	0.058	0.058	786	2,186	0.009	0	0	0.000
ta(1000000)	0.316	0.316	0.316	785	2,146	0.013	99	10,884,136	0.042
ts(1000000)	0.214	0.214	0.214	1,703	2,190	0.023	0	0	0.000
tl1(1000000)	1.051	1.051	1.051	8,570	2,267	0.108	0	0	0.000
tl2(1000000)	0.437	0.437	0.437	3,893	2,231	0.077	0	0	0.000
c(4,t0(1000000))	0.252	0.000	0.065	0	0	0.000	0	0	0.000
c(4,ta(1000000))	6.300	0.000	1.924	0	0	0.000	332	36,442,981	0.227
c(4,ts(1000000))	0.886	0.000	0.232	0	0	0.000	0	0	0.000
c(4,tl1(1000000))	4.463	0.000	1.143	0	0	0.000	0	0	0.000
c(4,tl2(1000000))	1.731	0.000	0.441	0	0	0.000	0	0	0.000

Table 2. Comparing atom and string handling performance. The **Time** columns represent the time spent by the process, calling thread and the wall time in seconds. The **GC** columns are described with table 1. The **AtomGC** columns represent the atom garbage collector, showing the number of invocations, number of reclaimed atoms and the time spent in seconds. Note that the GC values for the concurrent tests are all zero because GC is monitored in the main thread, which just waits for the others to complete.

and strings are internally represented as arrays and thus provide O(1) access to the i-th character, but lists allow splitting in head and tail cheaply and can exploit DCGs.

Table 2 makes us draw the following conclusions regarding construction of a short text from short pieces:

- To our surprise, constructing text as atoms is faster than using lists of codes.
- Strings are constructed considerably faster than atoms.
- Atom handling significantly harms performance of multi-threaded application. Disabling atom garbage collection and looking at the internal contention statistics indicate that the slowdown is caused by contention on the mutex that guards the atom table.

6 Conclusions

With SWI-Prolog version 7 we have decided to narrow the gap between Prolog and other key components in the IT infrastructure by introducing commonly found data types and harmonizing the syntax with modern languages. Besides more transparent interfacing, these changes are also aimed at simplifying the transition from other languages. The most important changes are the introduction of dicts (key-value sets) as primary citizens with access to keys using functional notation, the introduction of strings, including the common double-quoted notation and an unambiguous representation of lists. Previous changes added [. . .] and {. . . } as operators and introduced quasi quotations. These extensions aim at smooth exchange of data with other IT infrastructure, a natural syntax for accessing structured data and the ability to define syntax for DSLs that is more natural to those not familiar with Prolog's history.

Acknowledgements

The described extensions result for a large part from interaction with users and other Prolog developers. Nicos Angelopoulos pointed at the issues interacting with the R-project. Michiel Hildebrand and Jacco van Ossenbruggen helped formulating the issues with JavaScript and JSON. Jose Morales suggested the 'block operators' [...] and {...}. Michael Hendricks brought quasi quotations to our attention. Manuel Hermenegildo and Bart Demoen supported the idea to start moving during the ICLP in Istanbul. Joachim Schimpf helped achieving a shared set of string operations with ECLiPSe and provided a test suite for them. Vitor Santos Costa commented on several of the ideas and is likely to implement most of them for YAP after the dust has settled. Other people that commented on the drafts include Richard O'Keefe, Ulrich Neumerkel, Paulo Moura, Feliks Kluzniak, Alan Baljeu, Kilian Evang, Shon Feder, Jeff Schultz and Carlo Capelli.

References

1. Hassan Ait-Kaci. An overview of LIFE. In *Next generation information system technology*, pages 42–58. Springer, 1991.
2. Nicos Angelopoulos, Vítor Santos Costa, João Azevedo, Jan Wielemaker, Rui Camacho, and Lodewyk Wessels. Integrative functional statistics in logic programming. In Konstantinos F. Sagonas, editor, *PADL*, volume 7752 of *Lecture Notes in Computer Science*, pages 190–205. Springer, 2013.
3. Gosse Bouma, Gertjan Van Noord, and Robert Malouf. Alpino: Wide-coverage computational analysis of dutch. *Language and Computers*, 37(1):45–59, 2001.
4. Amadeo Casas, Daniel Cabeza, and Manuel V Hermenegildo. A syntactic approach to combining functional notation, lazy evaluation, and higher-order in lp systems. In *Functional and Logic Programming*, pages 146–162. Springer, 2006.
5. Christopher Ireland, David Bowers, Michael Newton, and Kevin Waugh. A classification of object-relational impedance mismatch. In *Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA'09. First International Conference on*, pages 36–43. IEEE, 2009.
6. Micha Meier and Joachim Schimpf. An architecture for prolog extensions. In *Extensions of Logic Programming*, pages 319–338. Springer, 1993.
7. Jan Wielemaker and Nicos Angelopoulos. Syntactic integration of external languages in Prolog. In *Proceedings of WLPE 2012*, 2012.
8. Jan Wielemaker and Michael Hendricks. Why It's Nice to be Quoted: Quasiquoting for Prolog. *CoRR*, abs/1308.3941, 2013.

A Parallel Virtual Machine for Executing Forward-Chaining Linear Logic Programs

Flavio Cruz[1 2], Ricardo Rocha[2], and Seth Copen Goldstein[1]

[1] Carnegie Mellon University, Pittsburgh, PA 15213, USA
{fmfernan, seth}@cs.cmu.edu
[2] CRACS & INESC TEC, Faculty of Sciences, University Of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{flavioc, ricroc}@dcc.fc.up.pt

Abstract. Linear Meld is a concurrent forward-chaining linear logic programming language where logical facts can be asserted and retracted in a structured way. The database of facts is partitioned by the nodes of a graph structure which leads to parallelism if nodes are executed simultaneously. Communication arises whenever nodes send facts to other nodes by fact derivation. We present an overview of the virtual machine that we implemented to run Linear Meld on multicores, including code organization, thread management, rule execution and database organization for efficient fact insertion, lookup and deletion. Although our virtual machine is a work-in-progress, our results already show that Linear Meld is not only capable of scaling graph and machine learning programs but it also exhibits some interesting performance results when compared against other programming languages.

Keywords: linear logic programming, virtual machine, implementation

1 Introduction

Logic programming is a declarative programming paradigm that has been used to advance the state of parallel programming. Since logic programs are declarative, they are much easier to parallelize than imperative programs. First, logic programs are easier to reason about since they are based on logical foundations. Second, logic programmers do not need to use low level programming constructs such as locks or semaphores to coordinate parallel execution, because logic systems hide such details from the programmer.

Logic programming languages split into two main fields: *forward-chaining* and *backwards-chaining* programming languages. Backwards-chaining logic programs are composed of a set of rules that can be activated by inputing a query. Given a query $q(\hat{x})$, an interpreter will work backwards by matching $q(\hat{x})$ against the head of a rule. If found, the interpreter will then try to match the body of the rule, recursively, until it finds the program axioms (rules without body). If the search procedure succeeds, the interpreter finds a valid substitution for the $\hat{x}$ variables. A popular backwards-chaining programming language is Prolog [4], which has been a productive research language for executing logic programs in parallel. Researchers took advantage of Prolog's non-determinism to evaluate subgoals in parallel with models such as *or-parallelism* and *and-parallelism* [8].

In a forward-chaining logic programming language, we start with a database of facts (filled with the program's axioms) and a set of logical rules. Then, we use the facts of the database to fire the program's rules and derive new facts that are then added to the database. This process is repeated until the database reaches *quiescence* and no more information can be derived from the program. A popular forward-chaining programming language is Datalog [14].

In this paper, we present a new forward-chaining logic programming language called Linear Meld (LM) that is specially suited for concurrent programming over graphs. LM differs from Datalog-like languages because it integrates both classical logic and linear logic [6] into the language, allowing some facts to be retracted and asserted logically. Although most Datalog and Prolog-like programming languages allow some kind of state manipulation [11], those features are extra-logical, reducing the advantages brought forward by logic programming. In LM, since mutable state remains within the logical framework, we can reason logically about LM programs.

The roots of LM are the P2 system [12] and the original Meld [2,1]. P2 is a Datalog-like language that maps a computer network to a graph, where each computer node performs local computations and communicates with neighbors. Meld is itself inspired by the P2 system but adapted to the concept of massively distributed systems made of modular robots with a dynamic topology. LM also follows the same graph model of computation, but, instead, applies it to parallelize graph-based problems such as graph algorithms, search algorithms and machine learning algorithms. LM programs are naturally concurrent since the graph of nodes can be partitioned to be executed by different workers.

To realize LM, we have implemented a compiler and a virtual machine that executes LM programs on multicore machines[3]. We have implemented several parallel algorithms, including: belief propagation [7], belief propagation with residual splash [7], PageRank, graph coloring, N-Queens, shortest path, diameter estimation, map reduce, quick-sort, neural network training, among others.

As a forward-chaining linear logic programming language, LM shares similarities with Constraint Handling Rules (CHR) [3,10]. CHR is a concurrent committed-choice constraint language used to write constraint solvers. A CHR program is a set of rules and a set of constraints (which can be seen as facts). Constraints can be consumed or generated during the application of rules. Some optimization ideas used in LM such as join optimizations and using different data structures for indexing facts are inspired by research done in CHR [9].

This paper describes the current implementation of our virtual machine and is organized as follows. First, we briefly introduce the LM language. Then, we present an overview of the virtual machine, including code organization, thread management, rule execution and database organization. Finally, we present preliminary results and outline some conclusions.

[3] Source code is available at `http://github.com/flavioc/meld`.

2 The LM Language

Linear Meld (LM) is a logic programming language that offers a declarative and structured way to manage state. A program consists of a database of facts and a set of derivation rules. The database includes persistent and linear facts. Persistent facts cannot be deleted, while linear facts can be asserted and retracted.

The dynamic (or operational) semantics of LM are identical to Datalog. Initially, we populate the database with the *program's axioms* (initial facts) and then determine which derivation rules can be applied using the current database. Once a rule is applied, we derive new facts, which are then added to the database. If a rule uses linear facts, they are retracted from the database. The program stops when *quiescence* is achieved, that is, when rules no longer apply.

Each fact is a predicate on a tuple of *values*, where the type of the predicate prescribes the types of the arguments. LM rules are type-checked using the predicate declarations in the header of the program. LM has a simple type system that includes types such as *node*, *int*, *float*, *string*, *bool*. Recursive types such as *list X* and *pair X; Y* are also allowed. Each rule in LM has a defined priority that is inferred from its position in the source file. Rules at the beginning of the file have higher priority. We consider all the new facts that have been not used yet to create a set of *candidate rules*. The set of candidate rules is then applied (by priority) and updated as new facts are derived.

2.1 Example

We now present an example LM program in Fig. 1 that implements the key update operation for a binary tree represented as a key/value dictionary. We first declare all the predicates (lines 1-4), which represent the kinds of facts we are going to use. Predicate `left/2` and `right/2` are persistent while `value/3` and `replace/3` are linear. The `value/3` predicate assigns a key/value pair to a binary tree node and the `replace/3` predicate represents an update operation that updates the key in the second argument to the value in the third argument.

The algorithm uses three rules for the three cases of updating a key's value: the first rule performs the update (lines 6-7); the second rule recursively picks the left branch for the update operation (lines 9-10); and the third rule picks the right branch (lines 12-13). The axioms of this program are presented in lines 15-22 and they describe the initial binary tree configuration, including keys and values. By having the `update(@3, 6, 7)` axiom instantiated at the root node @3, we intend to change the value of key 6 to 7. Note that when writing rules or axioms, persistent facts are preceded with a `!`.

Figure 2 represents the trace of the algorithm. Note that the program database is partitioned by the tree nodes using the first argument of each fact. In Fig. 2a we present the database filled with the program's axioms. Next, we follow the right branch using rule 3 since $6 > 3$ (Fig. 2b). We then use the same rule again in Fig. 2c where we finally reach the key 6. Here, we apply rule 1 and `value(@6, 6, 6)` is updated to `value(@6, 6, 7)`.

```
type left(node, node).
type right(node, node).
type linear value(node, int, int).
type linear replace(node, int, int).

replace(A, K, New), value(A, K, Old)
   -o value(A, K, New). // we found our key

replace(A, RKey, RValue), value(A, Key, Value), !left(A, B), RKey < Key
   -o value(A, Key, Value), replace(B, RKey, RValue). // go left

replace(A, RKey, RValue), value(A, Key, Value), !right(A, B), RKey > Key
   -o value(A, Key, Value), replace(B, RKey, RValue). // go right

// binary tree configuration
value(@3, 3, 3). value(@1, 1, 1). value(@0, 0, 0).
value(@2, 2, 2). value(@5, 5, 5). value(@4, 4, 4).
value(@6, 6, 6).
!left(@1, @0). !left(@3, @1). !left(@5, @4).
!right(@1, @2). !right(@3, @5). !right(@5, @6).

replace(@3, 6, 7). // replace value of key 6 to 7
```

Fig. 1: Binary tree dictionary: replacing a key's value.

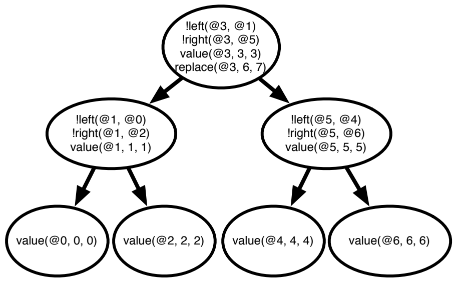
2.2 Syntax

Table 1 shows the abstract syntax for rules in LM. An LM program *Prog* consists of a set of derivation rules Σ and a database D. Each derivation rule R can be written as $BE \multimap HE$ where BE is the body of a rule and HE is the head. Rules without bodies are allowed in LM and they are called *axioms*. Rules without heads are specified using 1 as the rule head. The body of a rule, BE, may contain linear (L) and persistent (P) *fact expressions* and constraints (C). Fact expressions are template facts that instantiate variables from facts in the database. Such variables are declared using either $\forall_x.R$ or $\exists_x.BE$. If using $\forall_x.R$ variables can also be used in the head of the rule. Constraints are boolean expressions that must be true in order for the rule to be fired. Constraints use variables from fact expressions and are built using a small functional language that includes mathematical operations, boolean operations, external functions and literal values. The head of a rule, HE, contains linear (L) and persistent (P) *fact templates* which are uninstantiated facts to derive new facts. The head can also have *comprehensions* (CE) and *aggregates* (AE). Head expressions may use the variables instantiated in the body.

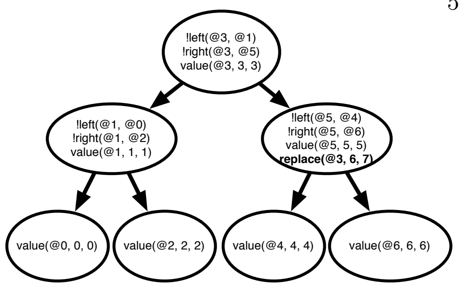
Comprehensions Sometimes we need to consume a linear fact and then immediately generate several facts depending on the contents of the database. To solve this particular need, we created the concept of comprehensions, which are subrules that are applied with all possible combinations of facts from the database. In a comprehension $\{\ \widehat{x};\ BE;\ SH\ \}$, $\widehat{x}$ is a list of variables, BE is the body of the comprehension and SH is the head. The body BE is used to generate all possible combinations for the head SH, according to the facts in the database.

The following example illustrates a simple program that uses comprehensions:

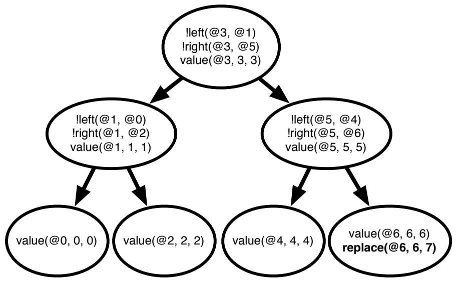
```
!edge(@1, @2).
!edge(@1, @3).
```



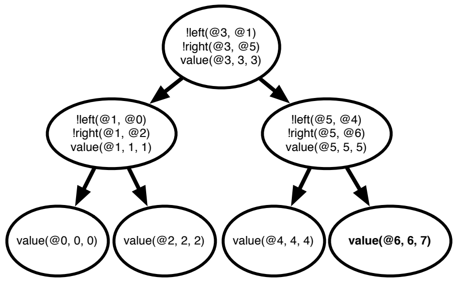
(a) Initial database. Replace axiom instantiated at root node @3.



(b) After applying rule 3 at node @3. Replace fact sent to node @5.



(c) After applying rule 3 at node @5. Replace fact reaches node @6.



(d) After applying rule 1 at node @6. Value of key 6 has changed to 7.

Fig. 2: An execution trace for the binary tree dictionary algorithm.

```
iterate(@1).
iterate(A) -o {B | !edge(A, B) | perform(B)}.
```

When the rule is fired, we consume `iterate(@1)` and then generate the comprehension. Here, we iterate through all the `edge/2` facts that match `!edge(@1, B)`, which are: `!edge(@1, @2)` and `!edge(@1, @3)`. For each fact, we derive `perform(B)`, namely: `perform(@2)` and `perform(@3)`.

Aggregates Another useful feature in logic programs is the ability to reduce several facts into a single fact. In LM we have aggregates (AE), a special kind of sub-rule that works very similarly to comprehensions. In the abstract syntax $[\ A \ \Rightarrow \ y;\ \widehat{x};\ BE;\ SH_1;\ SH_2\ ]$, A is the aggregate operation, $\widehat{x}$ is the list of variables introduced in BE, SH_1 and SH_2 and y is the variable in the body BE that represents the values to be aggregated using A. Like comprehensions, we use $\widehat{x}$ to try all the combinations of BE, but, in addition to deriving SH_1 for each combination, we aggregate the values represented by y and derive SH_2 only once using y. As an example, consider the following program:

```
price(@1, 3).
price(@1, 4).
price(@1, 5).
count-prices(@1).
count-prices(A) -o [sum => P | . | price(A, P) | 1 | total(A, P)].
```

Program	$Prog$	$::= \Sigma, D$
Set Of Rules	Σ	$::= \cdot \mid \Sigma, R$
Database	D	$::= \Gamma; \Delta$
Rule	R	$::= BE \multimap HE \mid \forall_x.R$
Body Expression	BE	$::= L \mid P \mid C \mid BE, BE \mid \exists_x.BE \mid 1$
Head Expression	HE	$::= L \mid P \mid HE, HE \mid CE \mid AE \mid 1$
Linear Fact	L	$::= l(\hat{x})$
Persistent Fact	P	$::= !p(\hat{x})$
Constraint	C	$::= c(\hat{x})$
Comprehension	CE	$::= \{\ \widehat{x};\ BE;\ SH\ \}$
Aggregate	AE	$::= [\ A\ \Rightarrow\ y;\ \widehat{x};\ BE;\ SH_1;\ SH_2\ ]$
Aggregate Operation	A	$::=$ `min` $\mid$ `max` $\mid$ `sum` $\mid$ `count`
Sub-Head	SH	$::= L \mid P \mid SH, SH \mid 1$
Known Linear Facts	Δ	$::= \cdot \mid \Delta, l(\hat{t})$
Known Persistent Facts	Γ	$::= \cdot \mid \Gamma, !p(\hat{t})$

Table 1: Abstract syntax of LM.

By applying the rule, we consume `count-prices(@1)` and derive the aggregate which consumes all the `price(@1, P)` facts. These are summed and `total(@1, 12)` is derived. LM provides several aggregate operations, including the *minimum*, *maximum*, *sum*, and *count*.

2.3 Concurrency

LM is at its core a concurrent programming language. The database of facts can be seen as a graph data structure where each node contains a fraction of the database. To accomplish this, we force the first argument of each predicate to be typed as a *node*. We then restrict the derivation rules to only manipulate facts belonging to a single node. However, the expressions in the head may refer to other nodes, as long as those nodes are instantiated in the body of the rule.

Due to the restrictions on LM rules, nodes are able to run rules independently without using other node's facts. Node computation follows a *don't care* or *committed choice* non-determinism since any node can be picked to run as long as it contains enough facts to fire a derivation rule. Facts coming from other nodes will arrive in order of derivation but may be considered partially and there is no particular order among the neighborhood. To improve concurrency, the programmer is encouraged to write rules that take advantage of the non-deterministic nature of execution.

3 The Virtual Machine

We developed a compiler that compiles LM programs to byte-code and a multi-threaded virtual machine (VM) using POSIX threads to run the byte-code. The goal of our system is to keep the threads as busy as possible and to reduce inter-thread communication.

The load balancing aspect of the system is performed by our work scheduler that is based on a simple work stealing algorithm. Initially, the system will partition the application graph of N nodes into P subgraphs (the number of threads) and then each thread will work on their own subgraph. During execution, threads can steal nodes of other threads to keep themselves busy.

Reduction of inter-thread communication is achieved by first ordering the node addresses present in the code in such a way that closer nodes are clustered together and then partitioning them to threads. During compilation, we take note of predicates that are used in communication rules (arguments with type *node*) and then build a graph of nodes from the program's axioms. The nodes of the graph are then ordered by using a breadth-first search algorithm that changes the nodes of addresses to the domain $[0, n[$, where n is the number of nodes. Once the VM starts, we simply partition the range $[0, n[$.

Multicore When the VM starts, it reads the byte-code file and starts all threads. As a first step, all threads will grab their own nodes and assign the `owner` property of each. Because only one thread is allowed to do computation on a node at any given time, the owner property defines the thread with such permission. Next, each thread fills up its *work queue* with the initial nodes. This queue maintains the nodes that have new facts to be processed. When a node sends a fact to another node, we need to check if the target node is owned by the same thread. If that is not the case, then we have a point of synchronization and we may need to make the target thread active.

The main thread loop is shown in Fig. 3, where the thread inspects its work queue for active nodes. Procedure `process_node()` takes a node with new candidate rules and executes them. If the work queue is empty, the thread attempts to steal one node from another thread before becoming idle. Starting from a random thread, it cycles through all the threads to find one active node. Eventually, there will be no more work to do and the threads will go idle. There is a global atomic counter, a global boolean flag and one boolean flag for each thread that are used to detect termination. Once a thread goes idle, it decrements the global counter and changes its flag to idle. If the counter reaches zero, the global flag is set to idle. Since every thread will be busy-waiting and checking the global flag, they will detect the change and exit the program.

Byte-Code A byte-code file contains meta-data about the program's predicates, initial nodes, partitioning information, and code for each rule. Each VM thread has 32 registers that are used during rule execution. Registers can store facts, integers, floats, node addresses and pointers to runtime data structures (lists and structures). When registers store facts, we can reference fields in the fact through the register.

Consider a rule `!a(X,Y), b(X,Z), c(X,Y) -o d(Y)` and a database with `!a(1,2)`, `!a(2,3)`, `b(1,3)`, `b(5,3)`, `c(1,2)`, `c(1,3)`, `c(5,3)`. Rule execution

```
void work_loop(thread_id tid):
   while (true):
      current_node = NULL;
      if(has_work(tid)):
         current_node = pop_work(tid); // take node from the queue
      else:
         target_thread = random(NUM_THREADS);
         for (i = 0; i < NUM_THREADS && current_node == NULL; ++i): // need to steal a node
            target_thread = (target_thread + 1) % NUM_THREADS;
            current_node = steal_node_from_thread(target_thread)
      if(current_node == NULL):
         become_idle(tid);
         if(!synchronize_termination(tid)):
            return;
         become_active(tid);
      else:
         process_node(current_node, tid);
```

Fig. 3: Thread work loop.

proceeds in a series of recursive loops, as follows: the first loop retrieves an iterator for the persistent facts of `!a/2` and moves the first valid fact, `!a(1,2)`, to register 0; the inner loop retrieves linear facts that match `b(1,Z)` (from the *join constraint*) and moves `b(1,3)` to register 1; in the final loop we move `c(1,2)` to register 2 and the body of the rule is successfully matched. Next, we derive `d(2)`, where `2` comes from register 0. Fig. 4 shows the byte-code for this example.

```
PERSISTENT ITERATE a MATCHING TO reg 0
  LINEAR ITERATE b MATCHING TO reg 1
      (match).0=0.0
    LINEAR ITERATE c MATCHING TO reg 2
        (match).0=0.0
        (match).1=0.1
      ALLOC d TO reg 3
      MVFIELDFIELD 0.1 TO 3.0
      ADDLINEAR reg 3
      REMOVE reg 2
      REMOVE reg 1
      TRY NEXT
    NEXT
  NEXT
RETURN
```

Fig. 4: Byte-code for rule `!a(X,Y), b(X,Z), c(X,Y) -o d(Y).`

In case of failure, we jump to the previous outer loop in order to try the next candidate fact. If a rule matches and the head is derived, we backtrack to the inner most *valid loop*, i.e., the first inner loop that uses linear facts or, if there are no linear facts involved, to the previous inner loop. We need to jump to a valid loop because we may have loops with linear facts that are now invalid. In our example, we would jump to the loop of `b(X,Z)` and not `c(X,Y)`, since `b(1,3)` was consumed.

The compiler re-orders the fact expressions used in the body in order to make execution more efficient. For example, it forces the join constraints in rules to appear at the beginning so that matching will fail sooner rather than later. It also does the same for constraints. Note that for every loop, the compiler adds a *match object*, which contains information about which arguments need to match, so that runtime matching is efficient.

Our compiler also detects cases where we re-derive a linear fact with new arguments. For example, as shown in Fig. 5, the rule `a(N) -o a(N+1)` will compile to code that reuses the old `a(N)` fact. We use a `flags` field to mark updated nodes (presented next).

```
LINEAR ITERATE a MATCHING TO reg 0
  MVFIELDREG 0.0 TO reg 1
  MVINTREG INT 1 TO reg 2
  reg 1 INT PLUS reg 2 TO reg 3
  MVREGFIELD reg 3 TO 0.0
  UPDATE reg 0
  TRY NEXT
RETURN
```

Fig. 5: Byte-code for rule `a(N) -o a(N+1)`.

Database Data Structures We said before that LM rules are constrained by the first argument. Because nodes can be execute independently, our database is indexed by the node address and each sub-database does not need to deal with synchronization issues since at any given point, only one thread will be using the database. Note that the first argument of each fact is not stored.

The database must be implemented efficiently because during matching of rules we need to restrict the facts using a given match object, which fixes arguments of the target predicate to instantiated values. Each sub-database is implemented using three kinds of data structures:

- *Tries* are used exclusively to store persistent facts. Tries are trees where facts are indexed by the common arguments.
- *Doubly Linked Lists* are used to store linear facts. We use a double linked list because it is very efficient to add and remove facts.
- *Hash Tables* are used to improve lookup when linked lists are too long and when we need to do search filtered by a fixed argument. The virtual machine decides which arguments are best to be indexed (see "Indexing") and then uses a hash table indexed by the appropriate argument. If we need to go through all the facts, we just iterate through all the facts in the table. For collisions, we use the above doubly linked list data structure.

Figure 6 shows an example for a hash table data structure with 3 linear facts indexed by the second argument and stored as doubly linked list in bucket 2. Each linear fact contains the regular list pointers, a `flags` field and the fact arguments. Those are all stored continuously to improve data locality. One use of the `flags` field is to mark that a fact is already being used. For example, consider the rule body `a(A,B), a(C,D) -o ...`. When we first pick a fact for `a(A, B)` from the hash table, we mark it as being used in order to ensure that, when we retrieve facts for `a(C, D)`, the first one cannot be used since that would violate linearity.

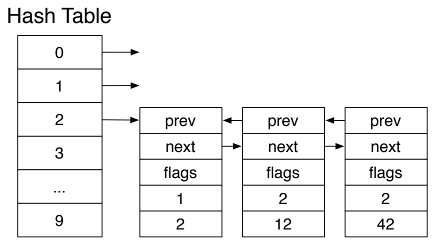
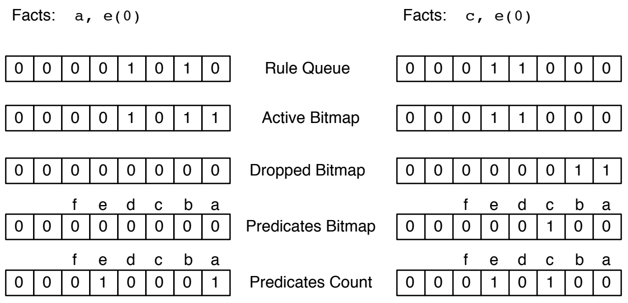


Fig. 6: Hash table data structure for storing predicate `a(int,int)`.

Rule Engine The rule engine decides which rules may need to be executed while taking into account rule priorities. There are 5 main data structures for scheduling rule execution; `Rule Queue` is the bitmap representing the rules that will be run; `Active Bitmap` contains the rules that can be fired since there are enough facts to activate the rule's body; `Dropped Bitmap` contains the rules that must be dropped from `Rule Queue`; `Predicates Bitmap` marks the newly derived facts; and `Predicates Count` counts the number of facts per predicate. To understand how our engine works, consider the program in Fig. 7a.

```
a, e(1) -o b.
a -o c.
b -o d.
e(0) -o f.
c -o e(1).
```

(a) Program



(b) Before and after applying the 2nd rule.

Fig. 7: Program and rule engine data structures.

We take the least significant rule from the `Rule Queue` bitmap, which is the candidate rule with the higher priority, and then run it. In our example, since we have facts `a` and `e(0)`, we will execute the second rule `a -o c`. Because the derivation is successful, we will consume `a` and derive `c`. We thus mark the `c` predicate in the `Predicates Bitmap` and the first and second rules in `Dropped Bitmap` since such rules are no longer applicable (`a` is gone). To update the `Rule Queue`, we remove the bits marked in `Dropped Bitmap` and add the active rules marked in `Active Bitmap` that are affected by predicates in `Predicates Bitmap`. The engine thus schedules the fourth and fifth rules to run (Fig. 7b).

Note that every node in the program has the same set of data structures present in Fig. 7. We use 32 bits integers to implement bitmaps and an array of 16 bits integers to count facts, resulting in $32 + 2P$ bytes per node, where P is the number of predicates.

We do a small optimization to reduce the number of derivations of persistent facts. We divide the program rules into two sets: *persistent rules* and *non persistent rules*. Persistent rules are rules where only persistent facts are involved. We compile such rules incrementally, that is, we attempt to fire all rules where a persistent fact is used. This is called the *pipelined semi-naive* evaluation and it originated in the P2 system [12]. This evaluation method avoids excessing rederivations of the same fact. The order of derivation does not matter for those rules, since only persistent facts are used.

Thread Interaction Whenever a new fact is derived through rule derivation, we need to update the data structures for the corresponding node. This is trivial if the thread that owns the node derived the fact also. However, if a thread $T1$ derives a fact for a node owned by another thread $T2$, then we may have problems because $T2$ may be also updating the same data structures. We added a lock and a boolean flag to each node to protect the access to its data structures. When a node starts to execute, we activate the flag and lock the node. When another thread tries to use the data structures, if first checks the flag and if not activated, it locks the node and performs the required updates. If the flag is activated, it stores the new fact in a list to be processed before the node is executed.

Indexing To improve fact lookup, the VM employs a fully dynamic mechanism to decide which argument may be optimal to index. The algorithm is performed in the beginning of the execution and empirically tries to assess the argument of each predicate that more equally spreads the database across the values of the argument. A single thread performs the algorithm for all predicates.

The indexing algorithm is performed in three main steps. First, it gathers statistics of lookup data by keeping a counter for each predicate's argument. Every time a fact search is performed where arguments are fixed to a value, the counter of such arguments is incremented. This phase is performed during rule execution for a small fraction of the nodes in the program.

The second step of the algorithm then decides the candidate arguments of each predicate. If a predicate was not searched with any fixed arguments, then it will be not indexed. If only one argument was fixed, then such argument is set as the indexing argument. Otherwise, the top 2 arguments are selected for the third phase, where *entropy statistics* are collected dynamically.

During the third phase, each candidate argument has an entropy score. Before a node is executed, the facts of the target predicate are used in the following formula applied for the two arguments:

$$Entropy(A, F) = -\sum_{v \in values(F,A)} \frac{count(F, A = v)}{total(F)} \log_2 \frac{count(F, A = v)}{total(F)}$$

Where A is the target argument, F is the multi-set of linear facts for the target predicate, $values(F, A)$ is set of values of the argument A, $count(F, A = v)$ counts the number of linear facts where argument A is equal to v and $total(F)$ counts the number of linear facts in F. The entropy value is a good metric because it tells us how much information is needed to describe an argument. If more information is needed, then that must be the best argument to index.

For one of the arguments to score, $Entropy(A, F)$ multiplied by the number of times it has been used for lookup must be larger than the other argument.

The argument with the best score is selected and then a global variable called `indexing_epoch` is updated. In order to convert the node's linked lists into hash tables, each node also has a local variable called `indexing_epoch` that is compared to the global variable in order to rebuild the node database according to the new indexing information.

Our VM also dynamically resizes the hash table if necessary. When the hash table becomes too dense, it is resized to the double. When it becomes too sparse, it is reduced in half or simply transformed back into a doubly linked list. This is done once in a while, before a node executes.

We have seen very good results with this scheme. For example, for the all-pairs shortest paths program, we obtained a 2 to 5-fold improvement in sequential execution time. The overhead of dynamic indexing is negligible since programs run almost as fast as if the indices have been added from the start.

4 Preliminary Results

This section presents preliminary results for our VM. First, we present scalability results in order to show that LM programs can take advantage of multicore architectures. Next, we present a comparison with similar programs written in other programming languages in order to show evidence that our VM is viable.

For our experimental setup, we used a machine with two 16 (32) Core AMD Opteron (tm) Processor 6274 @ 2.2 GHz with 32 GBytes of RAM memory and running the Linux kernel 3.8.3-1.fc17.x86_64. We compiled our VM using GCC 4.7.2 (g++) with the flags `-O3 -std=c+0x -march=x86-64`. We ran all experiments 3 times and then averaged the execution time.

Scalability Results For this section, we run each program using 1, 2, 4, 6, 8, 10, 12, 14 and 16 threads and compared the runtime against the execution of the sequential version of the VM. We used the following programs:

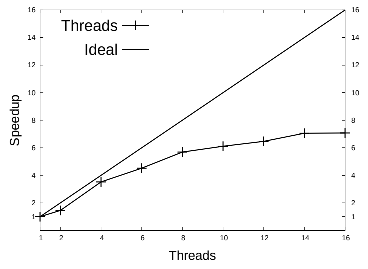
– Greedy Graph Coloring (GGC) colors nodes in a graph so that no two adjacent nodes have the same color. We start with a small number of colors and then we expand the number of colors when we cannot color the graph.
– PageRank implements a PageRank algorithm without synchronization between iterations. Every time a node sends a new rank to its neighbors and the change was significant, the neighbors are scheduled to recompute their ranks.
– N-Queens, the classic puzzle for a 13x13 board.
– Belief Propagation, a machine learning algorithm to denoise images.

Figure 8 presents the speedup results for the GGC program using 2 different datasets. In Fig. 8a we show the speedup for a search engine graph of 12,000 webpages[4]. Since this dataset follows the power law, that is, there is a small number of pages with a lots of links (1% of the nodes have 75% of the edges), the speedup is slightly worse than the benchmark shown in Fig. 8b, where we use a random dataset of 2,000 nodes with an uniform distribution of edges.

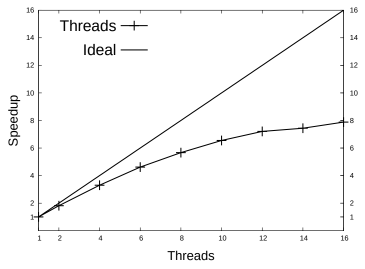
The PageRank results are shown in Fig. 9. We used the same search engine dataset as before and a new random dataset with 5,000 nodes and 500,000 edges. Although the search engine graph (Fig. 9a) has half the edges (around 250,000), it scales better than the random graph (Fig. 9b), meaning that the PageRank program depends on the number of nodes to be more scalable.

The results for the N-Queens program are shown in Fig. 10a. The program is not regular since computation starts at the top of the grid and then rolls down, until only the last row be doing computation. Because the number of valid states for the nodes in the upper rows is much less than the nodes in the lower rows, this may potentially lead to load balancing problems. The results show that our system is able to scale well due to work stealing.

[4] Available from `http://www.cs.toronto.edu/~tsap/experiments/download/download.html`

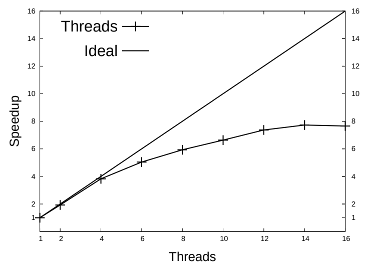


(a) Using a graph of web pages with around 12,000 nodes and 292,000 edges.

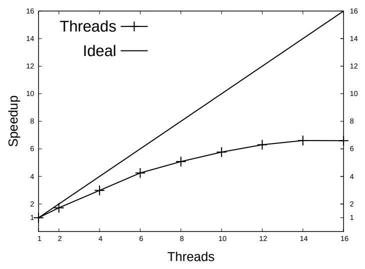


(b) Using a random graph with 2,000 nodes and 600,000 edges.

Fig. 8: Experimental results for the GGC algorithm.



(a) Using a graph of web pages collected from a search engine (around 12,000 nodes and 292,000 edges)



(b) Using a random graph with 5,000 nodes and 500,000 edges.

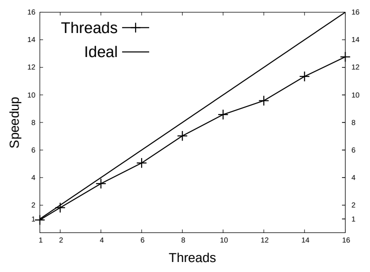
Fig. 9: Experimental results for the asynchronous PageRank algorithm.

Finally, we shown the results for the Belief Propagation (BP) program in Fig. 10b. BP is a regular and asynchronous program and benefits (as expected) from having multiple threads executing since the belief values of each node will converge faster. The super-linear results prove this assertion.

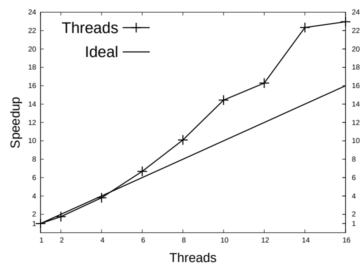
Absolute Execution Time As we have seen, our VM scales reasonably well, but how does it compare in terms of absolute execution time with other competing systems? We next present such comparison for the execution time using one thread.

In Fig. 11a we compare the LM's N-Queens version against 3 other versions: a straightforward sequential program implemented in C using backtracking; a sequential Python [15] implementation; and a Prolog implementation executed in YAP Prolog [5], an efficient implementation of Prolog. Numbers less than 1 mean that LM is faster and larger than 1 mean that LM is slower. We see that LM easily beats Python, but is 5 to 10 times slower than YAP and around 15 times slower than C. However, note that if we use at least 16 threads in LM, we can beat the sequential implementation written in C.

In Fig. 11b we compare LM's Belief Propagation program against a sequential C version, a Python version and a GraphLab version. GraphLab [13] is a parallel C++ library used to solve graph-based problems in machine learning. C and GraphLab perform about the same since both use C/C++. Python runs



(a) N-Queens program (13x13 board).



(b) BP (400x400 image).

Fig. 10: Experimental Results for N-Queens and Belief Propagation.

Size	C	Python	YAP Prolog
10	16.92	**0,62**	5,42
11	21.59	**0.64**	6.47
12	10.32	**0.73**	7.61
13	14.35	**0.88**	10.38

(a) N-Queens problem.

Size	C	Python	GraphLab
10	**0.67**	**0,03**	**1.00**
50	*1.77*	**0.04**	*1.73*
200	*1.99*	**0.05**	*1.79*
400	*2.00*	**0.04**	*1.80*

(b) Belief Propagation program.

Fig. 11: Comparing absolute execution times (System Time / LM Time).

very slowly since it is a dynamic programming language and BP has many mathematical computations. We should note, however, that the LM version uses some external functions written in C++ in order to improve execution time, therefore the comparison is not totally fair.

We also compared the PageRank program against a similar GraphLab version and LM is around 4 to 6 times slower. Finally, our version of the all-pairs shortest distance algorithm is 50 times slower than a C sequential implementation of the Dijkstra algorithm, but it is almost twice as fast when compared to the same implementation in Python.

5 Conclusions

We have presented a parallel virtual machine for executing forward-chaining linear logic programs, with particular focus on parallel scheduling on multicores, rule execution and database organization for fast insertion, lookup, and deletion or linear facts. Our preliminary results show that the VM is able to scale the execution of some programs when run with up to 16 threads. Although this is still a work in progress, the VM fairs relatively well against other programming languages. Moreover, since LM programs are concurrent by default, we can easily get better performance from the start by executing them with multiple threads.

In the future, we want to improve our work stealing algorithm so that each thread steals nodes that are a better fit to the set of nodes owned by the thread (e.g. neighboring nodes). We also intend to take advantage of linear logic to perform whole-program optimizations, including computing program invariants and loop detection in rules.

Acknowledgments

This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme; by FCT (Portuguese Foundation for Science and Technology) through the Carnegie Mellon Portugal Program and within projects SIBILA (NORTE-07-0124-FEDER-000059) and PEst (FCOMP-01-0124-FEDER-037281); and by the Qatar National Research Fund under grant NPRP 09-667-1-100. Flavio Cruz is funded by the FCT grant SFRH / BD / 51566 / 2011.

References

1. Ashley-Rollman, M.P., Lee, P., Goldstein, S.C., Pillai, P., Campbell, J.D.: A language for large ensembles of independently executing nodes. In: International Conference on Logic Programming (ICLP) (2009) 1
2. Ashley-Rollman, M.P., Rosa, M.D., Srinivasa, S.S., Pillai, P., Goldstein, S.C., Campbell, J.D.: Declarative programming for modular robots. In: Workshop on Self-Reconfigurable Robots/Systems and Applications at IROS 2007 (2007) 1
3. Betz, H., Frühwirth, T.: A linear-logic semantics for constraint handling rules. In: Principles and Practice of Constraint Programming - CP 2005, Lecture Notes in Computer Science, vol. 3709, pp. 137–151 (2005) 1
4. Colmerauer, A., Roussel, P.: The birth of prolog. In: The Second ACM SIGPLAN Conference on History of Programming Languages. pp. 37–52. New York, NY, USA (1993) 1
5. Costa, V.S., Damas, L., Rocha, R.: The yap prolog system. CoRR abs/1102.3896 (2011) 4
6. Girard, J.Y.: Linear logic. Theoretical Computer Science 50(1), 1–102 (1987) 1
7. Gonzalez, J., Low, Y., Guestrin, C.: Residual splash for optimally parallelizing belief propagation. In: Artificial Intelligence and Statistics (AISTATS) (2009) 1
8. Gupta, G., Pontelli, E., Ali, K.A.M., Carlsson, M., Hermenegildo, M.V.: Parallel execution of prolog programs: A survey. ACM Transactions on Programming Languages and Systems (TOPLAS) 23(4), 472–602 (2001) 1
9. Holzbaur, C., de la Banda, M.J.G., Stuckey, P.J., Duck, G.J.: Optimizing compilation of constraint handling rules in hal. CoRR cs.PL/0408025 (2004) 1
10. Lam, E.S.L., Sulzmann, M.: Concurrent goal-based execution of constraint handling rules. CoRR abs/1006.3039 (2010) 1
11. Liu, M.: Extending datalog with declarative updates. In: International Conference on Database and Expert System Applications (DEXA). vol. 1873, pp. 752–763 (1998) 1
12. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M.: Declarative networking: Language, execution and optimization. In: International Conference on Management of Data (SIGMOD). pp. 97–108 (2006) 1, 3
13. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Graphlab: A new framework for parallel machine learning. In: Conference on Uncertainty in Artificial Intelligence (UAI). pp. 340–349 (2010) 4
14. Ramakrishnan, R., Ullman, J.D.: A survey of research on deductive database systems. Journal of Logic Programming 23, 125–149 (1993) 1
15. van Rossum, G.: Python reference manual. Report CS-R9525, Centrum voor Wiskunde en Informatica, Amsterdam, the Netherlands (Apr 1995), http://www.python.org/doc/ref/ref-1.html 4

A Portable Prolog Predicate for Printing Rational Terms

Theofrastos Mantadelis and Ricardo Rocha

CRACS & INESC TEC, Faculty of Sciences, University of Porto
Rua do Campo Alegre, 1021/1055, 4169-007 Porto, Portugal
{theo.mantadelis,ricroc}@dcc.fc.up.pt

Abstract. Rational terms or rational trees are terms with one or more infinite sub-terms but with a finite representation. Rational terms appeared as a side effect of omitting the *occurs check* in the unification of terms, but their support across Prolog systems varies and often fails to provide the expected functionality. A common problem is the lack of support for printing query bindings with rational terms. In this paper, we present a survey discussing the support of rational terms among different Prolog systems and we propose the integration of a Prolog predicate, that works in several existing Prolog systems, in order to overcome the technical problem of printing rational terms. Our rational term printing predicate could be easily adapted to work for the top query printouts, for user printing and for debugging purposes.

Keywords: Rational Terms, Implementation, Portability.

1 Introduction

From as early as [3, 8], Prolog implementers have chosen to omit the *occurs check* in unification. This has resulted in generating cyclic terms known as *rational terms* or *rational trees*. Rational terms are infinite terms that can be finitely represented, i.e., they can include any finite sub-term but have at least one infinite sub-term. A simple example is `L=[1|L]`, where the variable `L` is instantiated to an infinite list of ones. Prolog implementers started omitting the occurs check in order to reduce the unification complexity from $O(Size_{Term1} + Size_{Term2})$ to $O(min(Size_{Term1}, Size_{Term2}))$.

While the introduction of cyclic terms in Prolog was a side effect of omitting the occurs check, soon after applications for cyclic terms emerged in fields such as definite clause grammars [3, 5], constraint programming [10, 2], coinduction [6, 1, 11, 12] or infinite automata [7]. But support for rational terms across Prolog systems varies and often fails to provide the functionality required by most applications. A common problem is the lack of support for printing query bindings with rational terms [11]. Furthermore, several Prolog features are not designed for compatibility with rational terms and can make programming using rational terms challenging and cumbersome.

In this paper, we address the problem of printing rational terms for a large number of Prolog systems. We thus propose a compliant with ISO Prolog predicate that can be used in several Prolog systems in order to print rational terms. The predicate functions properly in the Ciao, SICStus, SWI, XSB and YAP Prolog systems. The predicate was also tested with the BProlog, ECLiPSe, GNU Prolog, Jekejeke and Strawberry Prolog systems but for different reasons it failed to work (details about the Prolog versions tested are presented next).

The remainder of the paper is organized as follows. First, we discuss how rational terms are supported across a set of well-known Prolog systems. Next, we present our compliant with ISO Prolog predicate and discuss how it can be used to improve the printing of rational terms in several Prolog systems. We end by outlining some conclusions.

2 Rational Term Support in Prolog Systems

We tested several Prolog systems to figure out their available support for rational terms. Table 1 presents in brief our results. Initially, we performed ten different tests that we consider to be the very minimal required support for rational terms. First, we tested the ability of Prolog systems to create rational terms via the `=/2` operator (unification without occurs check). Second, and most critical test, was for the systems to be able to perform unification among two rational terms. Third, we checked whether the Prolog systems can infer the equality of two rational terms by using `==/2` operator. Our fourth test was to see whether a Prolog system can de-construct/construct rational terms through the `=../2` operator, we also investigated whether the Prolog system supports any form of build-in printing predicates for rational terms. The results of the above five tests are presented in Table 1(a).

Furthermore, we checked the support of the `acyclic_term/1` ISO predicate [4], we tested whether `assertz/1` supports asserting rational terms, checked if the `copy_term/2` and `ground/1` predicates work with rational terms and finally, we checked `recordz/3` and `recorded/3` functions with rational terms as an alternative for `assert/1`. The results of these tests appear in Table 1(b).

Finally, we performed a few more compatibility tests as we present in Table 1(c). We want to point out that the results of this table are expected and are sub covered by the test for `==/2` operator. We have the strong conviction that the same reason that forbids the `==/2` operator to function with rational terms in some Prolog systems is the same reason for the failure of the comparison support tests.

Currently, only three Prolog systems appear to be suitable for programming and handling rational terms, namely SICStus, SWI and YAP. The rest of the systems do not provide enough support for rational terms, which makes programming with rational terms in such systems challenging and cumbersome, if

[1] At the time of the publication the current stable version of Yap presented a problem with `recorded/3`, but the development version (6.3.4) already had the problem solved.

Prolog System	**Create** `=/2`	**Compare** `=/2`	**Compare** `==/2`	**De-compose/ compose** `=../2`	**Build-in Printing**
BProlog (8.1)	✓	✗	✗	✓	✗
Ciao (1.14.2)	✓	✓	✓	✓	✗
toplevel query	✓	✓	✗	✗	✗
ECLiPSe (6.1)	✓	✗	✗	✓	✓
GNU (1.4.4)	✓	✗	✗	✓	✗
Jekejeke (1.0.1)	✓	✗	✗	✓	✗
SICStus (4.2.3)	✓	✓	✓	✓	✓
Strawberry (1.6)	✗	✗	✗	✗	✗
SWI (6.4.1)	✓	✓	✓	✓	✓
XSB (3.4.0)	✓	✓	✗	✓	✗
YAP (6.2.3)	✓	✓	✓	✓	✓

(a) Operator Support

Prolog System	`acyclic_term/1`	`assert/1`	`copy_term/2`	`ground/1`	`recordz/3`
BProlog (8.1)	✓	✗	✗	✗	✗
Ciao (1.14.12)	✓	✗	✗	✗	✗
toplevel query	✓	✗	✗	✗	✗
ECLiPSe (6.1)	✓	✓	✗	✗	✓
GNU (1.4.4)	✓	✗	✗	✗	✗
Jekejeke (1.0.1)	✗	✗	✗	✗	✗
SICStus (4.2.3)	✓	✓	✓	✓	✓
Strawberry (1.6)	✗	✗	✗	✗	✗
SWI (6.4.1)	✓	✗	✓	✓	✓
XSB (3.4.0)	✓	✗	✗	✗	✗
YAP (6.2.3)	✓	✗	✓	✓	✓[1]

(b) Predicate Support

Prolog System	**Compare** `@>/2`	**Compare** `@</2`	**Compare** `@>=/2`	**Compare** `@=</2`	`compare/3`
BProlog (8.1)	✗	✗	✗	✗	✗
Ciao (1.14.2)	✓	✓	✓	✓	✓
toplevel query	✗	✗	✗	✗	✗
ECLiPSe (6.1)	✗	✗	✗	✗	✗
GNU (1.4.4)	✗	✗	✗	✗	✗
Jekejeke (1.0.1)	✗	✗	✗	✗	✗
SICStus (4.2.3)	✓	✓	✓	✓	✓
Strawberry (1.6)	✗	✗	✗	✗	✗
SWI (6.4.1)	✓	✓	✓	✓	✓
XSB (3.4.0)	✗	✗	✗	✗	✗
YAP (6.2.3)	✓	✓	✓	✓	✓

(c) Comparison Operator Support

Table 1. Rational term support by Prolog systems

not impossible. All Prolog systems we tested appear to support the creation through unification of rational terms. For Jekejeke and Strawberry Prolog, we where not able to verify the correctness of the created rational term but the system appeared to accept the instruction. SICStus, SWI and YAP Prolog systems also provide built-in predicate implementations capable of handling rational terms without falling into infinite computations making them the most suitable systems to work with rational terms.

For printing purposes, Table 1 shows us that only a few Prolog systems are able to print rational terms without problems. The best printing is offered by SWI as illustrated on the following examples:

```
?- A = [1|A].
A = [1|A].

?- B = [2|B], A = [1|B].
B = [2|B],
A = [1|B].

?- A = [1|B], B = [2|B].
A = [1|_S1], % where
    _S1 = [2|_S1],
B = [2|_S1].
```

YAP offers an alternative printing which is ambiguous:

```
?- A = [1|A].
A = [1|**].

?- B = [2|B], A = [1|B].
A = [1,2|**],
B = [2|**].

?- A = [1|B], B = [2|B].
A = [1,2|**],
B = [2|**].
```

ECLiPSe and SICStus print rational terms in the following way:

```
?- A = [1|A].
A = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...]

?- B = [2|B], A = [1|B].
B = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
A = [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]

?- A = [1|B], B = [2|B].
A = [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
B = [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, ...]
```

The printed . . . from ECLiPSe and SICStus is a result of printing a higher depth term than what the system permits. Both ECLiPSe and SICStus have a depth limit option for their printing which terminates printing resulting to the partially printed rational terms. Disabling the depth limit, traps those systems in infinite cycles[2].

SICStus and SWI also provides the option `cycles(true)` for `write_term/2` in order to print terms using the finite `@/2` notation. This option returns similar printing output with SWI as the following examples illustrate:

```
?- _A = [1|_A], write_term(_A, [cycles(true)]).
@(_906,[_906=[1|_906]])

?- _A = [1|_B], _B = [2|_B], write_term(_A, [cycles(true)]).
@([1|_1055],[_1055=[2|_1055]])

?- _B = [2|_B], _A = [1|_B], write_term(_A, [cycles(true)]).
@([1|_1055],[_1055=[2|_1055]])
```

One can use this option in SICStus toplevel query printing, by setting appropriatly the Prolog flag `toplevel_print_options`.

GNU Prolog, identifies the term as a rational term and instead prints a message:

```
?- A = [1|A].
cannot display cyclic term for A
```

The rest of the systems get trapped in infinite calculation when printing rational terms. Specifically in the case of Ciao Prolog, we want to point out that the toplevel queries automatically print out unnamed variables making any query we tried to fall in infinite calculation. For that reason the Ciao toplevel is completely unsuitable for rational terms. On the other hand Ciao can run programs with a rather good support of rational terms making it the fourth in the row system to support rational terms.

3 Printing Rational Terms

The predicate `canonical_term/3` presented next at Algorithm 1 was originally designed to transform a rational term to its canonical form [9]. Here, we extended it in order to be able to compute a suitable to print term as its third argument. The predicate does not follow the optimal printing for rational terms but that was not our goal. We present a solution that can with minimal effort be used by several Prolog systems to print rational terms and for that we use the minimum amount of needed facilities.

Before explaining the `canonical_term/3` predicate, let's see some examples by using `canonical_term/3` with the XSB system:

```
?- _A = [a|_A], canonical_term(_A, _, Print).
Print = [a|cycle_at_depth(0)]
```

[2] We where unable to disable the depth limit for ECLiPSe toplevel query printing, but we could do it for non toplevel queries.

```
?- _A = [a|_B], _B = [b|_B], canonical_term(_A, _, Print).
Print = [a,b|cycle_at_depth(1)]

?- _A = [a|_B], _B = [b|_B], _F = f(foo, _A, _B, _F),
   canonical_term(_F, _, Print).
Print = f(foo, [a,b|cycle_at_depth(2)], [b|cycle_at_depth(1)],
        cycle_at_depth(0))
```

Notice that our rational term printing is similar with YAP's printing but instead of printing an ambiguous `**`, we print a special term `cycle_at_depth/1` that indicates at which tree depth of the specific tree branch the cyclic sub-term points at. Figure 1, illustrates the term `f(foo, [a,b|cycle_at_depth(2)], [b|cycle_at_depth(1)], cycle_at_depth(0))` using a tree notation. For illustrative purposes, we replaced `cycle_at_depth/1` with `'**'/1` and we use numbered superscripts to mark the respective tree node that each cyclic sub-term points at.

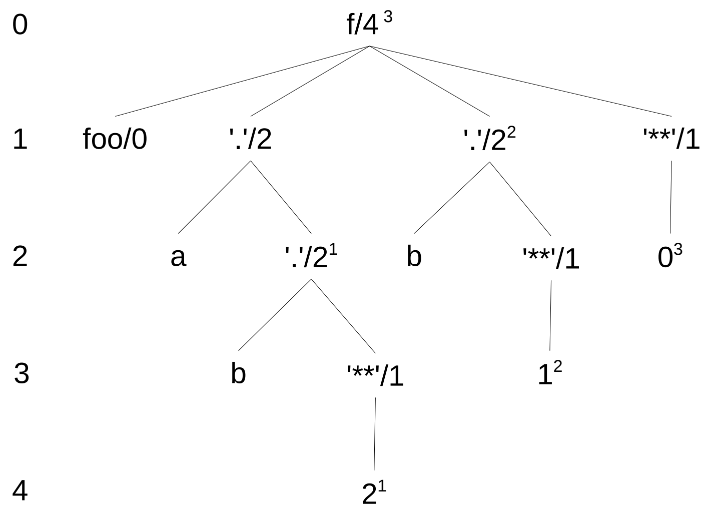


Fig. 1. Rational term: `f(foo, [a,b|cycle_at_depth(2)], [b|cycle_at_depth(1)], cycle_at_depth(0))` in tree notation

While our algorithm is not ambiguous when printing a rational term, it can become ambiguous if the term to be printed also contains `cycle_at_depth/1` terms and the reader of the printed term might falsely think that a cycle exists.

The idea behind the original algorithm as presented at Algorithm 1 is to first fragment the term to its cyclic sub-terms, continue by reconstructing each cyclic sub-term (now acyclic) and, finally, reintroduce the cycle to the reconstructed sub-terms. To reconstruct each cyclic sub-term as acyclic, the algorithm copies the unique parts of the term and introduces an unbound variable instead of the cyclic references. Then, the algorithm binds the unbound variable to the reconstructed sub-term, recreating the cycle.

Take for example the rational term `L=[1,2,1,2|L]`. Term `L` is being fragmented in the following sub-terms: `L0=[1|L1]`, `L1=[2|L3]` and `L3=[1,2|L0]`. We do not need to fragment the term `L3` as, at that point, our algorithm detects a cycle and replaces term `L3` with an unbound variable `OpenEnd`. Thus we get the following sub-terms: `L0=[1|L1]` and `L1=[2|OpenEnd]`. Binding `OpenEnd=L0` results to the canonical rational term `L0=[1,2|L0]`. One might notice that instead of recreating the cycles, if we bind the `OpenEnd` variables with the special term `cycle_at_depth/1` we get the desirable printout. Furthermore, we keep a counter for each decomposition we do in order to keep track of the tree depth of the term.

The bulk of the algorithm is at the fourth clause of `decompose_cyclic_term/7`. At that part we have detected a cyclic sub-term that we have to treat recursively. In particular, lines 31–37 implement an important step. Returning to our example when the cycle is detected, the algorithm returns the unbound variable to each fragmented sub-term. First, the sub-term `L1=[2|OpenEnd]` appears and the algorithm needs to resolve whether it must unify `OpenEnd` with `L1` or whether `OpenEnd` must be unified with a parent sub-term. In order to verify that, lines 31–37 of the algorithm unify the sub-term with the unbound variable and after attempt to unify the created rational term with the original rational term. For our example the algorithm generates `L1=[2|L1]` and attempt to unify with `L=[1,2,1,2|L]`, as the unification fails the algorithm propagates the unbound variable to be unified with the parent sub-term `L0=[1|L1]`.

The fifth clause of `decompose_cyclic_term/7` is the location where a cycle is actually found. At that point we can drop the original cyclic sub-term and place an unbound variable within the newly constructed term. The third clause of `decompose_cyclic_term/7` could be omitted; it operates as a shortcut for simplifying rational terms of the form `F=f(a,f(a,F,b),b)`. The rest of the algorithm is pretty much straightforward, the first clause of `decompose_cyclic_term/7` is the termination condition and the second clause copies the non-rational parts of the term to the new term.

Our algorithm ensures termination by reaching an empty list on the second clause of `decompose_cyclic_term/7`. This happens as at each iteration of the algorithm the second argument list will be reduced by one element. Cyclic elements are detected and removed and while the list might contain cyclic elements it is not cyclic as it is the decomposed list derived by the `=../2` operator that constructs the originally cyclic term. Finally, the call of `in_stack/2` at line 24 ensures that a cyclic term is not been processed more than once.

Complexity wise, our algorithm behaves linearly to the size of the term in all cases but one. Terms of the form `L = [1,2,3,...|L]` cause the algorithm to have a quadratic complexity ($O(N^2)$). The cause of the worst case complexity is the fourth clause of `decompose_cyclic_term/7`. We are currently considering an

Input: a rational term `Term`
Output: a rational term `Canonical` in canonical representation and `Print` an acyclic term that can be used for printing.

```
canonical_term(Term, Canonical, Print) :-
  Term =.. InList,
  decompose_cyclic_term(Term, InList, OutList, OpenEnd, [Term],
                                                PrintList-Cycle_mark, 0),
  Canonical =.. OutList,
  Canonical = OpenEnd,
  Print =.. PrintList,
  Cycle_mark = cycle_at_depth(0).

decompose_cyclic_term(_CyclicTerm, [], [], _OpenEnd, _Stack, []-_, _).
decompose_cyclic_term(CyclicTerm, [Term|Tail], [Term|NewTail], OpenEnd,
                     Stack, [Term|NewPrintTail]-Cycle_mark, DepthCount) :-
  acyclic_term(Term), !,
  decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack,
                                   NewPrintTail-Cycle_mark, DepthCount).
decompose_cyclic_term(CyclicTerm, [Term|Tail], [OpenEnd|NewTail], OpenEnd,
               Stack, [Cycle_mark|NewPrintTail]-Cycle_mark, DepthCount) :-
  CyclicTerm == Term, !,
  decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack,
                                    NewPrintTail-Cycle_mark, DepthCount).

decompose_cyclic_term(CyclicTerm, [Term|Tail], [Canonical|NewTail],
            OpenEnd, Stack, [Print|NewPrintTail]-Cycle_mark, DepthCount) :-
  \+ instack(Term, Stack), !,
  Term =.. InList,
  NewDepthCount is DepthCount + 1,
  decompose_cyclic_term(Term, InList, OutList, OpenEnd2, [Term|Stack],
                                  PrintList-Cycle_mark_2, NewDepthCount),
  Canonical =.. OutList,
  Print =.. PrintList,
  ( Canonical = OpenEnd2,
    Canonical == Term,
    Cycle_mark_2 = cycle_at_depth(NewDepthCount),
    !
  ; OpenEnd2 = OpenEnd,
    Cycle_mark_2 = Cycle_mark
  ),
  decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack,
                                    NewPrintTail-Cycle_mark, DepthCount).

decompose_cyclic_term(CyclicTerm, [_Term|Tail], [OpenEnd|NewTail], OpenEnd,
                Stack, [Cycle_mark|NewPrintTail]-Cycle_mark, DepthCount) :-
  decompose_cyclic_term(CyclicTerm, Tail, NewTail, OpenEnd, Stack,
                                    NewPrintTail-Cycle_mark, DepthCount).

instack(E, [H|_T]) :- E == H, !.
instack(E, [_H|T]) :- instack(E, T).
```

Alg. 1: Predicate `canonical_term/3`

improvement for this, one improvement would be to use a sorted binary tree instead of a list to store and recall the seen cyclic subterms. This improvement would improve the complexity to $(O(N \cdot log(N)))$ but would increase the required build-in support from the Prolog System.

As our target is to print out rational terms at different systems, we had to do a few modifications in order for the predicate to work in other systems. For SWI and YAP, the predicate `canonical_term/3` works as is. For Ciao and SICStus, we only needed to import the appropriate library that contains `acyclic_term/1` and, for XSB, we needed to bypass the lack of support for rational terms of the `==/2` operator by introducing the `compare_rational_terms/2` predicate and replacing the `==/2` operator at lines 1, 21 and 35.

```
% Needed in Ciao to import acyclic_term/1
:- use_module(library(cyclic_terms)).

% Needed in SICStus to import acyclic_term/1
:- use_module(library(terms)).

% Needed in XSB in order to replace ==/2 operator
compare_rational_terms(A, B) :-
  acyclic_term(A),
  acyclic_term(B), !,
  A == B.
compare_rational_terms(A, B) :-
  \+ var(A), \+ var(B),
  \+ acyclic_term(A),
  \+ acyclic_term(B),
  A = B.
```

We want to point out that `compare_rational_terms/2` predicate is not the same with `==/2` predicate and comparisons among terms like: `A = [1,_,2|A]`, `B = [1,a,2|B]` would give wrong results. But for our purpose, where the terms being compared are sub-terms, this problem does not appear as it compares the sub-terms after decomposing them to their smallest units.

4 Towards Optimal Printing of Rational Terms

4.1 SWI

As we earlier pointed out, SWI is the closest to the desirable printing system. However, SWI printing of rational terms suffers from two problems. First, SWI does not print the canonical form of rational terms, as the following example illustrates:

```
?-  A = [1,2|B], B = [1,2,1,2|B].
A = B, B = [1, 2, 1, 2|B].
```

This could be easily corrected by using our predicate to process the rational term before printing it.

The second problem is that SWI can insert auxiliary terms that are not always necessary. For example:

```
?- A = [1|B], B = [2|B].
A = [1|_S1], % where
    _S1 = [2|_S1],
B = [2|_S1].
```

This problem could be addressed with SWI's built-in `term_factorized/3` predicate. Using the same example:

```
?- A = [1|B], B = [2|B], term_factorized((A, B), Y, S).
A = [1|_S1], % where
    _S1 = [2|_S1],
B = [2|_S1],
Y = ([1|_G34], _G34),
S = [_G34=[2|_G34]].
```

Notice that `Y` and `S` contain the desirable printouts. We also want to point out that `term_factorized/3` appears to compute also the canonical form of rational terms which would solve both printing issues. Using again the initial example:

```
?- A = [1,2|B], B = [1,2,1,2|B], term_factorized((A, B), Y, S).
A = B, B = [1, 2, 1, 2|B],
Y = (_G46, _G46),
S = [_G46=[1, 2|_G46]].
```

4.2 YAP

Similarly with SWI, YAP's development version implements a `term_factorized/3` predicate. Future printing of the Yap Prolog system should take advantage of the predicate in order to printout rational terms better.

```
?- A = [1|B], B = [2|B], term_factorized((A, B), Y, S).
A = [1,2|**],
B = [2|**],
S = [_A=[2|_A]],
Y = ([1|_A],_A).

?- A = [1,2|B], B = [1,2,1,2|B], term_factorized((A, B), Y, S).
A = B = [1,2,1,2,1,2,1,2,1,2|**],
S = [_A=[1,2,1,2|_A]],
Y = ([1,2|_A],_A).
```

Notice that YAP's current `term_factorized/3` predicate does not work exactly like SWI's and, currently, it still does not ensure canonical form for rational terms.

4.3 SICStus

SICStus should use the build-in `write_term/2` predicate in order to improve the printing of rational terms. The `write_term/2` predicate appears to both compute the canonical form of the rational term and to generate the minimal needed sub-terms for printing, as the following examples illustrate:

```
?- A = [1|B], B = [2|B], write_term((A, B), [cycles(true)]).
@(([1|_1092],_1092),[_1092=[2|_1092]])
A = [1,2,2,2,2,2,2,2,2,2,2|...],
B = [2,2,2,2,2,2,2,2,2,2|...] ?

?- A = [1,2|B], B = [1,2,1,2|B], write_term((A, B), [cycles(true)]).
@((_1171,_1171),[_1171=[1,2|_1171]])
A = [1,2,1,2,1,2,1,2,1,2|...],
B = [1,2,1,2,1,2,1,2,1,2|...] ?
```

4.4 Ciao

Ciao provides a rather good support of rational terms in comparison with other Prolog systems. However, it has the most problematic toplevel query interface. All queries that would contain rational terms are trapped on an infinite computation and using unnamed variables does not override the problem. The authors believe that this problem is directly related with the printing of rational terms and if Ciao would use a different printing strategy the problem would be solved. Our proposed solution would be an easy way for Ciao to support printing for rational terms. Similarly, printing should be improved also for debugging purposes.

4.5 XSB

XSB imposes several challenges to the programmer to use rational terms. Further than being trapped on infinite computations when trying to print rational terms, it also does not support comparison operators like `==/2`. Regardless of the limitations of the system, we believe that XSB would significantly benefit by using a better printing strategy for rational terms. Similarly, printing should be improved also for debugging purposes.

4.6 Other Prolog Systems

The other Prolog systems that we tried are further away from achieving even the basic rational term support. Even if we were able to print simple rational terms in BProlog, ECLiPSe and GNU Prolog, the lack of support for unification among two rational terms makes it impossible to work with. These systems still treat rational terms as a known bug of unification rather than a usable feature. GNU Prolog in that respect behaved rather well as it identifies rational terms and gives warning messages both when compiling and at runtime. Also, ECLiPSe is not caught in infinite computation and is able to print a representation of rational terms even if the programmer is unable to work with them.

4.7 About `term_factorized/3`

The predicate `term_factorized(+Term, -Skeleton, -Substitution)` is true when: (i) `Skeleton` is the *skeleton* term of `Term`, this means that all subterms of `Term` that appear multiple times are replaced by variables; and (ii) `Substitution` is a list of terms in the form of `VAR = SubTerm` that provides the necessary substitutions to recreate `Term` from `Skeleton` by unifying `VAR = SubTerm`.

The `term_factorized/3` predicate in SWI Prolog is implemented using the red-black tree Prolog library by Vítor Santos Costa. The red-black tree library originally appears in Yap Prolog and is an easy to port in other Prolog systems library. Using `term_factorized/3` for printing rational terms would increase the operators that require to support rational terms to at least: `==/2`, `@</2`, `@>/2`, `compare/3`. For these reasons migrating `term_factorized/3` would be more work than using our `canonical_term/3` predicate.

5 Conclusions

Rational terms, while not being very popular, they have found applications in fields such as definite clause grammars, constraint programming, coinduction or infinite automata. With this paper, we try to motivate Prolog developers to support rational terms better and to provide a minimal support for researchers and programmers to work with. We have presented a short survey of the existing support for rational terms in several well-know Prolog systems and we proposed a printing predicate that Prolog systems could use in order to improve their printing of rational terms.

In particular, Ciao and XSB Prolog systems would benefit the most from our predicate. As we explained, Ciao and XSB fall on infinite computations when they need to print a rational term. Our predicate gives them an easy to integrate solution that will allow printing of rational terms and debugging of code that contains rational terms. Our `canonical_term/3` predicate could also be used in YAP to improve the current ambiguous printing format of rational terms and to present rational terms in their canonical form. SWI could also use our predicate in order to benefit by printing rational terms in canonical form. Still, we believe that both YAP and SWI should do an integration of their `term_factorized/3` predicate with their printing of rational terms. Finally, SICStus can use our predicate to provide an alternative printing, but integrating `write_term/2` predicate on the default printing of terms would be more beneficial.

Acknowledgments

The authors want to thank Paulo Moura and Vítor Santos Costa for their suggestions and technical support. We also want to thank the anonymous reviewers for their constructive and supportive comments that helped us improve this paper. This work is partially funded by the ERDF (European Regional Development Fund) through the COMPETE Programme and by FCT (Portuguese Foundation for Science and Technology) within project SIBILA (NORTE-07-0124-FEDER-000059) and PEst (FCOMP-01-0124-FEDER-037281).

References

1. Ancona, D.: Regular Corecursion in Prolog. Computer Languages, Systems & Structures 39(4), 142–162 (2013), Special issue on the Programming Languages track at the 27th ACM Symposium on Applied Computing
2. Bagnara, R., Gori, R., Hill, P.M., Zaffanella, E.: Finite-Tree Analysis for Constraint Logic-Based Languages: The Complete Unabridged Version (2001)
3. Colmerauer, A.: Prolog and Infinite Trees. In: Clark, K.L., Tärnlund, S.A. (eds.) Logic Programming, pp. 231–251. Academic Press (1982)
4. Committee ISO/IEC JTC 1/SC 22: ISO/IEC 13211-1:1995/Cor.2:2012(en): Information technology — Programming languages — Prolog — Part 1: General core TECHNICAL CORRIGENDUM 2 (2012)
5. Giannesini, F., Cohen, J.: Parser generation and grammar manipulation using prolog's infinite trees. The Journal of Logic Programming 1(3), 253 – 265 (1984)
6. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive logic programming and its applications. In: Logic Programming, LNCS, vol. 4670, pp. 27–44. Springer-Verlag (2007)
7. J. E. Hopcroft, J.E., Karp, R.M.: A linear algorithm for testing equivalence of finite automata. Tech. rep., Cornell University (1971)
8. Jaffar, J., Stuckey, P.J.: Semantics of Infinite Tree Logic Programming. Theoretical Computer Science 46(0), 141–158 (1986)
9. Mantadelis, T., Rocha, R., Moura, P.: Tabling, Rational Terms, and Coinduction Finally Together! Journal of Theory and Practice of Logic Programming, International Conference on Logic Programming, Special Issue (2014 to appear)
10. Meister, M., Frühwirth, T.: Complexity of the CHR rational tree equation solver. In: Constraint Handling Rules. vol. 452, pp. 77–92 (2006)
11. Moura, P.: A Portable and Efficient Implementation of Coinductive Logic Programming. In: International Symposium on Practical Aspects of Declarative Languages, LNCS, vol. 7752, pp. 77–92. Springer-Verlag (2013)
12. Simon, L., Bansal, A., Mallya, A., Gupta, G.: Co-Logic Programming: Extending Logic Programming with Coinduction. In: Automata, Languages and Programming, Lecture Notes in Computer Science, vol. 4596, pp. 472–483. Springer-Verlag (2007)

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2011-01 *	Fachgruppe Informatik: Jahresbericht 2011
2011-02	Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
2011-03	Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems
2011-04	Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars
2011-06	Johannes Lotz, Klaus Leppkes, and Uwe Naumann: dco/c++ - Derivative Code by Overloading in C++
2011-07	Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV
2011-08	Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
2011-09	Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
2011-10	Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-11	Nils Jansen, Erika brahám, Jens Katelaan, Ralf Wimmer, Joost-Pieter Katoen, Bernd Becker: Hierarchical Counterexamples for Discrete-Time Markov Chains
2011-12	Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13	Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14	Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16	Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-17	Carsten Fuhs: SAT Encodings: From Constraint-Based Termination Analysis to Circuit Synthesis
2011-18	Kamal Barakat: Introducing Timers to pi-Calculus
2011-19	Marc Brockschmidt, Thomas Ströder, Carsten Otto, Jürgen Giesl: Automated Detection of Non-Termination and NullPointerExceptions for Java Bytecode

2011-24	Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations
2011-25	Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26	Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01	Fachgruppe Informatik: Annual Report 2012
2012-02	Thomas Heer: Controlling Development Processes
2012-03	Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04	Marcus Gelderie: Strategy Machines and their Complexity
2012-05	Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting
2012-06	Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data
2012-07	André Egners, Björn Marschollek, and Ulrike Meyer: Hackers in Your Pocket: A Survey of Smartphone Security Across Platforms
2012-08	Hongfei Fu: Computing Game Metrics on Markov Decision Processes
2012-09	Dennis Guck, Tingting Han, Joost-Pieter Katoen, and Martin R. Neuhäußer: Quantitative Timed Analysis of Interactive Markov Chains
2012-10	Uwe Naumann and Johannes Lotz: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations
2012-12	Jürgen Giesl, Thomas Ströder, Peter Schneider-Kamp, Fabian Emmes, and Carsten Fuhs: Symbolic Evaluation Graphs and Term Rewriting — A General Methodology for Analyzing Logic Programs
2012-15	Uwe Naumann, Johannes Lotz, Klaus Leppkes, and Markus Towara: Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Solvers for Systems of Nonlinear Equations
2012-16	Georg Neugebauer and Ulrike Meyer: SMC-MuSe: A Framework for Secure Multi-Party Computation on MultiSets
2012-17	Viet Yen Nguyen: Trustworthy Spacecraft Design Using Formal Methods
2013-01 *	Fachgruppe Informatik: Annual Report 2013
2013-02	Michael Reke: Modellbasierte Entwicklung automobiler Steuerungssysteme in Klein- und mittelständischen Unternehmen
2013-03	Markus Towara and Uwe Naumann: A Discrete Adjoint Model for OpenFOAM
2013-04	Max Sagebaum, Nicolas R. Gauger, Uwe Naumann, Johannes Lotz, and Klaus Leppkes: Algorithmic Differentiation of a Complex C++ Code with Underlying Libraries
2013-05	Andreas Rausch and Marc Sihling: Software & Systems Engineering Essentials 2013

2013-06	Marc Brockschmidt, Byron Cook, and Carsten Fuhs: Better termination proving through cooperation
2013-07	André Stollenwerk: Ein modellbasiertes Sicherheitskonzept für die extrakorporale Lungenunterstützung
2013-08	Sebastian Junges, Ulrich Loup, Florian Corzilius and Erika brahám: On Gröbner Bases in the Context of Satisfiability-Modulo-Theories Solving over the Real Numbers
2013-10	Joost-Pieter Katoen, Thomas Noll, Thomas Santen, Dirk Seifert, and Hao Wu: Performance Analysis of Computing Servers using Stochastic Petri Nets and Markov Automata
2013-12	Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl: Alternating Runtime and Size Complexity Analysis of Integer Programs
2013-13	Michael Eggert, Roger Häußling, Martin Henze, Lars Hermerschmidt, René Hummen, Daniel Kerpen, Antonio Navarro Pérez, Bernhard Rumpe, Dirk Thißen, and Klaus Wehrle: SensorCloud: Towards the Interdisciplinary Development of a Trustworthy Platform for Globally Interconnected Sensors and Actuators
2013-14	Jörg Brauer: Automatic Abstraction for Bit-Vectors using Decision Procedures
2013-19	Florian Schmidt, David Orlea, and Klaus Wehrle: Support for error tolerance in the Real-Time Transport Protocol
2013-20	Jacob Palczynski: Time-Continuous Behaviour Comparison Based on Abstract Models
2014-01 *	Fachgruppe Informatik: Annual Report 2014
2014-02	Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software
2014-03	Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide
2014-04	Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06	Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07	Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08	Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs

* These reports are only available as a printed version.
Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.