Proceedings of the 13th International Workshop on Logic Programming Environments

Fred Mesnard
Alexander Serebrenik (Eds.)

Report CW 371, November 2003

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

Proceedings of the 13th International Workshop on Logic Programming Environments

Fred Mesnard
Alexander Serebrenik (Eds.)

Report CW 371, November 2003

Department of Computer Science, K.U.Leuven

Preface

This volume contains papers presented at WLPE 2003, the 13th International Workshop on Logic Programming Environments. The aim of WLPE is to provide an informal meeting for researchers working on tools for development and analysis of logic programming. This year, the emphasis is on the presentation, pragmatics and experiences of such tools.

WLPE 2003 takes place in Tata Institute of Fundamental Research, Mumbai, India on December 8 and is a part of a bigger event, ICLP 2003, the 19th International Conference on Logic Programming, holding in conjunction with ASIAN 2003, the Eighth Asian Computing Science Conference, and FSTTCS 2003, the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science. This workshop continues the series of successful international workshops on logic programming environments held in Ohio, USA (1989), Eilat, Israel (1990), Paris, France (1991), Washington, USA (1992), Vancouver, Canada (1993), Santa Margherita Ligure, Italy (1994), Portland, USA (1995), Leuven, Belgium and Port Jefferson, USA (1997), Las Cruces, USA (1999), Paphos, Cyprus (2001) and Copenhagen, Denmark (2002).

We would like to express our gratitude to the ICLP organisers for hosting the workshop. Special thanks go to R.K.Shyamasundar for taking care of the many organisational matters, in particular, printing these proceedings. Also we would like to thank the program committee members for reviewing and discussing the submissions as well as the authors for submitting their work.

Out of 9 submissions the program committee has selected 5 works for presentation. In addition, Jan Wielemaker (University of Amsterdam, The Netherlands) was invited to present a number of typical problems Prolog users are faced with and illustrate how tools developed in SWI-Prolog may help to find them.

Fred Mesnard
Alexander Serebrenik
Mumbai, December 2003

Organisation

13th Workshop on Logic Programming Environments
WLPE 2003 December 8, 2003, Mumbai, India

Workshop organisers:

Fred Mesnard (Université de La Réunion, France)
Alexander Serebrenik (coordinator, Katholieke Universiteit Leuven, Belgium)

Program committee:

Roberto Bagnara (Università degli studi di Parma, Italy)
Manuel Carro (Universidad Politécnica de Madrid, Spain)
Mireille Ducassé (INSA/IRISA, Rennes, France)
Pat Hill (University of Leeds, U.K.)
Naomi Lindenstrauss (Hebrew University of Jerusalem, Israel)
Jan-Georg Smaus (Universität Freiburg, Germany)
Fausto Spoto (Università di Verona, Italy)
Alexandre Tessier (Université d'Orléans, France)

Reviewers:

Pieter Bekaert
Maurice Bruynooghe
Daniel Cabeza
Pierre Deransart
Gerard Ferrand
José Manuel Gómez
Arnaud Lallouet
Tom Schrijvers
Zoltan Somogyi
Joost Vennekens

Table of Contents

An Overview of the SWI-Prolog Programming Environment 1
Jan Wielemaker

TCLP: A type checker for CLP(X) 17
Emmanuel Coquery

Analyzing and Visualising Prolog programs based on XML representations 31
Dietmar Seipel, Marbod Hopfner, Bernd Heumesser

Demonstration proposal: Debugging constraint problems with portable tools 46
Pierre Deransart, Ludovic Langevine and Mireille Ducasse

Proving Termination One Loop at a Time 48
Michael Codish and Samir Genaim

Hasta-La-Vista: Termination Analyser for Logic Programs 60
Alexander Serebrenik and Danny De Schreye

Constructive combination of crisp and fuzzy logic in a Prolog compiler . . 75
Susana Munoz, Claudio Vaucheret and Sergio Guadarrama

An Overview of the SWI-Prolog Programming Environment

Jan Wielemaker

Social Science Informatics (SWI),
University of Amsterdam,
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands,
jan@swi.psy.uva.nl

Abstract. The Prolog programmer's needs have always been the focus for guiding the development of the SWI-Prolog system. This article accompanies an invited talk about how the SWI-Prolog environment helps the Prolog programmer solve common problems. It describes the central parts of the graphical development environment as well as the command line tools which we see as vital to the success of the system. We hope this comprehensive overview of particularly useful features will both inspire other Prolog developers, and help SWI-Prolog users to make more productive use of the system.

1 Introduction

SWI-Prolog has become a popular Free Software implementation of the Prolog language. Distributed freely through the internet, it is difficult to get a clear picture about its users, how these users use the system and which aspects of the system have contributed most to its popularity. Part of the users claim the programmer's environment described in this article is an important factor.

The majority of the SWI-Prolog users are students using it for their assignments. The community of developers, however, expend effort on large portable Prolog applications where scalability, (user-) interfaces, networking are often important characteristics. Compared to the students, who are mostly short-term novice users, we find many expert software developers in the research and development community.

The material described in this paper is the result of about 18 years experience as a Prolog programmer and developer of the SWI-Prolog system. Many of the described tools are features not unique to SWI-Prolog and can be found in other Prolog implementations or other programming language environments. Experiments are yet to be performed to evaluate the usefulness of features and therefore the opinions presented are strictly based on our own experiences, observations of users, and E-mail reactions.

After describing the SWI-Prolog user community in Sect. 2 we describe some problems Prolog programmers frequently encounter in Sect. 3. In Sect. 4 we describe the command line tools, and in Sect. 5 the graphical tools written in SWI-Prolog's XPCE GUI toolkit [10].

2 User profiles

Students having to complete assignments for a Prolog course have very different needs from professionals developing large systems. They want easy access to common tasks as closely as possible to the conventions they are used to. Scalability of supporting tools is not an important issue as the programs do not require many resources. Visualization of terms and program state can concentrate their contribution to *explanation* and disregard, for example, the issue that most graphical representations scale poorly. The *SWI-Prolog-Editor*[1] shell for MS-Windows by Gerhard Röhner makes SWI-Prolog much more natural to a student who is first of all familar with MS-Windows.

SWI-Prolog comes from the Unix and Emacs tradition and targets the professional programmer who uses it frequently to develop large Prolog-based applications. As many users in this category have their existing habits, and a preferred set of tools to support these, SWI-Prolog avoids presenting a single comprehensive IDE (*Integrated Development Environment*), but instead provides individual components that can be combined and customised at will.

3 Problems

Many problems that apply to programming in Prolog also relate the programming in other languages. Some, however, are Prolog specific. Prolog environments can normally be used interactively and changed dynamically.

3.1 Problem areas

– *Managing sources*
  Besides the normal problems such as locating functions and files, Prolog requires a tool that manages consistency between the sources and running executable during the interactive test-edit cycle. Section 4.1 and Sect. 5.1 describe the SWI-Prolog support to manage sources.

[1] http://www.bildung.hessen.de/abereich/inform/skii/material/swing/indexe.htm

– *Entering and reusing queries*
  Interaction through the Prolog top level is vital for managing the program and testing individual predicates. Command line editing, command completion, do what I mean (DWIM) correction, history, and storing the values of top level variables reduces typing and speed up the development cycle.
– *Program completeness and consistency*
  SWI-Prolog has no tradition in rigid static analysis. It does provide a quick completeness test as described in Sect. 4.6 which runs automatically during the test-edit cycle. A cross-referencer is integrated into the built-in editor (Sect. 5.1) and provides immediate feedback to the programmer about common mistakes while editing a program.
– *Error context*
  If an error occurs, it is extremely important to provide as much context as possible. The SWI-Prolog exception handling differs slightly from the ISO standard to improve such support. See Sect. 4.10.
– *Failure/wrong answer*
  A very common and time consuming problem are programs producing the wrong (unexpected) answer without producing an error. Although research has been carried out to attribute failure and wrong answers to specific procedures [3, 9], none of this is realised in SWI-Prolog.
– *Determinism*
  Although experience and discipline help, controlling determinism in Prolog programs to get all intended solutions quickly is a very common problem. The source-level debugger (Sect. 5.3) displays choicepoints and provides immediate graphical feedback on the effects of the cut, greatly simplifying this task and improving understanding for novices.
– *Performance bottlenecks*
  Being a high level language, the relation between Prolog code and required resources to execute it is not trivial. Profiling tools cannot fix poor overall design, but do provide invaluable insight to programmer. See Sect. 5.4.
– *Porting programs from other systems*
  Porting Prolog programs has been simplified since more Prolog systems have adopted part I of the ISO standard. Different extensions and libraries cause many of the remaining problems. Compiler warnings and static analysis form the most important tools to locate the problem areas quickly. A good debugger providing context on errors together with support for the test-edit cycle improve productivity.

4 Command line Tools

4.1 Supporting the edit cycle

Prolog systems offer the possibility to interactively edit and reload a program even while the program is running. There are two simple but very frequent tasks involved in the edit-reload cycle: finding the proper source, and reloading the modified source files. SWI-Prolog supports these tasks with two predicates:

make
: SWI-Prolog maintains a database of all loaded files with the file last-modified time stamp when it was loaded and —for the sake of modules— the context module(s) from which the file was loaded. The **make/0** predicate checks whether the modification time of any of the loaded files has changed and reload these file into the proper module context. This predicate has proven to be very useful.

edit(*+Specifier***)**
: Find all entities with the given *specifier*. If there are multiple entities related to different source-files ask the user for the desired one and call the user-defined editor on the given location. *All entities* implies (loaded) files, predicates and modules. Both locating named entities and what is required to call the editor on a specific file and line can be hooked to accomodate extensions (e.g. XPCE classes) and different editors. Furthermore, SWI-Prolog maintains file and line-number information for modules and clauses. Below is an example:

```
?- edit(rdf_tree).
Please select item to edit:

  1 class(rdf_tree)             'rdf_tree.pl':27
  2 module(rdf_tree)            'rules.pl':460

Your choice? 2
```

SWI-Prolog's *completion* and *DWIM* described in Sect. 4.4 and Sect. 4.3 improve the usefulness of these primitives.

4.2 Autoloading and auto import

Programmers tend to be better at remembering the names of library predicates than the exact library they belong to. Similar, programmers of

large modular applications often have a set of personal *favourites* and application specific *goodies*. SWI-Prolog supports this style of programming with two mechanisms, both of which require a module system. The SWI-Prolog module system is very close to the Quintus and SICStus Prolog module systems [2].

Auto import tries to import undefined predicates from the module's *import module*. The module `system` contains all built-in predicates, `user` all global predicates and all other modules import from `user` as illustrated in Fig. 1. This setup allows programmers to define or import commonly used predicates into `user` and have them available without further actions from the interactive top level and all modules.

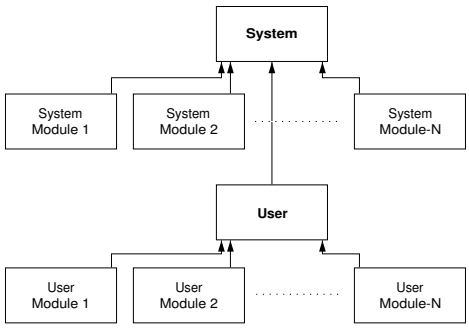


Fig. 1. Modules and their *auto-import* relations

Library *auto loading* avoids the need for explicit **use_module/[1,2]** declarations. Whenever the system encounters an unknown predicate it examines the library index. If the predicate appears in the index the library is loaded using **use_module/2**, only importing the missing predicate.

The combination of auto import, auto loading and a structuring module system has proven to support both *sloppy* programming for rapid prototyping and the use of more maintainable explicit module relations. The predicate **list_autoload/0** as described in Sect. 4.6 supports a smooth transition.

4.3 DWIM: Do What I Mean

DWIM (*Do What I Mean*) is implemented at the top level to quickly fix mistakes and allow for underspecified queries. It corrects the following errors:

- *Simple spelling errors*
  DWIM checks for missing, extra and transposed characters that result from typing errors.
- *Word breaks and order*
  DWIM checks for multi-word identifiers using different conventions (e.g. *fileExists* vs. *file_exists*) as well as different order (e.g. *exists_file* vs. *file_exists*)
- *Arity mismatch*
  Of course such errors cannot be corrected.
- *Wrong module*
  DWIM adds a module specification to predicate references that lack one or replaces a wrong module specification.

DWIM is used in three areas. Queries typed at the top level are checked and if there is a unique correction the system prompts whether to execute the corrected rather than the typed query. Especially adding the module specifier improves interaction from the top level when using modules. If there is no unique correction the system reports the missing predicates and all close candidates. Queries of the development system such as **edit/1** and **spy/1** provide alternative matches one-by-one. Spy/1 and **trace/1** act on the specified predicate in any module if the module is omitted. Finally, if a predicate existence error reaches the top level the DWIM system is activated to report likely candidates.

4.4 Command line editing

Developers spend a lot of time entering commands for the development system and (test-)queries for (parts of) their application under development. SWI-Prolog provides the following features to support this:

- *Using (GNU-)readline*
  Emacs-style editing is supported in the Unix version based on the GNU *readline* library and in Windows using our own code. This facilitates quick and natural command reuse and editing. In addition, *completion* is extended with completion on alphanumerical atoms which allow for fast typing of long predicate identifiers and atom arguments as well as inspect the possible alternative (using Alt-?). The completion algorithm uses the builtin completion of files if no atom matches, which ensures that quoted atoms representing a file path is completed as expected.

– *Command line history*
SWI-Prolog provides a history facility that resembles the Unix csh and bash shells. Especially viewing the list of executed commands is a valuable feature.
– *Top level bindings*
When working at the Prolog top level, bindings returned by previous queries are normally lost while they are often required for further analysis of the current Prolog state or to test further queries. For this reason SWI-Prolog stores the resulting bindings from top level queries, provided they are not too large (default $\leq$ 1000 tokens) in the database under the name of the used variable. Top level query expansion replaces terms of the form $\$Var$ ($ is a prefix operator) into the last recorded binding for this variable. New bindings do to backtracking or new queries overwrite the old value.
This feature is particularly useful to query the state of data stored in related dynamic predicates and deal with handles provided by external stores. Here is a typical example using XPCE that avoids typing or copy/paste of the object reference.

```
?- new(X, picture).

X = @12946012
?- send($X, open).
```

4.5 Compiler

An important aspect of the SWI-Prolog compiler is its performance. Loading the 21 Mb sources of WordNet [7] requires 6.6 seconds from the source and 1.4 seconds from precompiled virtual machine code (Multi-threaded SWI-Prolog 5.2.9, SuSE Linux on dual AMD 1600+ using one thread). Fast compilation is very important during the interactive development of large applications.

SWI-Prolog supports the commonly found set of compiler warnings: syntax errors, singleton variables, predicate redefinition, system predicate redefinition and discontiguous predicates. Messages are processed by the hookable **print_message/2** predicate and where possible associated with a file and line number. The graphics system contains a tool that exploits the message hooks to create a window with error messages and warnings that can be selected to open the associated source location.

4.6 Quick consistency check

The library *check* provides quick tests on the completeness of the loaded program. The predicate **list_undefined/0** searches the internal database for predicate structures that are undefined (i.e. have no clauses and are not defined as dynamic or multifile). Such structures are created by the compiler for a call to a predicate that is not yet defined. In addition the system provides a primitive that returns the predicates referenced from a clause by examining the compiled code. Figure 2 provides partial output running **list_undefined/0** on the *chat 80* [8] program:

```
1 ?- [library(chat)].
%   ...
%  library('chat/chat') compiled into chat 0.18 sec, 493,688 bytes
% library(chat) compiled into chat 0.18 sec, 494,756 bytes

Yes
2 ?- list_undefined.
% Scanning references for 9 possibly undefined predicates
Warning: The predicates below are not defined. If these are defined
Warning: at runtime using assert/1, use :- dynamic Name/Arity.
Warning:
Warning: chat:ditrans/12, which is referenced by
Warning:         5-th clause of chat:verb_kind/6
```

Fig. 2. Using **list_undefined/0** on chat 80 wrapped into the module `chat`. To save space only the first of the 9 reported warnings is included. The processing requires 0.25 sec. on a 733 Mhz PIII.

The **list_autoload/0** predicate lists undefined predicates that can be autoloaded from one of the libraries. It is illustrated in Fig. 3.

```
3 ?- list_autoload.
% Into module chat (library('chat.pl'))
%     display/1          from library(edinburgh)
%     last/2             from library(lists)
%     time/1             from library(statistics)
% Into module user
%     prolog_ide/1       from library(swi_ide)
```

Fig. 3. Using **list_autoload/0** on chat 80

4.7 Help and explain facility

The help facility uses outdated but still effective technology. The LaTeX maintained source is translated to plain text. A generated Prolog index file provides character ranges for predicate descriptions and sections in the manual. Each predicate has, besides the full documentation, a ± 40 character summary description used for *apropos* search as well as to provide a summary string in the editor as illustrated in Fig. 4.

The *explain* facility examines the database to gather all information known about an identifier (atom). Information displayed includes predicates with that name and references to the atoms, compound terms and predicates with the given name. Here is an example:

```
explain(setof).
"setof" is an atom
        Referenced from 1-th clause of chat:decomp/3
system:setof/3 is a built-in meta predicate imported from module
               $bags defined in
        /staff/jan/lib/pl-5.2.9/boot/bags.pl:59
        Summary: ‘‘Find all unique solutions to a goal’’
        Referenced from 6-th clause of chat:satisfy/1
        Referenced from 7-th clause of chat:satisfy/1
        Referenced from 1-th clause of chat:seto/3
```

The graphical front end is described in Sect. 5.5.

4.8 File commands

Almost too trivial to name, but the predicates **ls/0**, **cd/1** and **pwd/0** are used very frequently.

4.9 Debugging from the terminal

SWI-Prolog comes with two tracers, a traditional 4-port debugger [1] to be used from the terminal and a graphical source level debugger which is described in Sect. 5.3. Less frequently seen features of the trace are:

- *Single keystroke operation*
  If the terminal supports it, commands are entered without waiting for RETURN.
- *List choicepoints*
  The tracer can provide a list of active choicepoints, similar to the goal stack, to facilitate choicepoint tuning and debugging.

– *The 'up' command*
  The 'up' command is like the traditional 'skip' command, but skips to the exit or failure of the *parent* goal rather than the current goal. It is very useful to stop tracing the details of failure driven control structures.
– *Search*
  The system can search for a specific port and goal that unifies with an entered term. The command `/f foo(_, bar)` will go into interactive debugging if **foo/2** where the second argument unifies with `bar` reaches the *fail* (f) port.

In addition to interactive debugging two types of non-interactive debugging are provided. Using **trace**(*Predicate, Ports*), the system prints all passes to the indicated ports of *Predicate*.

The library *debug* is a lightweight infrastructure to handle printing debugging messages (logging) and assertions. The library exploits goal-expansion to avoid runtime overhead when compiled with optimisation turned on. Debug messages are associated to a *Topic*, an arbitrary Prolog term used to group debug messages. Normally the *Topic* is an atom denoting some function or module of the application. Using Prolog unification of the active topics and the topic registered with the message provides opportunity for creativity.

debug(*+Topic, +Format, +Arguments***)**
  Prints a message through the system's **print_message/2** message dispatching mechanism if debugging is enabled on *Topic*.

debug/nodebug(*+Topic***)**
  Enable/disable messages for which *Topic* unifies. Note that topics are arbitrary Prolog terms, so **debug**(_) enables all debugging messages.

list_debug_topics
  List all registered topics and their current enable/disable setting. All known topics are collected during compilation using goal-expansion.

assume(*:Goal***)**
  Assume that *Goal* can be proven. Trap the debugger if *Goal* fails. This facility is derived from the C-language assert() macro defined in `<assert.h>`, renamed for obvious reasons. More formal assertion languages are described in [6, 5].

4.10 Exception context

On exception handling, the ISO standard dictates 'undo' back to the state at entry of a **catch/3** before unifying the *ball* with the *catcher*. SWI-

Prolog however uses a different technique. It walks the stack searching for a matching catcher without undoing changes. If it finds a matching **catch/3** call or when reaching a call from foreign code that indicates it is prepared to handle exceptions it performs the required 'undo' and executes the handler. The advantage is that if there is no handler for the exception the entire program state is still intact. The debugger is started immediately and can be used to examine the full context of the exception.[2]

5 Graphical Tools

5.1 Editor

PceEmacs is an Emacs clone written in XPCE/Prolog. It has two features that make it of special interest. It can be programmed in Prolog and therefore has transparent access to the environment of the application being developed, and the editor's buffer can be opened as a Prolog I/O stream. Based on these features, the level of support for Prolog development is far beyond what can be achieved in a stand-alone editor. Whenever the user pauses for two seconds the system performs a full cross-reference of the editing buffer, categorising and colouring predicates, goals and general Prolog terms. Predicates are categorised as *exported*, *called* and *not called*. Goals are categorised as *builtin*, *imported*, *auto-imported*, *locally defined*, *dynamic*, *(direct-)recursive* and *undefined*. Goals have a menu that allows jumps to the source, documentation (builtin), and listing of clauses (dynamic). Singleton variables are highlighted. If the cursor appears inside a variable all other occurrences of this variable in the clause are underlined. Figure 4 shows a typical screenshot.

5.2 Prolog Navigator

The Prolog *Navigator* provides a hierarchical overview of a project directory and its Prolog files. Prolog files are categorised as one of *loaded* or *not loaded* and are expanded to the predicates defined in them. The defined predicates are categorised as one of *exported*, *normal*, *fact* and *unreferenced*. Expanding predicates expands the *call tree*. The Navigator menus provide loading and editing files and predicates as well as the setting of trace- and spy-points. See Fig. 5.

[2] These issues have been discussed on the comp.lang.prolog newsgroup, April 15-18 2002, subject "ISO catch/throw question".

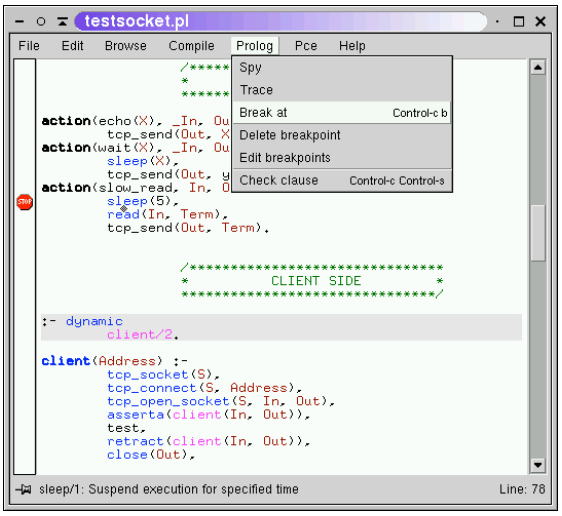


Fig. 4. PceEmacs in action

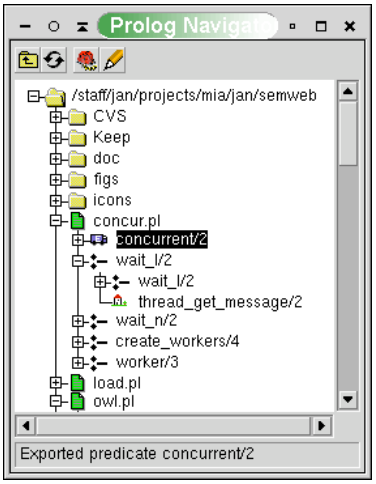


Fig. 5. The Prolog Navigator

5.3 Source-level Debugger

The SWI-Prolog debugger calls a hook (prolog_trace_interception/4) before reverting to the built-in command line debugger. The built in **prolog_frame_attribute/3** provides the infrastructure to analyse the Prolog stacks, providing information on the goal-stack, variable bindings and choicepoints. These hooks are used to realise more advanced debuggers such as the source-level debugger described in this section. The source-level debugger provides three views (Fig. 6):

- *The source*
  An embedded *PceEmacs* (see Sect. 5.1) running in *read-only* mode shows the current location, indicating the current port using colour and icons. *PceEmacs* also allows the setting of *breakpoints* at a specific call in specific clause. Breakpoints provide finer and more intuitive control where to start the debugger than traditional spy-points. Breakpoints are realised by replacing a virtual machine instruction with a *break* instruction which traps the debugger, finds the instruction it replaces in a table and executes this instruction.
- *Variables*
  The debugger displays a list of variables appearing in the current frame with their name and current binding in the top-left window. The representation of values can be changed using the familiar **portray/1** hook. Double-clicking a variable-value opens a separate window showing the variable binding. This window uses indentation to make the structure of the term more explicit and has a menu to control the layout.
- *The stack*
  The top-right window shows the stack as well as the recent active choicepoints. Any node can be selected to examine the context of that node. The stack view allows one to quickly examine choicepoints left after a goal succeeded. Besides showing the location of the choicepoint itself, the 'up' command can be used to examine the parent frame context of a choicepoint.

5.4 Execution Profiler

The *Execution Profiler* builds a call-tree at runtime and ticks the number of calls and redos to each node in this call-tree. The time spent in each

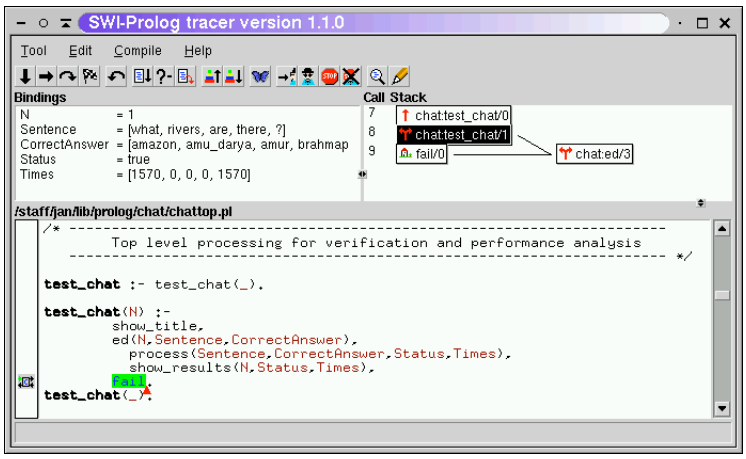


Fig. 6. The Source-level Debugger

node is established using stochastic sampling.[3] Recording the call-tree is complicated by three factors.

- *Last call optimisation*
  Due to last call optimisation exit ports are missing from the execution model. This problem is solved by storing the call-tree node associated with a goal in the environment stack, providing the exit with a reference to the node exited. Recording an *exit* can now exit all nodes until it reaches the referenced node.
- *Redo*
  Having a reference from each environment frame to the call-tree node also greatly simplifies finding the proper location in the call-tree on a *redo*.
- *Recursion*
  To avoid the uncontrolled expanding of the call-tree the system must record *recursive* calls. The problem lies in the definition of *recursion*. The most naïve definition is that recursion happens if there is a parent node running the same predicate. In this view meta predicates will often appear as unwanted 'recursive predicates' as will predicates called in a totally different context. The system provides **noprofile/1** to indicate some predicates do not create a new node and their time is included with their parent node. Examples are **call/1**, **catch/3** and **call_cleanup/2**. Calls are now regarded recursive if the parent node

[3] Using SIGPROF on Unix and using a separate thread and a multi-media timer in MS-Windows.

runs the same predicate (direct recursion) or somewhere in the parent nodes of the call-tree we can find a node running the same predicate with the same immediate parent.

Prolog primitives are provided to extract all information from the recorded call-tree. A graphical Prolog profiling tool presents the information interactively similar to the GNU gprof [4] tool (see Fig. 7).

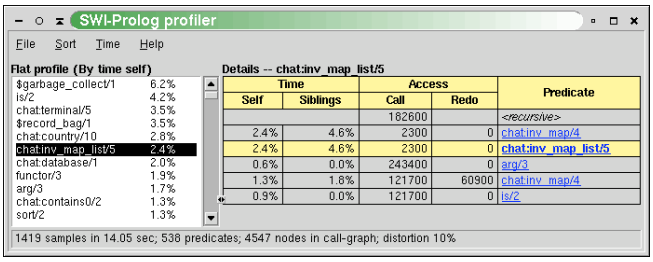


Fig. 7. The Profiler

5.5 Help System

The GUI front end to the help functionality described in Sect. 4.7 adds hyperlinks and hierarchical context to the command line version as illustrated in Fig. 8.

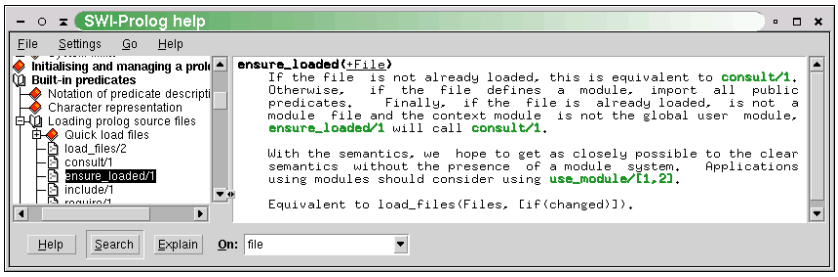


Fig. 8. Graphical front end to the help system

6 Conclusions

In this paper we have described commonly encountered tasks which Prolog programmers spend much of their time on, which tools can help solv-

ing them as well as an overview of the programming environment tools provided by SWI-Prolog. Few of these tools are unique to SWI-Prolog or very advanced. The popularity of the environment can possibly be explained by being complete, open, portable, scalable and free.

Acknowledgements

XPCE/SWI-Prolog is a Free Software project which, by its nature, profits heavily from user feedback and participation. We would like to thank Steve Moyle and Anjo Anjewierden for their comments on draft versions of this paper.

References

1. Lawrence Byrd. Understanding the control flow of Prolog programs. In S.-A. Tarnlund, editor, *Proceedings of the Logic Programming Workshop*, pages 127–138, 1980.
2. M. Carlsson, J. Widén, J. Andersson, S. Anderson, K. Boortz, H. Nilson, and T. Sjöland. *SICStus Prolog (v3) Users's Manual*. SICS, PO Box 1263, S-164 28 Kista, Sweden, 1995.
3. Mireille Ducassé. Analysis of failing Prolog executions. In *Workshop on Logic Programming Environments*, pages 2–9, 1991.
4. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.
5. M. Hermenegildo, G. Puebla, and F. Bueno. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 161–192. Springer-Verlag, 1999.
6. Marija Kulas. Debugging Prolog using annotations. In Mireille Ducassé, Anthony Kusalik, and German Puebla, editors, *Electronic Notes in Theoretical Computer Science*, volume 30. Elsevier, 2000.
7. G. Miller. WordNet: A lexical database for English. *Comm. ACM*, 38(11), November 1995.
8. Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*. Number 10 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, California, 1987. Distributed by Chicago University Press.
9. E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.
10. Jan Wielemaker and Anjo Anjewierden. An architecture for making object-oriented systems available from Prolog. In Alexandre Tessier, editor, *Computer Science, abstract*, 2002. http://lanl.arxiv.org/abs/cs.SE/0207053.

TCLP: A type checker for CLP($\mathcal{X}$)

Emmanuel Coquery
Emmanuel.Coquery@inria.fr

November 7, 2003

Abstract

This paper is a presentation of TCLP: a prescriptive type checker for Prolog/CLP($\mathcal{X}$). Using parametric polymorphism, subtyping and overloading, TCLP can be used with practical constraint logic programs that may use meta-programming predicates, coercions between constraint domains (like $\mathcal{FD}$ and $\mathcal{B}$) and constraint solver definitions, including the CHR language. It also features type inference for variables and predicates, so the user can get rid of numerous type declarations.

1 Introduction

Traditionally, the class CLP($\mathcal{X}$) of constraint logic programs, introduced by Jaffar and Lassez [11], is untyped. One of the advantages of being untyped is programming flexibility. For example, `-/2` can be used as the classical arithmetic operator as well as a constructor for pairs. On the other hand, type checking allows the static detection of some programming errors, like for example calling a predicate with an illegal argument.

Several type systems have been created for (constraint) logic programming. The type system of Mycroft and O'Keefe [12, 15] is an adaptation of the Damas-Milner type system [6] to logic programming. It has been implemented in Gödel [10] and Mercury [19]. This type system uses parametric polymorphism, that is, parameters (*i.e.* type variables) are allowed as and in types. For example the type *list* has an argument to specify the type of elements occurring in the list. However this type system is not flexible enough to be used with meta-programming predicates, such as `arg/3`, `=../2` or `assert/1`.

Subtyping is a fundamental concept introduced by Cardelli [2] and Mitchell [14]. The power of subtyping resides in the subtyping rule which states that an expression of type τ can be used instead of an expression of type τ' provided that τ is a subtype of τ':

$$(Sub)\ \frac{U \vdash t : \tau\ ,\ \tau \leq \tau'}{U \vdash t : \tau'}$$

Subtyping can be used to deal with meta-programming by the introduction of a type *term* as a supertype of all types. For example, the subtype relation $list(\alpha) \leq term$, allows to type check the query `arg(N,[X|L],T)`, using the type $int \times term \times term \rightarrow pred$ for `arg/3`, although the second argument is a list. Subtyping can also be used for coercions between constraint domains. For example, it is possible to share variables between CLP($\mathcal{B}$), with type *boolean*, and CLP($\mathcal{FD}$), with type *int*, simply

by adding the subtyping relation $boolean < int$. This way $\mathcal{B}$ variables can be used with $\mathcal{FD}$ predicates.

Most of the type systems with subtyping that where proposed for constraint logic programs are descriptive type systems, *i.e.* they aim to describe the set of terms for which a predicate is true. On the other hand, there where only few prescriptive type systems with subtyping for logic programming [1, 7, 13, 16, 18]. Moreover, in these systems, subtyping relations between type constructors with different arities, as in $list(\alpha) < term$, are not allowed. Algorithms to deal with such subtyping relations, called non-structural non-homogeneous subtyping, can be found in [17, 20] in the case where the subtyping order forms a lattice, or in [4] for the case of quasi-lattices.

The combined use of subtyping and parametric polymorphism thus offers a great programming flexibility. Still, it can not address the first example given in this paper, that is `-/2` being viewed sometimes as the arithmetic operator and sometimes as a constructor of pairs (as in the predicate `keysort/2`). The solution to this problem resides in overloading. Overloading consists in assigning multiple types to a single symbol. This notion has already been used in numerous languages, such as C, to deal with multiple kinds of numbers in arithmetic operations. With overloading, `-/2` can have both type $int\_expr \times int\_expr \rightarrow int\_expr$ and type $\alpha \times \beta \rightarrow pair(\alpha, \beta)$.

In this paper, we describe TCLP, a type checker for Prolog/CLP($\mathcal{X}$), written in SICStus Prolog with Constraint Handling Rules (CHR) [9]. The type system of TCLP combines parametric polymorphism, subtyping and overloading in order to keep the flexibility of the traditionally untyped CLP($\mathcal{X}$) languages, yet statically detecting programming errors. Section 2 shows examples of how the type system takes advantage of these three features. Section 3 presents the type system of TCLP. In section 4, we describe the basic type declarations and output of TCLP, while section 5 shows how the type system can be extended to handle constraint solver programming, like new CLP($\mathcal{FD}$) constraints or CHR rules. Some benchmarks are presented in section 6 and section 7 concludes.

2 Motivating examples

The aim of the TCLP type checker is to introduce a typing discipline in constraint logic programs in order to find programming errors, while offering enough flexibility for practical programming. That means dealing with Prolog/CLP($\mathcal{X}$) programming facilities like meta-programming or the simultaneous use of multiple constraint solvers. This goal is achieved using a combination of parametric polymorphism, subtyping and overloading. In the rest of this section, we give examples of how they are used in TCLP.

2.1 Prolog examples

A first use of parametric polymorphism is the typing of structures that may be used with any type of data. For example, using the type $list(\alpha)$ for lists allows typing `[1,2]` with the type $list(int)$ and `['a','b']` with the type $list(char)$. A consequence is the use of polymorphic types for predicates manipulating these data structures in a generic way. For

example, the type of the predicate `append/3` for concatenating lists is $list(\alpha) \times list(\alpha) \times list(\alpha) \rightarrow pred$. Of course, some other predicates may use non generic types when manipulating the data inside structures, like `sum_list/2` having type $list(int) \times int \rightarrow pred$.

Another use of parametric polymorphism is for constraints or predicates that can be used on any term, the best example being `=/2` with type $\alpha \times \alpha \rightarrow pred$. This type simply express that the two arguments of `=/2` must have the same type. Another example resides in term comparison predicates like `'@=<'/2`, which also has type $\alpha \times \alpha \rightarrow pred$.

On the other hand, predicates for manipulating terms cannot be typed using only parametric polymorphism. An example is the predicate `=../2` for decomposing terms. Indeed `T=..L` unifies `L` with the list constituted by the head constructor of `T` and the arguments of `T`. This means that `L` is an non-homogeneous list. Subtyping provides a solution for typing this predicate, through the introduction of the type *term* as the supertype of all types, that is for all types τ, $\tau \leq term$. Using the type $term \times list(term) \rightarrow pred$ for `=../2`, it is possible to type check a query like `[1] =.. ['.',1,[]]` with type $list(int)$ for `[1]`, *atom* for `'.'`, *int* for `1`, $list(\alpha)$ for `[]` and $list(term)$ for `['.',1,[]]`.

Subtyping is also interesting when typing programs that use dynamic predicates, using `assert/1`. The type of `assert/1` is $clause \rightarrow pred$ and the type of `':-'/2` is $pred \times goal \rightarrow clause$. This allows typing queries like `assert((p(X) :- X<1))`. However, without subtyping, queries like `assert(p(1))` are not correctly typed because `p(1)` would be typed *pred*, while `assert/1` expects the type *clause*. Using subtyping with $pred < clause$, `p(1)` is seen with the type *clause* and the query is well-typed.

The operator `-/2`, as showed in the introduction, provides a good example of the use of overloading, with types $int\_expr \times int\_expr \rightarrow int\_expr$, $float\_expr \times int\_expr \rightarrow float\_expr$, $int\_expr \times float\_expr \rightarrow float\_expr$, $float\_expr \times float\_expr \rightarrow float\_expr$ and $\alpha \times \beta \rightarrow pair(\alpha, \beta)$. This example shows the more classical overloading of `-/2` with respect to the different kinds of number as well as its use as a coding for pairs. In this case, subtyping can also be used to deal with the different kind of numbers, with $int\_expr < float\_expr$, using the type $\alpha \times \alpha \rightarrow \alpha, \alpha \leq float\_expr$. However, in the Prolog dialects that we considered, the unification `1=1.0` fails. This led us to choose overloading instead of subtyping for dealing with numerical expressions, thus making a clear distinction in types between integers and floats. An other example is `=/2`. It is used both as the equality constraint and to build pairs of the form `Name=Var` in an option of the predicate `read_term/3`. Thus is has both types $\alpha \times \alpha \rightarrow pred$ and $atom \times term \rightarrow varname$. Other examples include options shared by several different predicates or `','/2` used both as the conjunction and as a constructor for sequences.

2.2 Combining constraint domains

A first example is the combination of the Herbrand domain CLP($\mathcal{H}$) with an other domain, such as CLP($\mathcal{FD}$). Prolog is mainly used to handle data structures and for posting constraints. However there can be a stronger interaction when defining, e.g., predicates for labelling. The type used to represent $\mathcal{FD}$ is *int*, already present in the type hierarchy of CLP($\mathcal{H}$). This way $\mathcal{FD}$ variables can be also used as Prolog variables when needed.

Another interesting example is combining CLP($\mathcal{FD}$) and CLP($\mathcal{B}$). Indeed, variables can be shared between the two constraint solvers. This is possible when $\mathcal{B}$ is represented as the set {0,1}. In this case `0` and `1` have type *boolean* and *boolean* $<$ *int*. In this way, $\mathcal{B}$ variables can also be used with $\mathcal{FD}$ constraints.

A last example is reified constraints. This represent a combination of CLP($\mathcal{H}$), CLP($\mathcal{FD}$) and CLP($\mathcal{B}$). Constraints like `'#<=>'/2` accept other constraints as arguments. In order to handle these cases, $\mathcal{FD}$ constraints are typed with type *fd_constraint*. The subtype relations *fd_constraint* $<$ *pred* and *fd_constraint* $<$ *boolean_expr* allows these constraints to be used both in boolean expressions and as predicates in Prolog clauses.

3 The type system

3.1 CLP($\mathcal{X}$) programs

CLP programs are built upon a denumerable set $\mathcal{V}$ of variables, a finite set $\mathcal{S}$ of symbols, given with their arity, a set $\mathcal{F} \subseteq \mathcal{S}$ of function symbols and a set $\mathcal{P} \subseteq \mathcal{S}$ of predicate and constraint symbols. $\mathcal{P}$ is supposed to contain the equality constraint symbol `=/2`. Terms are built upon $\mathcal{F} \cup \mathcal{V}$. An atom is of the form $p(t_1, \ldots, t_n)$, where $p/n \in \mathcal{P}$ and $t_1, \ldots, t_n$ are terms. A query is a finite sequence of atoms. When it is necessary to distinguish predicate atoms (built using a predicate symbol) and constraint atoms (built with a constraint symbol), queries are noted $c \mid \alpha$ where c is the constraint part of the query and α is the predicate part of the query. A clause is an expression $A \leftarrow Q$ where A is a predicate atom and Q is a query. A constraint logic program is a set of clauses and queries.

The execution model we consider for constraint logic programs is the CSLD rewriting relation :

Definition 1 *Let P be a CLP($\mathcal{X}$) program. The rewriting relation $\longrightarrow_{CSLD}$ over queries is defined as the smallest relation satisfying the following CSLD rule:*

$$\frac{\begin{array}{c}\mathtt{p}(N_1, \ldots, N_k) \leftarrow c' \mid A_1, \ldots, A_n \in \theta(P) \\ \mathcal{X} \models \exists(c \wedge M_1 = N_1 \wedge \ldots \wedge M_k = N_k \wedge c')\end{array}}{\begin{array}{l}c \mid \alpha, \mathtt{p}(M_1, \ldots, M_k), \alpha' \longrightarrow_{CSLD} \\ \qquad c, M_1 = N_1, \ldots, M_k = N_k, c' \mid \alpha, A_1, \ldots, A_n, \alpha'\end{array}}$$

where θ is a renaming of the clause with fresh variables.

3.2 Types

Types are (possibly infinite) terms built upon a signature of *type constructors*, denoted by κ, and *type variables* also called *parameters*, noted $\alpha, \beta, \ldots$. Types are noted τ and the set of types is noted $\mathcal{T}$. The subtyping order $\leq$ on types is induced by an order $<_{\mathcal{K}}$ on type constructors and a relation $\iota_{\kappa_1,\kappa_2}$ between the argument positions of each pair (κ_1, κ_2) of type constructors. For all type constructors κ_1, κ_2, $\iota_{\kappa_1,\kappa_2}$ is an injective partial function and $\iota^{-1}_{\kappa_1,\kappa_2} = \iota_{\kappa_2,\kappa_1}$. For all $\kappa_1 \leq_{\mathcal{K}} \kappa_2 \leq_{\mathcal{K}} \kappa_3$, $\iota_{\kappa_1,\kappa_3} = \iota_{\kappa_2,\kappa_3} \circ \iota_{\kappa_1,\kappa_2}$. For two types $\tau = \kappa(\tau_1, \ldots, \tau_m)$ and $\tau' = \kappa'(\tau'_1, \ldots, \tau'_n)$, $\tau \leq \tau'$ if and only if $\kappa \leq_{\mathcal{K}} \kappa'$ and for all $i, j \in \iota_{\kappa,\kappa'}$, $\tau_i \leq \tau'_j$. Moreover the type order is supposed to form a quasi-lattice, that is a partial order where the existence of a lower (resp. upper) bound to a non-empty

set of types implies the existence of a greatest lower bound (resp. least upper bound) for this set. A type substitution is a mapping from type variable to types, extended the usual way into a mapping from types to types. A type substitution Θ is ground if for all type variable α, $\Theta(\alpha)$ is ground.

Ground types are interpreted as sets of terms, while non ground types are interpreted as mappings from ground substitutions to sets of terms. For example, the type $list(int)$ is interpreted as the set of the lists of integers, while the infinite type $list(list(\ldots))$ is interpreted as the set of lists that contain only lists that contain only lists ... [1]. The subtyping relation is interpreted as the inclusion of these sets of terms. A more formal description of types and of the subtyping relation can be found in [4].

To each functor f/n is associated a set $types(f/n)$ of *type schemes* of the form $\forall\alpha_1 \ldots \forall\alpha_n \tau_1 \times \ldots \times \tau_n \rightarrow \tau$, (abbreviated $\forall\tau_1 \times \ldots \times \tau_n \rightarrow \tau$), where $\{\alpha_1, \ldots, \alpha_n\}$ is the set of variables appearing in $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$. We assume the existence of a particular type *pred* for the type of predicates: for all predicate and constraint symbols $p/n \in \mathcal{P}$, it is supposed that there is at least one type scheme $\forall\tau_1 \times \ldots \times \tau_2 \rightarrow \tau \in types(p/n)$ such that $\tau \leq pred$. On can note that some symbols may be overloaded both as function symbols and predicates symbols, such as `=/2` with types $\forall\alpha.\alpha \times \alpha \rightarrow pred$ and $atom \times term \rightarrow varname$.

3.3 Well typed programs

The typing rules of TCLP, given in Table 1, allow to deduce type judgment of the form $U \vdash$ typed expression, where U is a *typing environment*, that is a mapping from $\mathcal{V}$ to $\mathcal{T}$. A clause $p(t_1, \ldots, t_n) \leftarrow Q$ is *well-typed* if for all type schemes $\forall\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in types(p/n)$ with $\tau \leq pred$, there exists a typing environment U such that $U \vdash p(t_1, \ldots, t_n) \leftarrow Q\ Clause_{\tau_1 \times \ldots \times \tau_n}$. A program is *well-typed* if all its clauses are well-typed. A query Q is *well-typed* if there exists a typing environment U such that $U \vdash Q\ Query$.

Basically, the type system of TCLP adds the subtyping rule [2, 14] to the rules of Mycroft and O'Keefe [15]. Overloading is handled in the side condition of rules (*Func*), (*Atom*) and (*Head*) by considering all possible type schemes for each occurrence of overloaded symbols. The type annotations appearing in the rules (*Head*) and (*Clause*) are used to keep track of the type used for the head of the clause. The distinctions between rules (*Head*) and (*Atom*) express the principle of *definitional genericity* [12], that the type of the head of a clause must be equivalent up-to renaming to the type of the predicate defined by this clause. This condition of definitional genericity is useful for the correctness properties ("subject reduction") of the type system [3, 8]. The rule (*Head*), used for typing heads of clauses, thus allows only renaming substitutions of the type declared for the predicate.

Theorem 1 (subject reduction) *[3] Let P be a well-typed program and Q a well typed query, i.e. $U \vdash Q\ Query$ for some typing environment U. If $Q \longrightarrow_{CSLD} Q'$ then there is a typing environment U' such that $U' \vdash Q'\ Query$.*

[1] this does not mean that the terms in this set are infinite: for example `[]`, `[[]]` and `[[],[]]` are in this set.

$$(Var) \qquad \{x : \tau, \ldots\} \vdash x : \tau$$

$$(Func) \qquad \frac{U \vdash t_1 : \sigma_1 \;\; \sigma_1 \leq \tau_1\Theta \;\; \ldots \;\; U \vdash t_n : \sigma_n \;\; \sigma_n \leq \tau_n\Theta}{U \vdash f(t_1,\ldots,t_n) : \tau\Theta}$$

where Θ is a type substitution
and $\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in types(f/n)$

$$(Atom) \qquad \frac{U \vdash t_1 : \sigma_1 \;\; \sigma_1 \leq \tau_1\Theta \;\; \ldots \;\; U \vdash t_n : \sigma_n \;\; \sigma_n \leq \tau_n\Theta}{U \vdash p(t_1,\ldots,t_n) \;\; Atom}$$

where Θ is a type substitution
and $\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in types(p/n)$, with $\tau \leq pred$.

$$(Head) \qquad \frac{U \vdash t_1 : \sigma_1 \;\; \sigma_1 \leq \tau_1\Theta \;\; \ldots \;\; U \vdash t_n : \sigma_n \;\; \sigma_n \leq \tau_n\Theta}{U \vdash p(t_1,\ldots,t_n) \;\; Head_{\tau_1 \times \ldots \times \tau_n}}$$

where Θ is a renaming substitution
and $\tau_1 \times \ldots \times \tau_n \rightarrow \tau \in types(p/n)$, with $\tau \leq pred$.

$$(Query) \qquad \frac{U \vdash A_1 \;\; Atom \;\; \ldots \;\; U \vdash A_n \;\; Atom}{U \vdash A_1,\ldots,A_n \;\; Query}$$

$$(Clause) \qquad \frac{U \vdash Q \;\; Query \qquad U \vdash A \;\; Head_{\tau_1 \times \ldots \times \tau_n}}{U \vdash A \leftarrow Q \;\; Clause_{\tau_1 \times \ldots \times \tau_n}}$$

Table 1: The TCLP typing rules with overloading.

It is worth noting that the CSLD resolution is an abstract execution model, which proceeds only by constraint accumulation. The theorem above does not hold for more concrete execution models that perform substitution steps. Let us consider the predicates `p/1` and `q/1`, with $int \rightarrow pred \in types($`p/1`$)$ and $byte \rightarrow pred \in types($`q/1`$)$. Let us suppose that `p/1` is defined by `p(500)`. The query `p(X),q(X)` is well typed with `X` : *byte*. A step of CSLD resolution produces the query `X=500,q(X)`. A substitution step produces the query `p(500)`, which is not well typed since `500` does not have type *byte*. This can be viewed as a weakness of the type system, but we believe this is the price to pay for flexibility. Moreover, it is possible to keep the type of variables at run-time in order to get stronger subject reduction theorem [8] for an execution model that performs substitution steps.

3.4 Type checking

The typing rules of Table 1 are syntax directed. Without overloading, the type checking algorithm, given a typing environment U and the type of symbols, basically collects subtype inequalities along the derivation of the expression to check and then check the satisfiability of collected subtyping constraints, using the algorithm described in [4]. This type checking algorithm can be extended to infer a typing environment U for which the expression is well-typed, simply by replacing the type of variables appearing in the expression to type check by parameters. Then checking the satisfiability of the resulting subtyping constraint system determines the existence of a typing environment U.

Overloading introduces non-determinism in the rules (*Func*) and (*Atom*). For type checking expressions, subtype inequalities are first collected along the derivation by replacing the type of overloaded symbols by type variables. Then the possible typings for each occurrence of overloaded symbols are enumerated by checking the satisfiability of the subtype constraint system. In order to remain efficient, the enumeration proceeds with the Andorra principle. This principle, first introduced for the parallelization of Prolog [5], consists in delaying choice points until time where all deterministic goals have been executed. This strategy proves to be sufficient to deal with overloading in TCLP, mainly because in most cases the type information coming from the context of an expression is sufficient to disambiguate the type of overloaded symbols in this expression.

The type checking algorithm used in TCLP is simply the combination of the type inference for variables with the enumeration of possible types for overloaded symbols.

3.5 Type inference for predicates

In a prescriptive type system, type reconstruction can be used to omit type declarations and still type check the program by inferring the type of undeclared predicates using their defining clauses [12], if it exists, and raising an error otherwise. Since in TCLP, a predicate can accept any argument of a subtype of the type of declared predicate, the type $term \times \ldots \times term \rightarrow pred$ is always a possible type. Because this type is not very informative, we use a heuristic type inference algorithm [8]. Basically it tries to combine the different type informations taken from the functors and variables appearing the head of the defining clauses to deduce a more informative type. In the presence of overloaded symbols, several heuristic types can be found by enumerating the possible types for these symbols. The current implementation uses only the first one in the typing of the remaining part of the program. This choice was made to avoid the multiplication of overloaded predicates. The enumeration proceeds by first choosing the last declared type for each overloaded symbol. This enumeration strategy proves to be right most of the time, because the last declared type for an overloaded symbol usually correspond to the currently defined predicate.

4 Standard use of TCLP

4.1 Type declarations

We now introduce the concrete syntax of TCLP type declarations. These declarations take the form of Prolog directives. They can be placed either in the program source or in a separated file with the suffix `.typ`. They consist in type constructor declarations, type order declarations and type scheme declarations.

Type constructor declarations are done using any one of the following two syntaxes:

`:- type` t`/`n`.` `:- type` t`(`A_1`,...,`A_n`).`

Both directives declare a a type constructor t with n arguments. For example the type constructor *list* can be declared by

```
:- type list/1.
```

Type order declarations are done using the directive `order`:

```
:- order t(A1,...,Am) < u(B1,...,Bn).
```

which declares that $t <_{\mathcal{K}} u$. The relation $\iota_{t,u}$ is deduced from the variables appearing as arguments: if $A_i = B_j$ then $(i,j) \in \iota_{t,u}$. For example:

```
:- order assoc(A,B) < tree(B).
```

declares that $assoc <_{\mathcal{K}} tree$ and that $\iota_{assoc,tree} = \{(2,1)\}$.

The syntax for declaring type schemes is:

```
:- typeof f(t1,...,tn) is t.
```

where t_i and t are types. This declares that the type scheme $\forall t_1 \times \ldots \times t_n \rightarrow t$ is in $types(f/n)$. For example:

```
:- typeof append(list(A),list(A),list(A)) is pred.
```

declares that $\forall\alpha.list(\alpha) \times list(\alpha) \times list(\alpha) \rightarrow pred \in types($`append/3`$)$. Overloaded symbols simply have several declarations (one per type scheme).

Type constructor and type scheme syntax can also be combined:

```
:- type list(A) is [ [] , [ A | list(A) ] ].
```

is syntactic sugar for

```
:- type list/1.
:- typeof [] is list(A).
:- typeof [ A | list(A) ] is list(A).
```

In addition to explicit declarations, TCLP implicitly adds default declarations. For every declared type constructor κ, the declaration that $\kappa <_{\mathcal{K}} term$ is added to ensure that it is still a supertype of all types. Numbers are implicitly declared to have either type *byte*, *int* or *float*. All non-numeric constants are declared to have type *atom* except for characters, which are declared to have type *char* with $char <_{\mathcal{K}} atom$. Still, thanks to overloading, non-numeric constants may also have other types corresponding to their use in specific situations. For example, `write/0` has both the type *atom* and the type *io_mode*. Using these types, the following query, for opening a file named "write" in writing mode, is well typed: `open(write,write,Stream)`, the first occurrence of write being typed as *atom* and the second as *io_mode*. Finally any functor f/n that has no declared type scheme has the default type scheme $term \times \ldots \times term \rightarrow term$.

4.2 TCLP invocation

TCLP can be used either as a stand-alone executable (by typing `tclp file.pl` in the shell) or as a library for SICStus Prolog. When invoked, TCLP determines and loads a standard type library, usually named `stdlib.typ`. This library contains the type definitions and types for built-in predicates of the selected Prolog dialect, currently either ISO, GNU or SICStus Prolog. In the case of SICStus Prolog, type files for each library are automatically loaded when encountering the corresponding `use_module` directive.

When invoked on a source file, TCLP prints the types inferred for undeclared predicates using the syntax for type scheme declarations. This allows to reuse the types inferred by TCLP for type checking other libraries or same file after some modifications. For example, the type inference of the predicate `append/3`:

```
append([],L,L).
append([X|L],L2,[X|R]) :- append(L,L2,R).
```

produces the following output:

```
:- typeof append(list(A),list(A),list(A)) is pred.
```

If a type error is encountered, TCLP prints it and exits immediately. Here we give examples of ill-typed queries and clauses with the error message displayed by TCLP:

- Illegal type for an argument

```
:- X is Y << 3.5 .

! Incompatible type : 3.5 has type float but is
  required to have type int_expr
```

- No type can be found for a variable

```
:- length(N,L), member(a,L).

! Incompatible types for L : int and list(top)
```

- Violation of definitional genericity

```
:- typeof p(list(A)) is pred.
p([1]).

! Incompatible type : 1 has type byte but is
  required to have type A
```

- Error on an overloaded symbol

```
:- X is 3 << (2 - 3.5).

! Can't find a good type for (-)/2
```

5 Advanced definitions

An interesting feature of TCLP is the possibility to extend the typing rules. The aim is the type checking of phrases that are similar to clauses from the type checking point of view. This extension uses declarations that specify how these phrases must be cut into sets of heads and bodies. The heads are type checked using rules similar to the (*Head*) rule and the bodies are type checked as queries. Note, however, that a new subject reduction theorem must be proved in order to ensure the correctness of the system thus obtained. We show two examples of type system extensions, one for primitive CLP($\mathcal{FD}$) constraints definitions in SICStus Prolog and another for the CHR language [9].

5.1 CLP($\mathcal{FD}$) primitive constraints

In SICStus, primitive constraints can be declared using `'+:'/2`, `'-:'/2`, `'+?'/2` and `'-?'/2`. In order to type check these declarations one may want to introduce new typing rules. This is achieved using the declaration

```
:- tclp__define_clause_op(BinOp,Type).
```

where `BinOp` is the binary operator that separates the head and the body and `Type` is the type of the head. For example, the declaration

```
:- tclp__define_clause_op('+:',fd_constraint).
```

adds the following typing rule:

$$\frac{U \vdash H\ \mathit{Head}' \qquad U \vdash B\ \mathit{Query}}{U \vdash H\ \texttt{+:}\ \ B\ \mathit{Clause}}$$

where $U \vdash H\ \mathit{Head}'$ is derived using the rule (Head'), which differ from (Head) only by the side condition: $\tau \leq \mathit{pred}$ becomes $\tau \leq \mathit{fd\_constraint}$ in (Head').

5.2 CHR rules

There are three kinds of CHR rule: C `==>` Q (propagation rule), C `<=>` Q (simplification rule) and $C_1 \setminus C_2$ `<=>` Q (simpagation rule, *i.e.* both a simplification rule and a propagation rule). C, C_1 and C_2 are sequences of CHR constraints. Q is either a query or Q_1 `|` Q_2 where Q_1 and Q_2 are queries. In order to handle these rules, the declaration `tclp__define_clause/5` is used. We refer to the TCLP documentation for the precise syntax of these declarations. The declarations for CHR rules are given in appendix A. Here we give the typing rule for propagation rules (other rules are similar). Type judgment of the form $U \vdash H\ \mathit{Head}''$ are derived from a rule (Head'') similar to (Head) excepted that the side condition $\tau \leq \mathit{pred}$ is replaced by $\tau \leq \mathit{chr\_constraint}$.

$$\frac{U \vdash H_1\ \mathit{Head}''\ \ldots\ U \vdash H_n\ \mathit{Head}''\ U \vdash B\ \mathit{Query}}{U \vdash H_1, \ldots, H_n\ \texttt{<=>}\ B\ \mathit{Clause}}$$

Type inference can still be used with the new type system, as shown in the following example. This example consists in a constraint solver for finding greatest common divisor and was taken from the CHR web page. The CHR rules:

```
gcd(0) <=> true.
gcd(N) \ gcd(M) <=> N=<M | L is M mod N, gcd(L).
```

produce the following output in TCLP

```
:- typeof gcd(int) is chr_constraint.
```

6 Experimental evaluation

The performance of the system has been evaluated on a GNU/Linux 2.4 system with an Intel Pentium 4 CPU at 2 GHz, 256 Mb of RAM using SICStus 3.9.1 and a preliminary version of TCLP 0.4. Running times for 16 SICStus Prolog libraries are shown in Table 2. The first column indicates the name of the library. The remaining column are divided in two groups: the first group indicates running times when using pure type

checking, that is without type inference for predicates, while the second group indicates running times using type inference for all predicates that are not exported by the library. Each group contains three columns. The first one, *Overld*, is the time consumed to solve ambiguous overloaded symbols. The second one, *T.check* indicates the type checking time (including type inference in the case of the second group). The last column, *Total*, indicates the running total time, including loading type libraries and building the resulting type order.

	Pure type checking			With predicate type inference		
File	Overld	T.check	Total	Overld	T.check	Total
arrays	0.18 s	0.80 s	2.52 s	0.19 s	1.00 s	2.73 s
assoc	0.52 s	2.16 s	3.89 s	0.87 s	3.88 s	5.61 s
atts	0.75 s	1.92 s	3.72 s	1.35 s	3.26 s	5.04 s
bdb	0.84 s	3.14 s	6.08 s	1.07 s	4.19 s	7.06 s
charsio	0.07 s	0.40 s	1.99 s	0.09 s	0.43 s	2.00 s
clpr	29.10 s	47.05 s	49.68 s	97.60 s	142.49 s	145.76 s
fastrw	0.05 s	0.20 s	1.83 s	0.12 s	0.32 s	1.98 s
heaps	0.49 s	1.87 s	3.58 s	1.51 s	5.50 s	7.24 s
jasper	0.32 s	0.98 s	3.21 s	0.48 s	1.36 s	3.52 s
lists	0.96 s	1.86 s	3.44 s	1.23 s	2.63 s	4.24 s
ordsets	0.89 s	2.35 s	3.92 s	3.64 s	7.33 s	8.92 s
queues	0.12 s	0.44 s	2.14 s	0.17 s	0.55 s	2.26 s
sockets	0.82 s	1.83 s	4.02 s	0.77 s	2.12 s	4.15 s
terms	0.44 s	1.32 s	2.90 s	0.54 s	1.72 s	3.31 s
trees	0.27 s	0.79 s	2.47 s	0.32 s	1.17 s	2.89 s
ugraphs	7.39 s	14.17 s	16.28 s	11.20 s	31.97 s	34.04 s

Table 2: Running times

Running times prove that TCLP is fast enough to be used in practice, the worst time being obtained for the clpr library which represents about 4400 lines of code and 527 inferred predicates. When running on small files, most of the running time is used to compute all data structures related to TCLP declarations. These computations usually take 2 to 3 s depending on declarations that are specific to each library. The time used to solve overloaded symbols is very low, usually less than 50% (68% in the worst case) of the total type checking time, thanks to the enumeration strategy. The overhead of type inference w.r.t. pure type checking can be explained by the fact that pure type checking considers the program clauses one by one, while type checking with predicate type inference considers clauses grouped by strongly connected components of the call graphs, which leads to considerably larger subtyping constraint systems and to a higher number of overloaded symbols to be treated at once.

7 Conclusion

We presented TCLP, a prescriptive type checker for Prolog/CLP($\mathcal{X}$), which can be used with practical constraint logic programs. Thanks to parametric polymorphism, subtyping and overloading, it can type check queries and goals using generic data structures, term decomposition and meta-programming predicates, overloaded symbols such as `'-'/2`, or the combination of multiple constraint solvers including reified constraints. The possibility to extend the type system, makes it possible to use TCLP

for constraint solver programming like extending CLP($\mathcal{FD}$) with new constraints or using the CHR language. TCLP features type inference for variables and for predicates, so the user can get rid of numerous type declarations. The experimental evaluation of TCLP on 16 SICStus Prolog libraries, including CLP($\mathcal{R}$), proved that the type checker is fast enough to be used in practice. For these reasons, we believe that TCLP is a good tool for type checking constraint logic programs.

As future work, we intend to develop a formalization of the extensions of the type system. We also want to extend TCLP to other Prolog dialects such as, e.g., Ciao Prolog or SWI Prolog.

Availability TCLP is distributed under the GNU Lesser General Public License, and is available as sources, binaries for Linux/x86 and MacOSX or as a library for SICStus Prolog. An online demo can be found on the TCLP web site:
`http://contraintes.inria.fr/~coquery/tclp`

References

[1] C. Beierle. Type inferencing for polymorphic order-sorted logic programs. In *12th International Conference on Logic Programming ICLP'95*, pages 765–779. The MIT Press, 1995.

[2] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[3] E. Coquery and F. Fages. TCLP: overloading, subtyping and parametric polymorphism made practical for constraint logic programming. Technical report, INRIA Rocquencourt, 2002.

[4] E. Coquery and F. Fages. Subtyping constraints in quasi-lattices. In P. Pandya and J. Radhakrishnan, editors, *Proceeding of the 23rd Conference On Foundations Of Software Technology And Theoretical Computer Science*, LNCS. Springer, 2003.

[5] V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I preprocessor: Supporting full Prolog on the basic Andorra model. In *Proceedings of the 8th International Conference on Logic Programming ICLP'91*, pages 443–456. MIT Press, 1991.

[6] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 207–212, 1982.

[7] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In H. Ganzinger, editor, *Proceedings of the European Symposium on Programming ESOP'88*, LNCS, pages 79–93. Springer-Verlag, 1988.

[8] F. Fages and E. Coquery. Typing constraint logic programs. *Theory and Practice of Logic Programming*, 1(6):751–777, November 2001.

[9] T. Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, October 1998.

[10] P. Hill and J. Lloyd. *The Gödel programming language*. MIT Press, 1994.

[11] J. Jaffar and J.L. Lassez. Constraint logic programming. In *Proceedings of the 1987 Symposium on Principles of Programming Languages POPL'87*, pages 111–119, 1987.

[12] T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proceedings of the 1991 International Symposium on Logic Programming*, pages 202–217. MIT Press, 1991.

[13] G. Meyer. Type checking and type inferencing for logic programs with subtypes and parametric polymorphism. Technical report, Informatik Berichte 200, Fern Universitat Hagen, 1996.

[14] J. Mitchell. Coercion and type inference. In *Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages POPL'84*, pages 175–185, 1984.

[15] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[16] F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

[17] F. Pottier. Simplifying subtyping constraints: a theory. *To appear in Information and Computation*, 2002.

[18] G. Smolka. Logic programming with polymorphically order-sorted types. In *Algebraic and Logic Programming ALP'88*, number 343 in LNCS, pages 53–70. J. Grabowski, P. Lescanne, W. Wechler, 1988.

[19] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, 1996.

[20] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium SAS'96*, number 1145 in LNCS, pages 349–365, 1996.

A TCLP declarations for CHR rules

We use an auxiliary Prolog file, `chrcore.pl`, to decompose CHR rules in sets of heads and bodies. The predicate `chr__heads/3` decomposes a sequence of heads into a list of Head-Location-Type triplets, while the predicate `chr__clauses/4` breaks a rule into a body and a list of heads. In the last clause, the type *chr_constraint* is specified, which leads TCLP to use the rule ($Head''$).

The predicate `user:arg_location/2` is predefined in TCLP and is used to provide the location of the different parts of the rule in the program source code to TCLP, mainly for reporting errors in the right place.

```
myappend([],X,X).
myappend([X|L],L2,[X|R]) :- myappend(L,L2,R).

%% rule decomposition
chr__clause((HeadsDef <=> Body), Location,
            Heads, [ Body - BodyLoc ]) :-
        user:args_location(Location,[HeadsLoc, BodyLoc]),
        chr__heads(HeadsDef, HeadsLoc, Heads).
chr__clause((HeadsDef ==> Body), Location,
            Heads, [ Body - BodyLoc ]) :-
```

```
        user:args_location(Location,[HeadsLoc, BodyLoc]),
        chr__heads2(HeadsDef, HeadsLoc, Heads).

%%  sequence of heads to list
chr__heads((H1\H2), Location, Heads) :- !,
        user:args_location(Location,[L1,L2]),
        chr__heads2(H1,L1,Heads1),
        chr__heads2(H2,L2,Heads2),
        myappend(Heads1, Heads2, Heads).
chr__heads(H,L,Hds) :-
        chr__heads2(H,L,Hds).

chr__heads2((H1,H2), Location, Heads) :- !,
        user:args_location(Location,[L1,L2]),
        chr__heads2(H1,L1,Heads1),
        chr__heads2(H2,L2,Heads2),
        myappend(Heads1, Heads2, Heads).
chr__heads2(H,L,[H-L-chr_constraint]).
```

The following code comes from the type declaration file for the CHR library, `chr.typ`. The first directive loads the code from `chrcore.pl`. The two last directives define, given a rule and its location, a list of heads and a list of bodies, using `chr__clause/4` from `chrcore.pl`. Using these directives, TCLP will decompose CHR rules in sets of heads and bodies, heads being type checked with the rule (*Head″*), while bodies are type checked as queries. The difference between the second and the third directive is that the second directive discards the name of rules (names are given to rules in CHR using the notation *Name* `@` *Rule*).

```
%% load prolog code for parsing CHR rules
:- tclp__load_prolog(tclplib('sicstus/chrcore.pl')).

%% the declarations simply consist in the call to predicates
%% defined in chrcore.pl
:- tclp__define_clause((_ @ Rule), Location, Heads, Bodies,
                       (user:args_location(Location,
                                           [_,RuleLoc]),
                           chr__clause(Rule, RuleLoc,
                                       Heads, Bodies))).
:- tclp__define_clause(Rule, Location, Heads, Bodies,
                       chr__clause(Rule, Location,
                                   Heads, Bodies)).
```

Analyzing and Visualizing PROLOG Programs based on XML Representations

Dietmar Seipel[1], Marbod Hopfner[2],
Bernd Heumesser[2]

[1] University of Würzburg, Institute for Computer Science
Am Hubland, D – 97074 Würzburg, Germany
seipel@informatik.uni-wuerzburg.de

[2] University of Tübingen, Wilhelm–Schickard Institute for Computer Science
Sand 13, D – 72076 Tübingen, Germany
{hopfner, heumesser}@informatik.uni-tuebingen.de

Abstract

We have developed a PROLOG package VISUR/RAR for reasoning about various types of source code, such as PROLOG rules, JAVA programs, and XSLT stylesheets. RAR provides techniques for *analyzing* and *improving* the *design* of PROLOG programs, and it allows for implementing *software engineering metrics* and *refactoring techniques* based on XML representations of the investigated code. The obtained results are visualized by graphs and tables using the component VISUR.

VISUR/RAR can significantly improve the development cycle of logic programming applications, and it facilitates the implementation of techniques for syntactically analyzing and visualizing source code. In this paper we have investigated the dependency structure between the different rules and the hierarchical structure of PROLOG software systems, as well as the internal structure of individual predicate definitions.

For obtaining efficiency and for representing complex deduction tasks we have used techniques from deductive database and non–monotonic reasoning.

Keywords. comprehension, refactoring, reasoning, visualization, PROLOG, XML

1 Introduction

For many programming languages, powerful *integrated development environments (IDEs)* have been developed, such as IBM's Eclipse for JAVA [12], and Together for JAVA, C++, Visual Basic, etc. They contain tools such as editors with *syntax highlighting*, *tracing* and *debugging* and tools for *graphical programming*. Advanced IDEs support programmers in managing large projects, e.g. by facilitating the tasks of correcting, completing and reusing source code. In the logic programming community, so far only few tools exist for comfortably programming and for analyzing source code, cf., e.g., the IDEs for XPCE–PROLOG [20] and for Visual PROLOG, and the tool Cider [7] for the functional–logic language Curry.

The package VISUR/RAR [9] provides some essential functionality of an IDE for PROLOG. It allows for the *visualization* of rules (VISUR: Visualization of Rules) together with the *inference* over rule structures (RAR: Reasoning about Rules). VISUR/RAR is a part of the toolkit DISLOG, which is developed under XPCE/SWI–PROLOG [20]. The functionality of DISLOG ranges from (non–monotonic) reasoning in disjunctive deductive databases to applications such as the management and visualization of stock information.

The goal of the system VISUR/RAR is to support the application of *software engineering* and *refactoring* techniques, and the further system development. VISUR/RAR facilitates program comprehension and review, design improvement by refactoring, the extraction of subsystems, and the computation of software metrics (such as, e.g., the degree of abstraction). It helps programmers in becoming acquainted with source code by visualizing dependencies between different predicates or source files of a project. It is possible to analyse source code customized to the individual needs of a user, and to visualize the results graphically or in tables.

VISUR/RAR can be applied to various kinds of rule–based systems, including expert systems, diagnostic systems, XSLT stylesheets, etc. In this paper we have applied VISUR/RAR to the source code of the system DISLOG, which currently contains about 80.000 lines of code in SWI–PROLOG in about 10.000 PROLOG rules. In previous papers [9, 10] we have shown how also JAVA source code can be analysed using VISUR/RAR. For gaining sufficient performance on large programs such as DISLOG we use techniques from the field of deductive databases.

The rest of the paper is organized as follows: In Section 2 we introduce our PROLOG library for managing XML documents. In Section 3 we briefly describe the graph visualization tool VISUR. In Sections 4 and 5 we investigate some typical problems and questions that might be asked about the global and the local structure of PROLOG rules, respectively, and we show how we can solve these problems using RAR.

2 Representation of Source Code

In VISUR/RAR complex structured objects, such as JAVA programs, PROLOG programs, and XSLT stylesheets, are conceptually treated in two different XML notations; for handling these XML data we use the library FNQUERY, which is part of the DISLOG unit `xml` [17].

Firstly, RAR uses a notation which is similar to RuleML [19] for representing PROLOG defintions and rules; our DTD differs slightly from RuleML, and moreover we only use some of the elements and attributes mentioned in RuleML. Secondly, VISUR transforms our XML representation of rules and our reportings into the *Graph eXchange Language* GXL [11]; we added some additional attributes to the GXL notation for configuring the graph display.

2.1 PROLOG Programs in XML

The following PROLOG predicate `tc/2` computes the transitive closure of the relation `arc/2`:

```
tc(U1, U2) :-
   arc(U1, U3), tc(U3, U2).
tc(U1, U2) :-
   arc(U1, U2).
```

We are representing PROLOG programs in XML according to a DTD (see appendix), which is suitable for handling disjunctive logic programs with arbitrarily many head atoms. The arguments of an atom are either terms or variables; constants are represented as terms without subterms, where the constant is stored in the attribute `functor`. E.g., the PROLOG rules for the predicate `tc/2` are represented as follows:

```
<definition predicate="(user:tc)/2">
   <rule file="transitive_closure">
      <head>
         <atom predicate="(user:tc)/2">
            <var name="U1"/> <var name="U2"/> </atom>
      </head>
      <body>
         <atom predicate="(user:arc)/2"> ... </atom>
         <atom predicate="(user:tc)/2"> ... </atom>
      </body>
   </rule>
   ...
</definition>
```

We use the naming convention `(Module:Predicate)/Arity` for predicates. If a predicate is not defined in a module, then it is automatically assigned to the global module `user`.

2.2 Complex Objects in PROLOG

A complex object can be represented as an *association list* $[a_1 : v_1, \ldots, a_n : v_n]$, where a_i is an *attribute* and v_i is the associated *value*; this representation is well–known from the field of artificial intelligence. Using the field notation has got several advantages compared to ordinary PROLOG facts "$\mathsf{object}(v_1, \ldots, v_n)$". The sequence of attribute/value–pairs is arbitrary. Values can be accessed by attributes rather than by argument positions. Null values can be omitted, and new values can be added at runtime.

In the PROLOG library FNQUERY this formalism has been extended to the *field notation* for XML documents: an XML element

$$\langle \mathsf{Tag}\ a_1 = "v_1" \ldots a_n = "v_n" \rangle\ \mathsf{Contents}\ \langle /\mathsf{Tag} \rangle$$

with the tag "Tag" can be represented as a PROLOG term $\mathsf{Tag} : \mathsf{As} : \mathsf{C}$, where As is an association list for the attribute/value–pairs $a_i = "v_i"$ and C represents the contents, i.e., the subelements. E.g., for the XML representation of the atom `tc(U1, U2)` we get:

```
atom:[predicate:'(user:tc)/2']:[
   var:[name:'U1']:[], var:[name:'U2']:[] ]
```

There exist several possibilities to *access* and *update* an object `O` in field notation using a binary infix predicate "`:=`", which evaluates its right argument and tries to unify the result with its left argument. Given an element tag `E` and an attribute `A`, we use the call `X := O^E` to select the `E`–subelement `X` of `O`, and we use `Y := O@A` to select the `A`-value `Y` of `O`; the application of selectors can be iterated, cf. path expressions in XML query languages [1]. On *backtracking* all solutions to a path expression can be obtained.

```
?- Atom = atom:[predicate:'(user:tc)/2']:[
      var:[name:'U1']:[], var:[name:'U2']:[] ]
   P := Atom@predicate, V := Atom^var,
   findall( N,
      N := Atom^var@name,
      Ns ).

P = '(user:tc)/2', V = var:[name:'U1']:[], Ns = ['U1', 'U2']

Yes
```

To change the values of attributes or subelements, the call `X := O*As` is used, where `As` specifies the new elements or attribute/value–pairs in the updated object `X`:

```
?- Atom = atom:[predicate:'(user:tc)/2']:[
      var:[name:'U1']:[], var:[name:'U2']:[] ],
   Atom_2 := Atom*[@predicate:'closure/2'].

Atom_2 = atom:[predicate:'closure/2']:[
   var:[name:'U1']:[], var:[name:'U2']:[] ]

Yes
```

The library FNQUERY also contains additional, more *advanced methods*, such as the selection/deletion of all elements/attributes of a certain pattern, the transformation of subcomponents according to substitution rules in the style of XSLT, and the manipulation of path or tree expressions.

3 Visualization of PROLOG Rules in VISUR

For visualizing the call structure of rule–based systems the concept of *dependency graphs*, which is well–known from deductive databases [2], is used. DATALOG programs can be analysed using diverse dependency graphs, e.g., the *rule/goal graph* and the *goal graph*.

All screenshots of dependency graphs that are shown in this paper have been obtained using our system VISUR. We use a *circle* for ordinary predicates; the name and the arity of the predicate are given below the symbol. For each rule, we use a *box*; the filename below the box gives the file in which the rule is defined.

The Rule/Goal Graph. Given a PROLOG program P and a rule

$$r = A \leftarrow B_1 \wedge \ldots \wedge B_m \in P,$$

the concept of the rule/goal graph $G_r^{\mathsf{rg}} = \langle V_r^{\mathsf{rg}}, E_r^{\mathsf{rg}} \rangle$ of r is well–known from literature:

$$\begin{aligned} V_r^{\mathsf{rg}} &= \{ p_A, r \} \cup \{ p_{B_i} \mid 1 \leq i \leq m \}, \\ E_r^{\mathsf{rg}} &= \{ \langle p_A, r \rangle \} \cup \{ \langle r, p_{B_i} \rangle \mid 1 \leq i \leq m \}, \end{aligned}$$

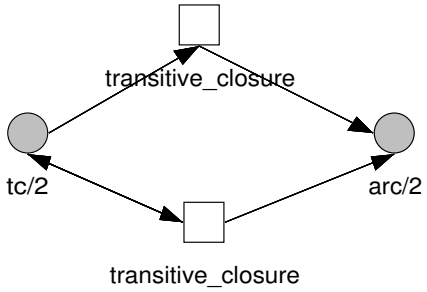


Figure 1: Rule/Goal Graph in VISUR

where p_X is the predicate name of an atom $X = p(t_1, \ldots, t_n)$, i.e. $p_X = p$. The rule/goal graph of P is $G_P^{\mathsf{rg}} = \bigcup_{r \in P} G_r^{\mathsf{rg}}$.

Meta–Call Predicates in PROLOG**.** Unfortunately, these graphs cannot handle practical applications of PROLOG properly, since they do not take into account meta–call predicates. Meta–call predicates allow for higher–order programming; some well–known meta–call predicates are for instance `call/1`, `findall/3`, `forall/2`, `maplist/3`, and `checklist/2`. E.g., the evaluation of `findall/3` within the following rule repeatedly calls the predicate `calls_uu_pred/2` and collects the results in the list `Predicates`:

```
calls_uu(Program, (Unit_1-Unit_2):N) :-
   unit(Unit_1), unit(Unit_2),
   Unit_1 =\= Unit_2,
   findall( Predicate,
      calls_uu_pred(Program, (Unit_1-Unit_2):Predicate),
      Predicates ),
   length(Predicates, N).
```

To treat *meta–call predicates* in PROLOG programs adequately, we will define extended dependency graphs, which include the predicates called in the parameter list of a meta–call predicate. Figure 2 contains the extended dependency graph of the rule generated by VISUR. We use a *rhombus* as the symbol for meta–call predicates in order to distinguish them from ordinary predicates, and we use a *triangle* as the symbol for built–in predicates.

In the rule/goal graph there would be no edge from `findall/3` to `calls_uu_pred/2`. We introduce the notation $p_B \prec p_A$ for expressing that an atom B with the predicate symbol p_B is called within an atom A with the predicate symbol p_A. Thus, in our example we get

`calls_uu_pred/2` $\prec$ `findall/3`.

In VISUR it is possible to define the relation $\prec$ customized to the individual user needs. E.g., if the predicate symbol p_B is constructed at runtime by appending some dynamic suffix p''_B to a static prefix p'_B, i.e. $p_B = p'_B \circ p''_B$, then we can specify that $p_B \prec p_A$ holds for all p_B with the prefix p'_B. In Section 4 we will see such predicates, where $p'_B = \mathsf{calls\_}$ and $p''_B \in \{\, \mathsf{fp}, \mathsf{uu} \,\}$.

The Extended Rule/Goal Graph. The extended rule/goal graph $G_P^{\mathsf{erg}} = \langle V_P^{\mathsf{erg}}, E_P^{\mathsf{erg}} \rangle$ takes care of the fact that an atom B can be called within another atom A. The node set $V_P^{\mathsf{erg}} = \cup_{A \in HB_P} V_A^{\prec}$ and the edge set $E_P^{\mathsf{erg}} = \cup_{A \in HB_P} E_A^{\prec}$ are obtained by collecting the following node and edge sets for the ground atoms A in the Herbrand base HB_P:

$$\begin{aligned} V_A^{\prec} &= \{ p_A \} \cup (\cup_{p_B \prec p_A} V_B^{\prec}), \\ E_A^{\prec} &= \cup_{p_B \prec p_A} (\{ \langle p_A, p_B \rangle \} \cup E_B^{\prec}). \end{aligned}$$

The extended rule/goal graph of Figure 2 visualizes most of the PROLOG rules of Section 4; the rule for the predicate `calls_uu/2`, that is shown above, is one of them.

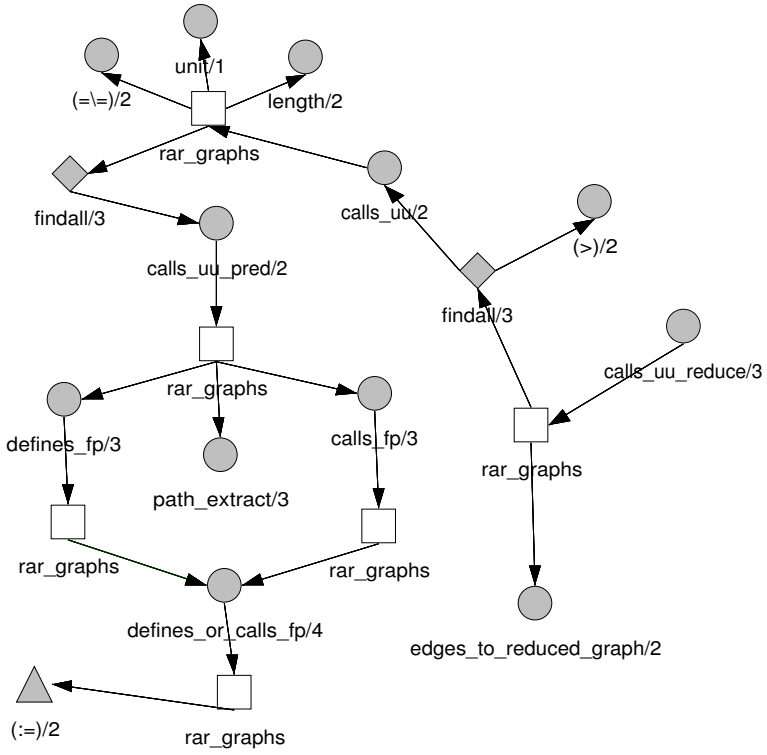


Figure 2: Extended Rule/Goal Graph in VISUR

Further graphs such as the *goal graph*, the *file dependency graph*, the *module dependency graph*, and the *unit dependency graph* can be derived from the extended rule/goal graph.

4 Analysis of Dependency Graphs in RAR

In this section we present some PROLOG rules for extracting structural design information from *hierarchically structured* PROLOG programs. We will demonstrate by some case studies how the quality of the arrangement of the predicates in the source code files can be measured.

We have applied our analysis to the system DISLOG, which is hierarchically structured into units, modules, and files; i.e., in DISLOG a module consists of serveral files, whereas in PROLOG a module usually consists of a single file. Currently, the complete name (path) of a DISLOG source file is of the form

```
DisLog/sources/<unit>/<module>/<file>,
```

and the call `path_extract(Type, Path, X)` extracts `X = <unit>`, `X = <module>`, and `X = <file>`, from such a path `Path`, where `Type` is one of `unit`, `module`, and `file`, respectively.

4.1 Basic Dependencies between Files and Predicates

Given a program `Program` in field notation, the following predicate `defines_fp/3` describes that the predicate `Predicate` is defined (i.e., it occurs in the head of a rule) in the file `File`, and `calls_fp/3` describes that `Predicate` is called in the body of a rule in `File`:

```
defines_fp(Program, File, Predicate) :-
   defines_or_calls_fp(Program, File, head, Predicate).

calls_fp(Program, File, Predicate) :-
   defines_or_calls_fp(Program, File, body, Predicate).

defines_or_calls_fp(Program, File, Selector, Predicate) :-
   Rule := Program^definition^rule,
   File := Rule@file,
   Predicate := Rule^Selector^atom@predicate.
```

The predicate `defines_or_calls_fp/4`, which extracts the common part of the two predicates `defines_fp/3` and `calls_fp/3`, uses the variable `Selector` in a path expression. E.g., calling `defines_fp(Program, File, '(user:calls_fp)/3')` for the representation of the DISLOG system determines the file that defines `'(user:calls_fp)/3'`:

```
File = 'DisLog/sources/databases/rar/rar_metrics'
```

4.2 Derived Dependencies between Units

A unit `Unit_1` calls a unit `Unit_2`, if there exists a file `File_2` in `Unit_2` defining a predicate `Predicate` that is called in a file `File_1` in `Unit_1`. In practice it turned out that in DISLOG there exist calls between most units, but there are great differences between the numbers of such calls. The following predicate `calls_uu/2` returns elements of the form `(Unit_1-Unit_2):N` by counting the number `N` of predicate calls from `Unit_1` to `Unit_2`:

```
calls_uu(Program, (Unit_1-Unit_2):N) :-
   unit(Unit_1),
   unit(Unit_2),
   Unit_1 =\= Unit_2,
   findall( Predicate,
      calls_uu_pred(Program, (Unit_1-Unit_2):Predicate),
      Predicates ),
   length(Predicates, N).
```

```
calls_uu_pred(Program, (Unit_1-Unit_2):Predicate) :-
   calls_fp(Program, File_1, Predicate),
   defines_fp(Program, File_2, Predicate),
   path_extract(unit, File_1, Unit_1),
   path_extract(unit, File_2, Unit_2).
```

In most situations where a unit calls another unit very few times, these calls are due to missplaced predicate definitions; these predicate definitions could be moved to other places, such that the calls disappear. In Figure 3 the remaining part of the unit dependency graph of DISLOG is shown.

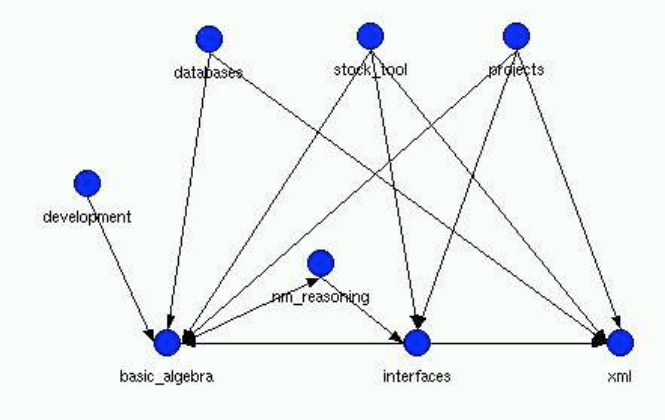


Figure 3: Unit Dependency Graph of DISLOG in VISUR

Figure 3 suggests to partition the set of 8 DISLOG units into two *levels*: The lower level consists of the units `basic_algebra`, `interfaces`, `nm_reasoning`, and `xml`; the higher level consists of `development`, `databases`, `stock_tool`, and `projects`.

We have computed the *reduction* of the unit dependency graph of Figure 3 to *strongly connected components* based on the standard library `ugraphs.pl` of SWI–PROLOG: `[basic_algebra, interfaces, nm_reasoning]` is a strongly connected component. By joining the strongly connected components into higher–level packages, we can obtain an *acyclic* package structure.

Extracting a Maximal Acyclic Unit Structure

Another alternative for *improving the design* of a system could be to identify a small set of predicates that should be moved from one unit to another unit, such that the resulting unit dependency graph is acyclic. This could be done by extracting a maximal acyclic subgraph of the unit dependency graph, such that the neglected edges represent the smallest number of calls. Thus, the following program derives a relation `arc/2` that is a subset of the relation `calls_uu/2`.

```
arc(U1, U2) :-
   calls_uu(U1, U2),
   not(tc(U2, U1)).

tc(U1, U2) :-
   arc(U1, U2).
```

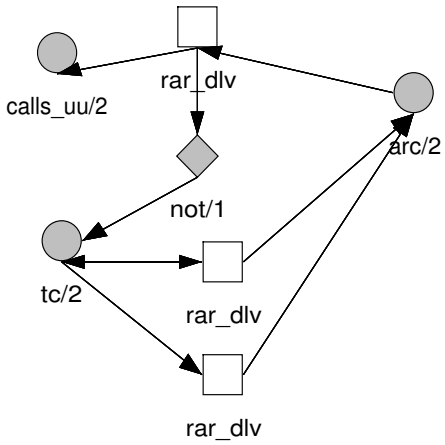


Figure 4: Rule/Goal Graph in VISUR

```
tc(U1, U2) :-
   arc(U1, U3), tc(U3, U2).
```

The program cannot be evaluated using PROLOG, since the recursion defining the predicate `arc/2` envolves negation, cf. Figure 4. Thus, we have used the *disjunctive* logic programming system dlv [4] for evaluating the program. dlv is able to derive the so–called *answer sets*, which are certain minimal sets of atoms satisfying the rules. Obviously, we can focus on the non–trivial strongly connected component `[basic_algebra, interfaces, nm_reasoning]`. The (partial) solutions obtained by *answer set programming* with dlv are given in Figure 5. They show that we can obtain three different maximal acyclic unit structures.

The most reasonable one is given by the answer set M_3, which indicates to neglect the edge from `basic_algebra` to `nm_reasoning` in the unit dependency graph; in that case only 20 predicate defintions would have to be moved from one unit to another unit (obviously, this might not always be possible). In comparison, M_1 would neglect 187 calls, and M_2 would neglect 191 calls.

Unit	Unit	Calls	M_1	M_2	M_3
basic_algebra	nm_reasoning	20	✓	✓	
interfaces	basic_algebra	41		✓	✓
nm_reasoning	interfaces	45	✓		✓
nm_reasoning	basic_algebra	146			✓

Figure 5: Answer Sets Computed by dlv

We have used dlv since we are planning to analyse *disjunctive* logic programs in the future as well. Our current program above, however, is a *normal* logic program, and we could also evaluate it using the answer set programming system Smodels, which is restricted to normal logic programs. Both dlv and Smodels can compute optimal models; thus, it is possible to compute the *optimal* answer set M_3 without computing all answer sets.

5 Analysis of the Internal Structure of Rules

In this section we analyze the internal structure of individual PROLOG rules or definitions. In particular we compute *software metrics*, we search for *design patterns*, and we apply *refactoring* techniques. The examples demonstrate the usefulness of our PROLOG library for handling XML representations of source code.

As an example of PROLOG–specific software metrics, in Section 5.1 we show how the percentage of directly recursive predicates in DISLOG can be calculated, and we compare it with the percentage of meta–call predicates.

In Section 5.2 we describe the detection and the refactoring of patterns. We detect directly recursive predicates having a special form that could in fact be replaced using the meta–call predicate `maplist/3`. Other patterns that one could look for are the accumulator pattern for traversing a list – as it can be found in the predicate `sumlist/2` of the library `lists.pl` of SWI–PROLOG.

5.1 Directly Recursive Rules vs. Meta–Call Predicates

The following predicate counts the number of directly recursive rules for each unit. The results are a multiset `Units` consisting of pairs `Unit:K`, such that `K` is the number of directly recursive rules in `Unit`, and an integer `N`, which is the overall number of directly recursive rules in all units.

```
directly_recursive_predicates(Program, Units, N) :-
   findall( Unit,
      ( Definition := Program^definition,
        Predicate := Definition@predicate,
        Rule := Definition^rule,
        Predicate := Rule^body^atom@predicate,
        Path := Rule@file,
        path_to_unit_name(Path, Unit) ),
      Units_2 ),
   list_to_multiset(Units_2, Units),
   length(Units_2, N).
```

The predicate `list_to_multiset/2` converts a list – with possibly multiply occurring elements – into a list of pairs `Element:Multiplicity`, which represents a multiset. Applying the predicate `directly_recursive_predicates/3` to DISLOG has shown that there are 1645 directly recursive rules in DISLOG. Figure 6 shows the percentage of such predicates per module.

Sometimes, meta–call predicates, such as `maplist/3` or `checklist/2` can be used instead of directly recursive calls. In many cases they are a better choice, since they are more readable. Moreover, meta–call predicates are often implemented in the kernel, and they are faster than directly recursive calls.

The distribution in Figure 6 shows, that in the units of level 1 often directly recursive calls are used, whereas the distribution in Figure 7 shows, that in the units of level 2 meta–call predicates are used more often; the only exception is the unit `xml`.

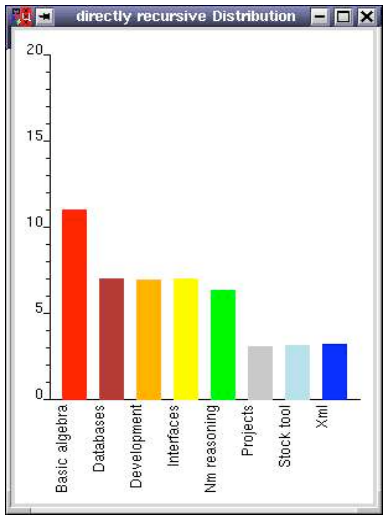


Figure 6: Rules with Directly Recursive Predicates (in %)

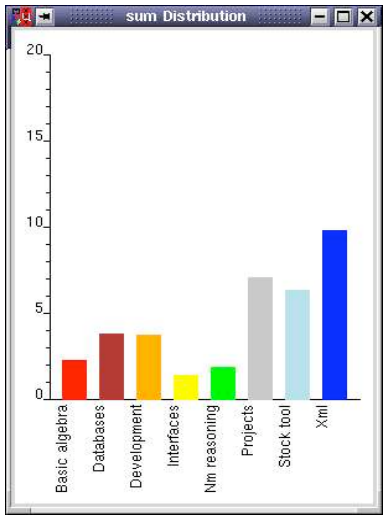


Figure 7: Rules with Meta–Call Predicates (in %)

5.2 Detection of Patterns and Refactoring

The following two rules define the predicate `vector_multiply/3` using direct recursion:

```
vector_multiply(F, [X|Xs], [Y|Ys]) :-
   Y is F * X,
   vector_multiply(F, Xs, Ys).
vector_multiply(_,[],[]).
```

Every list element `X` of the input list is multiplied by `F` to obtain a list element `Y` of the output list. A more abstract – and more elegant – implementation of `vector_multiply/3` can be obtained using the meta–call predicate `maplist/3`:

```
vector_multiply(F, Xs, Ys) :-
   maplist( multiply(F),
      Xs, Ys ).

multiply(F, X, Y) :-
   Y is F * X.
```

The version using `maplist/3` is shorter, since the predicate `multiply/3` already exists in DISLOG. For complex transformations – where the input element `X` is transformed into the output element `Y` using a complex sequence of operations – the version using `maplist/3` in most cases is much more readable. The same holds for predicates with many parametric arguments such as `F` in `vector_multiply/3`.

An alternative implementation is possible using the PROLOG library loops.pl, cf. [14], where it is argued that the enhancement of PROLOG by concepts – such as logical loops – that are familiar from other programming languages increases productivity and mantainability:

```
vector_multiply(F, Xs, Ys) :-
   ( foreach(X, Xs), foreach(Y, Ys) do
        Y is F * X ).
```

Thus, we have implemented an automatic detection and refactoring of maplist patterns. The following two important predicates are applied to a linear recursive rule for the predicate symbol `Predicate`. `list_argument/4` is applied to the head atom `Atom` of the rule; it detects a list pattern `X:Xs` and returns its argument position `N`. `variable_argument/4` is applied to a body atom `Atom` of the rule; it checks if the argument at position `N` is `Variable`:

```
list_argument(Atom, Predicate, N, X:Xs) :-
   Predicate := Atom@predicate,
   Argument := Atom-nth(N)^_,
   '.' := Argument@functor,
   X := Argument-nth(1)^var@name,
   Xs := Argument-nth(2)^var@name.

variable_argument(Atom, Predicate, N, Variable) :-
   Predicate := Atom@predicate,
   Argument := Atom-nth(N)^_,
   Variable := Argument@name.
```

`Argument := Atom-nth(N)^_` selects the `N`–th argument of `Atom`, independently of its tag, and, if `Argument` is a list (i.e., its functor is "`.`"), then `X := Argument-nth(1)^var@name` selects the head `X` of the list and `Xs := Argument-nth(2)^var@name` selects the tail `Xs`. For our recursive rule we can apply these predicates to the head and the body atom, respectively, with the predicate `vector_multiply/2`:

```
?- Atom_1 = atom:[predicate:'vector_multiply/3']:[
      var:[name:'F']:[],
      term:[functor:'.']:[
```

```
            var:[name:'X']:[], var:[name:'Xs']:[]],
         term:[functor:'.']:[
            var:[name:'Y']:[], var:[name:'Ys']:[]]],
      Atom_2 = atom:[predicate:'vector_multiply/3']:[
         var:[name:'F']:[],
         var:[name:'Xs']:[], var:[name:'Ys']:[]],
      list_argument(Atom_1, P, N, X:Xs),
      variable_argument(Atom_2, P, N, Xs).

P = 'vector_multiply/3', X = 'X', Xs = 'Xs', N = 2;
P = 'vector_multiply/3', X = 'Y', Xs = 'Ys', N = 3

Yes
```

If we find a pair `N,N'` of corresponding positions (above we found `N = 2` and `N' = 3`), then further predicates have to check that there is no other recursive rule for `Predicate`, that the list variables `Xs` and `Ys` don't occur in the remaining body atoms (i.e., in "`Y is F * X`" in our example), that the arguments in the other positions in the recursive atom in the head and the body, respectively, are identical (i.e., the first argument `F`), and that the non–recursive rule for `Predicate` is just an atom with "`[]`" in the positions `N,N'` and underscore variables in the remaining positions.

It turned out that there exist 176 maplist patterns in the 4 basic level units of DISLOG; on the other hand, we have computed that there are 78 calls to `maplist/3` in these 4 units. In the 4 higher–level units, there exist 54 maplist patterns, and there are 388 calls to `maplist/3`.

6 Conclusions

VISUR/RAR can be used for analyzing and for refactoring source code; it can help to improve the initial design of a system. By using an extended definition of dependency graphs it became possible to treat meta–call predicates in PROLOG adequately. Our approach is based on heuristics; in general it is not statically decidable which calls are to be actually done at run time in the presence of meta–calls and rules that are asserted or retracted at runtime.

The integration of XML processing, visualization, and reasoning in the logic programming environment XPCE–PROLOG has created a powerful and flexible tool. The library FNQUERY, which we are using for accessing the components of the XML representations of the source code, has also been used in other projects. For example, in [8] it is shown how mathematical knowledge in XML can be managed nicely using FNQUERY.

The predicates for analyzing dependency graphs of PROLOG programs of Section 4 can easily be adapted to JAVA programs by redefining the basic predicates `defines_fp/3` and `calls_fp/3`, and by considering classes instead of files and methods instead of predicates. In [10] we defined such predicates based on an XML representation of JAVA source code; in that paper we have also investigated some simple predicates for analyzing dependency graphs of PROLOG source code, which are different from the predicates described in Section 4 of the present paper.

In the future, we will gradually extend VISUR/RAR with additional features. We intend to implement sophisticated methods for *program analysis* from *software engineering* [3, 5, 6, 16], and

we want to integrate further *refactoring techniques* for PROLOG and for JAVA code, which have been developed in [18] for PROLOG.

References

[1] *S. Abiteboul, P. Bunemann, D. Suciu:* Data on the Web – From Relations to Semi–Structured Data and XML, Morgan Kaufmann, 2000.

[2] *S. Ceri, G. Gottlob, L. Tanca:* Logic Programming and Databases, Springer, 1990.

[3] *S. Diehl (Ed.):* Software Visualization: International Seminar, Dagstuhl Castle, Germany, Springer LNCS 2269, 2002.

[4] *T. Eiter, N. Leone, C. Mateis, G. Pfeifer, F. Scarcello:* A Deductive System for Non–Monotonic Reasoning, Proc. Fourth Intl. Conf. on Logic Programming and Non–Monotonic Reasoning LPNMR 1997, Springer LNAI 1265, 1997.

[5] *H. Erdogmus, O. Tanir (Eds.):* Advances in Software Engineering - Comprehension, Evaluation, and Evolution, Springer, 2002.

[6] *M. Fowler:* Refactoring – Improving the Design of Existing Code, Addison–Wesley, 1999.

[7] *M. Hanus, J. Koj:* An Integrated Development Environment for Declarative Multi–Paradigm Programming, Proc. Workshop on Logic Programming Environments WLPE 2001.

[8] *B. Heumesser, D. Seipel, U. Güntzer:* Flexible Processing of XML–Based Mathematical Knowledge in a PROLOG Environment, Proc. Intl. Conf. on Mathematical Knowledge Management MKM 2003, Springer LNCS 2594.

[9] *M. Hopfner, D. Seipel:* Reasoning about Rules in Deductive Databases, Proc. 17th Workshop on Logic Programming WLP 2002.

[10] *M. Hopfner, D. Seipel, J. Wolff von Gudenberg:* Comprehending and Visualising Software based on XML Representations and Call Graphs, Proc. 11th IEEE International Workshop on Program Comprehension IWPC 2003.

[11] *R. Holt, A. Winter, A. Schürr:* GXL: Towards a Standard Exchange Format, Proc. Working Conference on Reverse Engineering WCRE 2000, http://www.gupro.de/GXL/

[12] *IBM:* The Integrated Development Environment ECLIPSE, http://www.eclipse.org/

[13] *M. Kifer, G. Lausen:* F–Logic: A Higher–Order Language for Reasoning about Objects, Proc. ACM SIGMOD Conference on Management of Data, 1989.

[14] *J. Schimpf:* Logical Loops, Proc. Intl. Conference on Logic Programming ICLP 2002.

[15] *A. Serebrenik, B. Demoen:* Refactoring Logic Programs, Proc. Intl. Conference on Logic Programming ICLP 2003 (Poster Session).

[16] *J. Seemann, J. Wolff von Gudenberg:* Pattern–Based Design Recovery of JAVA Software, Proc. Intl. Symposium on the Foundations of Software Engineering 1998.

[17] *D. Seipel:* Processing XML Documents in PROLOG, Proc. 17th Workshop on Logic Programming WLP 2002.

[18] *R. Seyerlein:* Refactoring in deduktiven Datenbanken am Beispiel des Informationssystems Qualimed, Diploma Thesis, University of Würzburg, 2001.

[19] *G. Wagner:* How to Design a General Rule Markup Language, Proc. Intl. Workshop on XML Technologies for the Semantic Web, XSW 2002.

[20] *J. Wielemaker:* SWI–PROLOG 5.0 Reference Manual, http://www.swi-prolog.org/
J. Wielemaker, A. Anjewierden: Programming in XPCE/PROLOG http://www.swi-prolog.org/

Appendix

We are representing PROLOG programs in XML according to the following DTD, which is suitable for handling disjunctive logic programs with arbitrarily many head atoms:

```
<!ELEMENT program (definition*)>
<!ELEMENT definition (rule*)>
<!ELEMENT rule (head, body)>
<!ELEMENT head (atom*)>
<!ELEMENT body (atom*)>
<!ELEMENT atom ((term|var)*)>
<!ELEMENT term (term*)>

<!ATTLIST definition predicate CDATA #required>
<!ATTLIST rule file CDATA #required>
<!ATTLIST atom predicate CDATA #required>
<!ATTLIST term functor CDATA #implied>
<!ATTLIST var name CDATA #implied>
```

The arguments of an atom are either terms or variables; constants are represented as terms without subterms, where the constant is stored in the attribute `functor`.

Demonstration: Debugging Constraint problems with Portable Tools*

Pierre Deransart[1], Ludovic Langevine[1], and Mireille Ducassé[2]

[1] INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France
{Pierre.Deransart, Ludovic.Langevine}@inria.fr
[2] IRISA/INSA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
Mireille.Ducasse@irisa.fr

Presentation of the Demonstration

Many debugging tools for finite domain solvers have been developed in order to debug applications based on finite domain constraints resolution, for example, Grace [2], the Oz Explorer [3], the Oz Constraint Investigator [4], the CHIP global constraint visualizer [5], the S-box model [6], the Christmas Tree visualizer [7] and more DiSCiPl visualization tools [8].

In spite of their very interesting functionalities these tools have a major drawback, they have a low degree of portability: they are implemented on a specific platform and the work to port them from one platform to another may require a tremendous effort. From an industrial point of view, one would like to limit the efforts to port the tools when the same company develops applications with different solver platforms. From a research point of view, one would like to experiment with different tools on different platforms without too much re-engineering effort.

The previous drawback is all the most daunting that most of the tools are general purpose oriented, i.e. they do not depend on a particular application and they use quite the same kind of basic information collected in the course of the program execution. From this information, more or less complex views are generated. In the case of applications, specific animated views are frequently proposed, but their animation itself depends on the same basic information.

It is one of the objectives of the OADymPPaC project [1] to realize such a challenge: to allow several independently developed debugging tools to be used on different solvers. For this purpose an *observational semantics* which formalizes relevant aspects of constraint programming and solving has been introduced. Then a *generic trace schema* has been derived which can be used to then build tracers producing "generic traces" [9]. It is thus possible to develop independently debugging tools whose input data are built from the generic trace.

* This work is partly supported by OADymPPaC [1], a French RNTL project.

Several such tools have been independently developed by partners of the OADymPPaC project to visualize very large dynamic structures of constraint problems. In the meantime several tracers have been implemented whose trace contains the generic trace events.

We will present in the form of a demonstration the versatility of the approach to analyze, debug and solve a constraint problem. The demonstration will especially illustrate how the generic trace helps connect different visualization tools (in particular *adjacency matrices* [10] and Ilog Discovery [11]) to different platforms (in particular GNU-Prolog [12] and PaLM [13]).

References

1. OADymPPaC: Tools for dynamic analysis and debugging of constraint programs (2001) RNTL project. http://contraintes.inria.fr/OADymPPaC.
2. Meier, M.: Debugging constraint programs. In Montanari, U., Rossi, F., eds.: Proceedings of the First International Conference on Principles and Practice of Constraint Programming. Number 976 in LNCS. Springer Verlag (1995) 204–221
3. Schulte, C.: Oz Explorer: A Visual Constraint Programming Tool. In: Proceedings of the Fourteenth International Conference on Logic Programming (ICLP'97), Leuven, Belgium, The MIT Press (1997) 286–300
4. Müller, T.: Practical investigation of constraints with graph views. In: Principles and Practice of Constraint Programming – CP 2000. Number 1894 in LNCS, Springer-Verlag (2000)
5. Simonis, H., Aggoun, A., Beldiceanu, N., Bourreau, E.: Complex constraint abstraction : Global constraint visualisation. [8] chapter 12
6. Goualard, F., Benhamou, F.: Debugging Constraint Programs by Store Inspection. [8] chapter 11
7. Bracchi, C., Gefflot, C., Paulin, F.: Combining propagation information and search-tree visualization using opl studio. In Kusalik, A., Ducassé, M., Puebla, G., eds.: Proceedings of WLPE'01, Cyprus, Cyprus University (2001) 27–39
8. Deransart, P., Hermenegildo, M., Małuszyński, J., eds.: Analysis and Visualisation Tools for Constraint Programming. Number 1870 in LNCS. Springer Verlag (2000)
9. Deransart, P., Ducassé, M., Langevine, L.: A generic trace model for finite domain solvers. In O'Sullivan, B., ed.: Proceedings of User Interaction in Constraint Satisfaction (UICS'02), Cornell University (USA) (2002)
10. Ghoniem, M., Jussien, N., Fekete, J.D.: Visualizing explanations to exhibit dynamic structure in constraint problem. In O'Sullivan, B., ed.: Proceedings of Third Worshop on User-Interaction in Constraint Satisfaction, UICS'03, Lecture Notes in Computer Science, Springer-Verlag (2003) to appear.
11. Baudel, T., et al: DISCOVERY reference manual (2003) Manufactured and distributed by Ilog.
12. Langevine, L., Ducassé, M., Deransart, P.: A Propagation Tracer for GNU-Prolog: from Formal Definition to Efficient Implementation. In Palamidessi, C., ed.: Proceedings of the 19th International Conference on Logic Programming, ICLP'03, Mumbai, India (2003)
13. Jussien, N., Barichard, V.: The PaLM system: explanation-based constraint programming. In: Proceedings of TRICS: Techniques for Implementing Constraint programming Systems, a post-conference workshop of CP 2000, Singapore (2000) 118–133

Proving Termination One Loop at a Time

Michael Codish[1] and Samir Genaim[2]

[1] The Department of Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, Israel
mcodish@cs.bgu.ac.il

[2] Dipartimento di Informatica
Università degli Studi di Verona
Verona, Italy
genaim@sci.univr.it

Abstract. Classic techniques for proving termination require the identification of a measure that maps program states to the elements of a well-founded domain. Termination is guaranteed if this measure is shown to decrease with each iteration of a loop in the program. This is a *global* termination condition — there exists a measure which is shown to decrease over all of the loops in the program. In recent years, systems based on *local* termination conditions are emerging. Here, termination is guaranteed if for every loop there exists a measure which decreases as execution follows through that loop. In this paper we question the relationship between the two approaches. Reasoning locally is more convenient. But is the local approach really more powerful? We show that for a large class of termination problems the two approaches are equally powerful. To this end we demonstrate that given local conditions which support a proof of termination, a corresponding global condition can always be constructed. On the one hand, the local conditions are simpler and easier to find. Yet on the other hand, in the local approach one must consider a closure operation on loops which may require to consider an exponential number of local conditions.

1 Introduction

Classic techniques for proving termination require the identification of a measure that maps program states to the elements of a well-founded domain. Termination is guaranteed if this measure is shown to decrease with each iteration of a loop in the program. Traditionally, the termination condition is *global* and loops are characterized *syntactically*. Global, because there exists a single measure which is shown to decrease over all of the loops in the program; and syntactic, because loops are defined in terms of the simple cycles in a graph representing, or derived from, the program. Namely, a path with no repeated vertices, except for the initial and terminal vertex.

In recent years, systems based on *local* termination conditions and which characterize loops *semantically* are emerging. Local, because termination is guaranteed if for every loop there exists a (possibly different) measure which decreases as execution follows through that loop; and semantic, because loops are characterized in terms of program executions. In this approach a loop corresponds to a pair of "program states" at which execution visits the same "program point".

This paper investigates the relation between the global and local approaches to proving termination. In theory, if there exist local measures to determine termination then there must exist also a global measure. This follows from the correctness of the local approach and from the completeness of the global approach. On the other hand, in practice, analyzers based on the local approach can often prove termination for programs for which analyzers based on global functions cannot. The question posed is how serious is the rift between the two approaches?

We show that for termination analyzers based on the *size change termination* principle [] or equivalently, on the description of loops in terms of *monotonicity constraints* [], the two approaches are of equal power. To this end we illustrate how to construct a global termination measure from a given finite set of local measures.

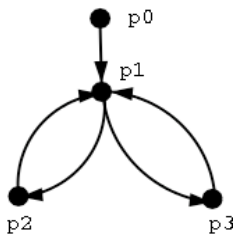
We also show that the construction does not work when loops are described in a richer language. Hence it is not clear, for the general case what is the relation between local and global termination analysis.

In this paper we consider a general setting independently of any particular programming language or paradigm. We assume some syntactic notion of a *program point* and some semantic notion of a *program state*. A program point is a node in the parse tree and represents the point just before (or after) the execution of a program statement. A program state associates values for program variables at a given program point and represents a run-time situation when execution is at the given point. A computation (or program execution) is a, possibly infinite, sequence of program states.

To clarify the terminology and to highlight the difference between the two characterizations of loops consider the following example.

Example 1. Consider the following program fragment, on the left, where p_0, p_1, p_2 and p_3 are program points. The graph, given on the right, highlights the flow of control between the program points. The simple cycles in the graph are $\langle p_1, p_2, p_1 \rangle$ and $\langle p_1, p_3, p_1 \rangle$. These are the syntactic loops in the program.

```
p0: int x = 1, y = 1;
p1: while (x + y > 0)
    {
      if (x > y)
p2:     { x := x - 1; y := y + 2; }
      else
p3:     { y := y - 1; x := x + 2; }
    }
```



For the program above, execution iterates returning to point p_1. The pair of program states $p_1(1,1) \rightsquigarrow p_1(3,0)$ represents an execution loop because there is a visit to p_1 with $x = 1, y = 1$ and a subsequent visit with $x = 3, y = 0$. Similarly, $p_1(3,0) \rightsquigarrow p_1(2,2)$ and $p_1(1,1) \rightsquigarrow p_1(2,2)$ are also execution loops. The set of all execution loops for the given program is infinite.

In the semantic-based approach to termination analysis, abstract interpretation is applied to provide a finite approximation of a program's (semantic) loops. A concrete semantics that is the basis for the termination analysis of logic programs is given in [4]. In this semantics the semantic objects are pairs of initial and terminal program states in a computation. The special case where the initial and terminal states are associated with the same program point indicates a semantic loop. In the corresponding abstract semantics, loops are described by *abstract binary clauses* of the form $p(\bar{x}) \leftarrow \pi, p(\bar{y})$ where $p(\bar{x})$ and $p(\bar{y})$ represent the program states at point p before and after executing the loop, and π is a constraint describing relations between the sizes of values of the variables of the states.

Termination analyzers based on abstract interpretation differ depending on the type of constraints allowed to appear in π. In [8] the authors use monotonicity constraints which are binary relations on variables. An equivalent approach is presented in [7] where the authors represent constraints using, so called, size change graphs. In such a graph, nodes correspond to the sizes of the values in the program states before and after execution of the loop, and labeled edges to binary relations on variables.

Example 2. Consider the procedure (on the left) defining the Ackerman function where four program points ($acker_0, \ldots, acker_3$) are indicated using subscripts.

```
int acker0(int m, int n)
{  /* restricted for m,n >= 0 */
  if (m == 0)
    return n+1;
  else if (n == 0)
    return acker1(m-1,1);
  else
    return acker3(m-1,acker2(m,n-1));
}
```

$\ell_1 =$ `acker`$_0(x_1, x_2) \leftarrow$
$x_1 \geq 0, x_2 \geq 0, y_1 \geq 0, y_2 \geq 0,$
$x_1 = y_1, x_2 > y_2,$ `acker`$_0(y_1, y_2)$.

$\ell_2 =$ `acker`$_0(x_1, x_2) \leftarrow$
$x_1 \geq 0, x_2 \geq 0, y_1 \geq 0, y_2 \geq 0,$
$x_1 > y_1,$ `acker`$_0(y_1, y_2)$.

Each recursive call $acker_i$, with $i > 0$ gives rise to a loop when execution returns to $acker_0$. The two abstract binary clauses (on the right) approximate the infinite number of concrete execution loops at program point $acker_0$ (for arbitrary non-negative initial values of m and n). The first abstract binary clause represents concrete loops in which the size of the first argument is invariant and the size of the second argument decreases. The second abstract binary clause represents concrete loops in which the size of the first argument is decreasing.

The following example illustrates the global and local approaches to proving termination. The formal justification of these are discussed in the next section.

Example 3. The measures $f_1(acker_0(x,y)) = y$ and $f_2(acker_0(x,y)) = x$ from Example 2 decrease respectively for the loop descriptions ℓ_1 and ℓ_2 in the example. These are local measures, one for each of the two loops. This provides a proof of termination by the local approach. A global measure can also be identified. Consider the function $f(acker_0(x,y)) = \langle f_2(acker_0(x,y)), f_1(acker_0(x,y))\rangle$ which decreases for both loops with respect to the lexicographic ordering on pairs. This provides a proof of termination by the global approach. □

Note that the local approach to proving termination is not necessarily correct when loops are represented in the more classic syntactic approach. At least not if we restrict attention only to the simple cycles in the graph.

Example 4. Consider the graph representation of the program from Example 1. The measures $f_1(p(x,y)) = x$ and $f_2(p(x,y)) = y$ decrease respectively for the two cycles in the graph. However the program does not terminate. A proof of termination in the local approach would have to consider also the non-simple loop $p_1p_2p_1p_3p_1$ for which neither x nor y decreases.. □

Reasoning locally about loops does have advantages. Local termination measures are often simpler. They are easier to identify, by the human user as well as when aiming for automated proofs. For example, there are many cases where proofs based on local measures involve linear functions while corresponding global measures involve tuples ordered lexicographically. As a consequence, analyzers implemented to use linear techniques are more successful when based on local instead of global termination conditions.

2 Preliminaries

This section presents the local approach to proving termination, recalls the classic justification for the global approach and explains why the justification of the local approach is problematic.

To simplify the presentation, we assume that the (syntactic) loops of a program P are given as abstract binary clauses, for example a program point p (where the execution iterates) is described using $p(\bar{x}) \leftarrow \pi, p(\bar{y})$, where $\bar{x}$ is the values of the program's variables (a state) when entering the loop, $\bar{y}$ is the state when revisiting the loop, and π is the size-relations between the two states. Formally speaking, (abstract) binary clauses are defined as follows.

Definition 1 (Abstract binary clauses). *An abstract binary clause C is a clause of the form $p(\bar{x}) \leftarrow \pi, p(\bar{y})$ where $\bar{x}$ and $\bar{y}$ consists of distinct tuples of variables and π are linear constrains over a well-founded domain $D = \langle D, <\rangle$ that involve only the variables of $\bar{x}$ and $\bar{y}$ only. Moreover, we assume that π contains the implicit constraint $\{v \geq 0 | v \in vars(\bar{x}) \cup vars(\bar{y})\}$.*

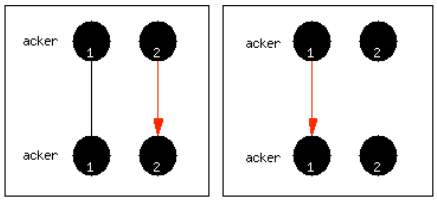
Often, the constraint domain D that used to describe a binary clause, is further restricted to contain only *monotonicity constraints* [8]. These are conjunctions of binary relations $(<, \leq, =)$ between pairs of variables. This restriction,

and in general the choice of D, is sometimes crucial to proving termination as show in the following example.

Example 5. Consider the following binary clause $p(x_1, x_2) \leftarrow \pi, p(y_1, y_2)$ where $\pi \equiv [y_1 = 2 * x_2 + 1 \wedge x_1 = 3 * y_2 + 2]$ and assume it is the only loop in the program. It is easy to verify that $f(a_1, a_2) = a_1 + 3 * a_2$ is decreasing for this binary clause, i.e $\pi \models f(y_1, y_2) > f(x_1, x_2)$, hence the program terminates. But, restricting π to *monotonicity constraints* results in $\pi' \equiv [y_1 > x_2, x_1 < y_2]$, and there is no f such that $[y_1 > x_2, x_1 < y_2] \models f(y_1, y_2) > f(x_1, x_2)$. Therefore, with this restriction we cannot prove termination.

Loop descriptions involving monotonicity constraints can also be represented as size change graphs [7] where directed edge from x to y indicates a constraint $x > y$, and non-directed edge indicates a constraint $x \geq y$.

Example 6. Consider the loop descriptions from Example 2. These will be depicted by the size change graphs.



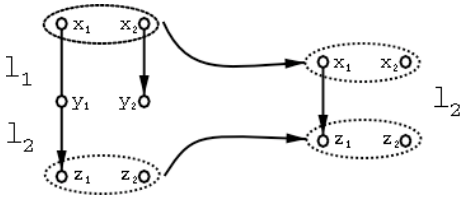
The vertices on the top of the graph correspond to those in the head of the binary clause and those on the bottom to the variables in the body of the clause.

The correctness of the local approach to proving termination relies on the fact that implicit loops are made explicit first, only then we can claim termination if we find (possibly different) decreasing measures for each of these loops. Making the implicit loops explicit is done by closing the set of (syntactic) binary clauses under compositions, where the composition of two loops is defined as follows.

Definition 2 (Composing loops). *Let $\ell_1 = p(\bar{x}) \leftarrow \pi_1, p(\bar{y})$ and $\ell_2 = p(\bar{y}) \leftarrow \pi_2, p(\bar{z})$ be (renaming of) programs loops descriptions (which do not share variables except those in $\bar{y}$). The composition of ℓ_1 and ℓ_2 is defined as $\ell_1 \circ \ell_2 = p(\bar{x}) \leftarrow \exists \bar{y}.\ \pi_1 \wedge \pi_2, p(\bar{z})$.*

The following example illustrates the composition of two loops. Note the projection of the variables (by $\exists \bar{y}$).

Example 7 (Composing loops). Consider the loop descriptions ℓ_1 and ℓ_2 from Example 2. Observe that $\ell_1 \circ \ell_1 = \ell_1$, $\ell_1 \circ \ell_2 = \ell_2 \circ \ell_1 = \ell_2$, and $\ell_2 \circ \ell_2 = \ell_2$. For example assuming that $\ell_1 = p(x_1, x_2) \leftarrow [x_1 = y_2, x_2 > y_2], p(y_1, y_2)$ and $\ell_2 = p(y_1, y_2) \leftarrow [y_1 > z_1], p(z_1, z_2)$, then the composition $\ell_1 \circ \ell_2$ results in $\ell_1 = p(x_1, x_2) \leftarrow [x_1 > z_1], p(z_1, z_2)$ (which is a variant of ℓ_2) as illustrated in the following diagram:



It is important to note that closing a set of loop descriptions under composition can introduce an exponential number of additional descriptions [7].

The following proposition provides conditions when we can claim termination based on the local approach.

Proposition 1. *Let P be a logic program, p a program point at which the execution can loop, and $\mathcal{L}_p$ a closed set (under composition) of binary clauses constructed from the syntactic loops of p, then: If for each $\ell \equiv p(\bar{x}) \leftarrow \pi, p(\bar{y}) \in \mathcal{L}_p$ there exists a function f_ℓ which maps states to a well-founded domain $D = (D, <)$ such that $\pi \models f(p(\bar{y})) < f(p(\bar{x}))$ then p terminates. The set of all such functions f_ℓ is denoted by $\mathcal{C}_p$.*

Before discussing the formal justification of the local approach, let us recall that for the more classic, global approach. The global approach is the special case where in Proposition 1 the set $\mathcal{C}_p$ consists of a single function f which decreases for all of the loop descriptions associated with program point p. The proof of correctness for this case is classic. In brief, consider that if there were to exist an infinite computation for P then there would be a program point p visited infinitely often in that computation. Denote the corresponding sequence of visits at p as

$$p^1 \rightsquigarrow p^2 \rightsquigarrow \cdots \rightsquigarrow p^k \rightsquigarrow \cdots$$

Each pair $p^i \rightsquigarrow p^{i+1}$ is a concrete loop described by one of the loop descriptions for p and the function f decreases for that pair. But f maps states to a well founded domain and cannot decrease infinitely often. Hence there cannot exist an infinite computation.

For the local approach this argument no longer holds as pairs of states $p^i \rightsquigarrow p^{i+1}$ may be associated with different functions and the pairs for which one function decreases may cause another to increase.

In some cases it is easy to work around this type of problem, for example when there is some regularity in the loops describing the corresponding transitions from program point p to program point p. For instance, if we consider a computation in which the concrete loops are described by ℓ_1 and ℓ_2 alternating as illustrated (the transitions are labeled by the corresponding loop description):

$$p \overset{\ell_1}{\rightsquigarrow} p \overset{\ell_2}{\rightsquigarrow} p \overset{\ell_1}{\rightsquigarrow} p \overset{\ell_2}{\rightsquigarrow} p \overset{\ell_1}{\rightsquigarrow} p \overset{\ell_2}{\rightsquigarrow} p \overset{\ell_1}{\rightsquigarrow} p \overset{\ell_2}{\rightsquigarrow} \cdots$$

then we also have an infinite computation of the form

$$p \overset{\ell_1 \circ \ell_2}{\rightsquigarrow} p \overset{\ell_1 \circ \ell_2}{\rightsquigarrow} p \overset{\ell_1 \circ \ell_2}{\rightsquigarrow} p \overset{\ell_1 \circ \ell_2}{\rightsquigarrow} \cdots$$

and there is a function f associated with the loop $\ell_1 \circ \ell_2$ which cannot decrease infinitely often.

But in general one cannot expect to always find such regularity, as for example illustrated by the infinite computation of the form

$$p \overset{\ell_3}{\rightsquigarrow} p \overset{\ell_1}{\rightsquigarrow} p \overset{\ell_4}{\rightsquigarrow} p \overset{\ell_1}{\rightsquigarrow} p \overset{\ell_5}{\rightsquigarrow} p \overset{\ell_9}{\rightsquigarrow} p \overset{\ell_2}{\rightsquigarrow} p \overset{\ell_6}{\rightsquigarrow} \cdots$$

where the loop descriptions correspond to the digits of the constant Π.

A formal justification of the local approach to proving termination is presented in [5]. The proof is based on an application of Ramsey's theorem [9]. We present here the proof of correctness for termination reasoning about individual loops. The proof is set as an application of Ramsey's Theorem [9]. It is essentially the same as that presented by Dershowitz *et al.*DLSS01 for logic programs in [5].

Theorem 1. *Let P be a program, and suppose we proved termination of P using the local approach as described in Proposition 1. Then P terminates.*

The proof is by contradiction as an application of Ramsey's Theorem [9].

Ramsey's Theorem: Let $\mathcal{A} = \left\{ \langle a, b\rangle \,\middle|\, a, b \in \mathbb{N} \text{ and } a < b \right\}$, $\mathcal{C}$ be a finite set of colors and let $\mathcal{F} : \mathcal{A} \mapsto \mathcal{C}$ be a function associating the elements of $\mathcal{A}$ with colors from $\mathcal{C}$. Then, there is a color $f \in \mathcal{C}$ and an infinite set $\mathcal{X} \subseteq \mathbb{N}$ such that $\mathcal{F}(\langle a, b\rangle) = f$ for each $a, b \in \mathcal{X}$ for which $a < b$.

Proof. (of Theorem 1)
Assume in contradiction that P has an infinite computation. This implies that there is a program point p which is visited infinitely often. Denote the corresponding sequence of visits at p as

$$\mathcal{S} = \quad p^1 \rightsquigarrow p^2 \rightsquigarrow \cdots \rightsquigarrow p^k \rightsquigarrow \cdots$$

To apply Ramsey's Theorem we should fix a finite set of colors $\mathcal{C}$ and a mapping $\mathcal{F}$ of elements of $\mathcal{A} = \left\{ \langle a, b\rangle \,\middle|\, a, b \in \mathbb{N} \text{ and } a < b \right\}$ to colors. We take as colors the set of functions $\mathcal{C}_p$ prescribed by Proposition 1. To fix the mapping $\mathcal{F}$, we note that each pair of visits p^a and p^b with $a < b$ denotes a (semantic) loop at p and is hence associated with a corresponding loop description and a function $f_i \in \mathcal{C}_p$. Ramsey's Theorem implies the existence of an infinite subsequence of $\mathcal{S}$ of the form

$$\mathcal{S}' = \quad p^{i_1} \rightsquigarrow p^{i_2} \rightsquigarrow \cdots \rightsquigarrow p^{i_k} \rightsquigarrow \cdots$$

for $\mathcal{X} = \{i_1, i_2, \ldots, i_k, \ldots\}$ such that the loop corresponding to any pair of visits p^{i_a} and p^{i_b} with $a < b$ decreases with respect to the same function $f \in \mathcal{C}_p$. But since f maps states to a well-founded domain, this is a contradiction. □

3 When Local Meets Global

In this section we illustrate that for termination analysis based on monotonicity constraints (or equivalently, on the size change principle), the local and global approaches are of equivalent power. One direction of the argument is straightforward, as a global measure for termination is also a local measure. To show the other direction, we show that if we can prove termination in the local approach then we can also construct a global termination condition. This provides an alternative justification of the local approach, for the given class of case of monotonicity constraints.

Finding local functions for termination is easy. In [8] the authors show that if adding the "up arrows" in the graphs introduces a cycle (with at least one strict decrease then there exists a function. in [4], the authors add to each π the constraints $x_i < y_i$ and check that the resulting constraint is inconsistent, and in [7] the authors prove that it is sufficient to find a single down arrow in all size change graphs that are idempotent (in the closure). The question is how to find a suitable global function.

Example 8. Consider the three loop descriptions for $i \in \{1, 2, 3\}$

$$\ell_i \equiv p(x_1, x_2, x_3) \leftarrow \pi_i,\ \ p(y_1, y_2, y_3)$$

with $\pi_1 = \ \ x_1 > y_1, x_2 > y1$, $\pi_2 = \ \ x_1 \geq y2, x_2 \geq y_2, x_3 > y_3$ and $\pi_3 = \ x_1 \geq y2, x_2 > y_2$ where all of the variables are assumed to be non-negative. In the local approach a proof of termination is obtained choosing the following functions corresponding to the three loops: $f_1(p(x, y, z)) = x$, $f_2(p(x, y, z)) = z$ and $f_3(p(x, y, z)) = y$. In the global approach a proof is obtained choosing a global function $f(p(x, y, z)) = \langle min(x, y), y, z\rangle$ with the standard lexicographic ordering.

Definition 3 (Monotonic functions for a set of loops). *Let $\mathcal{L}_p$ be a finite set of loop descriptions with corresponding functions $\mathcal{C}_p$ mapping states to well-founded domains such that for each $\ell = \ p(\bar{x}) \leftarrow \pi, p(\bar{y}) \in \mathcal{L}_p$ and corresponding function $f_\ell : State \rightarrow D$ it holds that $\pi \models f(p(\bar{y})) < f(p(\bar{x}))$. We say that $\mathcal{C}_p$ is* monotonic *for $\mathcal{L}$ if for any composition $\ell = \ell_1 \circ \cdots \circ \ell_k$ of (a subset of) loops from $\mathcal{L}_p$, the function f_ℓ associated with the loop ℓ satisfies: (1) $\forall 1 \leq i \leq k.\pi_i \models f_\ell(p(\bar{x})) \geq f_\ell(p(\bar{y}))$; and (2) $\exists 1 \leq i \leq k.\pi_i \models f_\ell(p(\bar{x})) > f_\ell(p(\bar{y}))$.*

□

Example 9. Consider again the three loops from Example 8.

1. The set of functions $f_1(p(x, y, z)) = x$, $f_2(p(x, y, z)) = z$ and $f_3(p(x, y, z)) = y$ is note monotonic for these three loops. Observe that $\ell_3 \circ \ell_2 \circ \ell_1 = \ell_1$ and f_1 does not decrease (even weakly) for ℓ_2.
2. The single function $f1(p(x, y, z)) = f_2(p(x, y, z)) = f_3(p(x, y, z)) = \langle min(x, y), y, z\rangle$ is monotonic for the three loops.

Theorem 2. *Let $\mathcal{C}_p$ be a monotonic set of functions for loop descriptions $\mathcal{L}$. Then there exists an ordered tuple $\langle f_1, \ldots, f_k \rangle$ of functions from $\mathcal{C}_p$ such that the function f defined by $f(s) = \langle f_1(s), \ldots, f_k(s) \rangle$ decreases globally for all of the loop descriptions in $\mathcal{L}_p$ with respect to the corresponding lexicographic order. Moreover, this ordered tuple can be constructed with time complexity $\mathcal{O}(|\mathcal{L}_p|^2)$.* □

The algorithm presented in Figure 1 constructs the required tuple of functions from $\mathcal{C}_p$ and provides the proof. The basic idea is that if we compose a *monotonic* set of local loop descriptions from $\mathcal{L}_p$ then we identify a function from $\mathcal{C}_p$ which decreases when passing through at least one of the composed loops but never increases when going through these loops. Hence this function may be placed to the right of these functions in the tuple we are constructing.

```
(0) S = L_p and Π = ∅
(1) WHILE ( S ≠ ∅ ) DO
(2)    ℓ = compose_loops(S) \* compose in any order *\
(3)    IF ( ℓ ∈ S ) THEN
(4)         Π = Π.f_ℓ
(5)         S = S \ {ℓ}
(6)    ELSE
(7)         S = { p(x̄) ← π, p(ȳ) ∈ S | π ⊭ f_ℓ(p(x̄)) > f_ℓ(p(ȳ)) }
(8)          \*  remove from S all loops that decrease for  f_ℓ  *\
(9) DONE
```

Fig. 1. Constructing a global decreasing measure

The algorithm proceeds as follows: We start (at line 0) with $S = \mathcal{L}_p$ and an empty tuple Π. At each iteration of the while loop (lines 1-9) we compose the loops of S (line 2). If the result ℓ is an element of S then we add it to the right in the tuple Π (line 4) and remove it from S (line 5). Otherwise, if $\ell \notin S$, then (since S is monotonic) there must be at least one loop $\ell' \equiv p(\bar{x}) \leftarrow \pi, p(\bar{y}) \in S$ such that $\pi \models f_\ell(p(\bar{x})) > f_\ell(p(\bar{y}))$. This means whenever traversing a loop described by ℓ', both of the functions $f_{\ell'}$ and f_ℓ decrease. Hence, we can remove any such ℓ' from S (lines 7-8).

The correctness of the algorithm stems from the following: (1) At each step we remove at least one loop from S so the algorithm terminates; (2) The fact that S is monotonic together with lines 3-5 guarantees that whenever a function f_i (in the tuple) decreases, any function to its left either decreases or remains invariant; and (3) lines 7-8 guarantees that for each loop description there is at least one corresponding function in the tuple.

Since we remove at least one loop at each iteration, the complexity of the algorithm is $\mathcal{O}(|\mathcal{L}_p|^2)$ – we count the number of times we compose two loops (the basic step of *compose_loops*) plus the operations on sets at lines 4, 7 and 8.

Example 10. Consider the following abstract loop descriptions (where all of the variables are assumed to be non-negative).

$$\begin{array}{ll} \ell_1 = & p(x_1,x_2,x_3) \leftarrow x_1 > y_1, x_2 > y_2, p(y_1,y_2,y_3). \\ \ell_2 = & p(x_1,x_2,x_3) \leftarrow x_1 = y_1, x_3 > y_3, p(y_1,y_2,y_3). \\ \ell_3 = & p(x_1,x_2,x_3) \leftarrow x_1 > y_1, p(y_1,y_2,y_3). \end{array}$$

As local functions associated with ℓ_1, ℓ_2 and ℓ_3 we take $f_1(p(x,y,z)) = x$, $f_2(p(x,y,z)) = z$, $f_3(p(x,y,z)) = x$. The algorithm starts with $S = \{\ell_1, \ell_2, \ell_3\}$ and proceeds as follows:

1. $\ell_1 \circ \ell_2 \circ \ell_3 = \ell_3$, so we take f_3 as the first function in the tuple and remove ℓ_3 from S.
2. $\ell_1 \circ \ell_2 = \ell_3$ and ℓ_3 is not in S. We see that f_3 decreases on ℓ_1 (as well as on ℓ_3), so we remove ℓ_1 from S.
3. ℓ_2 is the only remaining loop in S so we take f_2 as the next function in the tuple and remove ℓ_2 from S.

The function $f(s) = \langle f_3(s), f_2(s) \rangle$ decreases (globally) for all of the loops in the original description. □

It still remains unclear how to find a suitable set of functions which is monotonic for the given loops. For the special case where all binary constraints are between the same arguments (x_i with y_i), the task is more simple.

Lemma 1. *If all of the constraints occurring in the loop descriptions $\mathcal{L}_p$ are monotonicity constraints of the form $x_i < y_i$ or $x_i \leq y_i$ (namely between the same argument positions) then $\mathcal{L}_p$ is a monotonic set of loops.* □

To extend the result for the full class of monotonicity constraints we need to consider all local functions of the form $min_{i \in I}(x_i)$, $max_{i \in I}(x_i)$, $sum_{i \in I}(x_i)$ for I a subset of the arguments in the state. In addition, we do not want to fix from the start a set of functions which is monotonic for the full set of loop descriptions, but rather at each stage in the algorithm of Figure 1 we need a set of functions monotonic for the loops in the set S (those which still remain in the game). Finally, consider the i^{th} iteration of the algorithm and assume that the set of functions F have already been introduced to the tuple Π and that the loop descriptions left are in S. Hence, by the construction the functions in F remain invariant for the loops in S (otherwise they would have been removed in a previous stage of the algorithm). We may take this into account when looking for the monotonic set of functions in the next iteration for S.

4 The General Case

Theorem 2 does not hold for the general case for example, when loops are described are expressed using linear constraints (but not necessarily monotonicity constraints). We illustrate this by example.

In the following example, loop descriptions are given in a richer domain allowing linear constraints. In this case it is not obvious to detect a global function with which to prove termination. Existing analyzers based on the global approach do not succeed to provide the proof.

Example 11. Consider the following three loop descriptions where the variables x_i, y_i, z_i are assumed to be non-negative:

$$\begin{aligned}\ell_1 \equiv\ & p(x_1,y_1,z_1,k_1,l_1) \leftarrow \\ & \quad [k_1 = k_2, l_1 = l_2, x_1 > x_2, k_1 = 0, l_1 = 0]\,, \\ & \quad p(x_2,y_2,z_2,k_2,l_2)\,.\end{aligned}$$

$$\begin{aligned}\ell_2 \equiv\ & p(x_1,y_1,z_1,k_1,l_1) \leftarrow \\ & \quad [k_1 = k_2, l_1 = l_2, y_1 > y_2, x_1 > x_2 + k_1, k_1 + l_1 = 0]\,, \\ & \quad p(x_2,y_2,z_2,k_2,l_2)\,.\end{aligned}$$

$$\begin{aligned}\ell_3 \equiv\ & p(x_1,y_1,z_1,k_1,l_1) \leftarrow \\ & \quad [k_1 = k_2, l_1 = l_2, z_1 > z_2, x_1 = x_2 + l_1, k_1 = l_1]\,, \\ & \quad p(x_2,y_2,z_2,k_2,l_2)\,.\end{aligned}$$

With the local approach we can prove termination taking $f_{\ell_1}(p(x,y,z,k,k)) = x$, $f_{\ell_2}(p(x,y,z,k,l)) = y$ and $f_{\ell_3}(p(x,y,z,k,l)) = z$. Composing any two (different) loops results in ℓ_1. So, the set of abstract binary clauses is closed under composition. But there is no permutation $\langle f_{i_1}, f_{i_2}, f_{i_3}\rangle$ such that $f(s) = \langle f_{i_1}(s), f_{i_2}(s), f_{i_3}(s)\rangle$ is guaranteed to decrease globally on all three loops. □

5 Conclusion

This paper poses a question concerning the relation between the global and local approaches to proving termination. We show that in many cases we can combine local termination conditions into a tuple of functions ordered such that the tuple function decreases globally with respect to the lexicographic ordering.

While the results of the paper strengthen our belief that reasoning with local conditions has clear practical advantages, still, we lack the hard evidence that the local approach is more powerful than the global one or not. This is the effort of our ongoing research.

The very elegant result of Theorem 1 is not well known and is due to Dershowitz *et al.* [5]. It has subsequently been worked out independently by others working on the analysis of termination using local conditions as described in [3], [1] and [6].

Ben-Amram in [2] presents a transformation so that any program P for which the local approach works based on monotonicity constraints (or the size change principle) is transformed to a semantically equivalent program P^* with the same termination behavior for which termination can be identified in terms of lexicographic descent. This is the case handled by the special case of Lemma 1.

References

1. Maurice Bruynooghe, Michael Codish, John Gallagher, Samir Genaim, and Wim Vanhoof. Termination analysis through combination of type based norms, May 2003.
2. Amir M. Ben-Amram. General size-change termination and lexicographic descent. In Torben Mogensen, David Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 3–17. Springer-Verlag, 2002.
3. Michael Codish, Samir Genaim, Maurice Bruynooghe, John Gallagher, and Wim Vanhoof. One loop at a time. In *6th International Workshop on Termination (WST 2003)*, June 2003.
4. Michael Codish and Cohavit Taboch. A semantic basis for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
5. N. Dershowitz and N. Lindenstrauss and Y. Sagiv and A. Serebrenik. A General Framework for Automatic Termination Analysis of Logic Programs. *Applicable Algebra in Engineering, Communication and Computing*, Springer-Verlag Heidelberg, 12(1-2):117–156, 2001.
6. Chin Soon Lee. *Program Termination Analysis and Termination of Offline Partial Evaluation*. PhD thesis, University of Western Australia, August 2002.
7. Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36(3):81–92, 2001.
8. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. L. Naish, editor. *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77. Leuven, Belgium, 1997. The MIT Press.
9. F.P. Ramsey. On a problem of formal logic. In *Proceedings London Mathematical Society*, volume 30, pages 264–286, 1930.

Hasta-La-Vista: Termination Analyser for Logic Programs

Alexander Serebrenik, Danny De Schreye

Department of Computer Science, K.U. Leuven
Celestijnenlaan 200A, B-3001, Heverlee, Belgium
E-mail: {Alexander.Serebrenik,Danny.DeSchreye}@cs.kuleuven.ac.be

Abstract. Verifying termination is often considered as one of the most important aspects of program verification. In this paper we present Hasta-La-Vista—an automatic tool for analysing termination of logic programs. To the best of our knowledge, Hasta-La-Vista is unique in being able to prove termination of programs depending on integer computations.

1 Introduction

Proving termination is often considered as an important aspect of program verification. Logic programming languages, allowing us to program declaratively, increase the danger of non-termination. Therefore, termination analysis received considerable attention in logic programming (see, e.g., [8, 14, 27]). Unfortunately, most work on termination analysis is restricted to pure logic programs and thus many interesting real-world examples are left out of consideration. Aritmetic is a case in the point: while almost every real-world program contains a numerical part, most other works modelled it by 0 and the successor function. Therefore, in order to bridge the gap between programming practice and existing termination analysers, real-world programming techniques should be considered.

In this paper we present Hasta-La-Vista[1] — a powerful tool for analysing termination of logic programs with integer computations. While such programs are very common in real-world programming, until recently they mostly remained a *terra incognita* for the termination research community. In fact, none of the existing termination analysers we are aware of (TermiLog [22], TermiWeb [8], TALP [28], and cTI [26]) is powerful enough to prove termination even of the simplest integer computations such as the following program:

Example 1.

$$p(X) \leftarrow X < 7,\ X1 \text{ is } X + 1,\ p(X1).$$

[1] *Hasta la vista, baby!* The Terminator in [6]

Assuming the left-to-right selection rule of Prolog, this program terminates for queries $p(X)$, for all integer values of X. □

Our approach is based on transforming a program in a way that allows integrating and extending techniques originally developed for analysis of numerical computations in the framework of query-mapping pairs [15] with the well-known framework of acceptability [14].

Furthermore, our approach is not limited to proving termination, but can also infer termination. More precisely, we will be inferring conditions that, if imposed on the queries, will ensure that the queries will terminate. Inference of termination conditions was studied in [18,26]. Unlike termination conditions inferred by these approaches, stated in terms of groundness of arguments (calls to *append* terminate if either the first or the third argument is ground), our technique produces conditions based on numerical domains of the arguments as shown in Example 2. Combining the approaches to infer both kinds of conditions is considered as a future work.

Example 2. Consider the following program.

$$q(X,Y) \leftarrow X > Y,\ Z\ is\ X - Y,\ q(Z,Y).$$

Queries of the form $q(arg_1, arg_2)$, where arg_1 and arg_2 are integers, terminate with respect to this program if either $arg_1 \leq arg_2$ or $arg_1 > arg_2$ and $arg_2 > 0$. This is exactly the termination condition we will infer. □

The rest of the paper is organised as follows. In the next section we present a general overview of the system. The presentation is kept on the intuitive level, more formal results can be found in [33]. Section 3 contains detailed discussion of an experimental evaluation of the method. In Section 4 we discuss further extensions, such as proving termination of programs depending on numerical computations as well as symbolic ones. We summarise our contribution in Section 5, review related work and conclude.

2 System architecture

In this section we present an overview of the system architecture and illustrate the working of main components by analysing Example 2.

Conceptually, Hasta-La-Vista consists of three main parts: transformation, constraint generation and constraint solving. As a preliminary step, given a program and a set of atomic queries, type analysis of Janssens and Bruynooghe [20] computes the call set. We opted for a very simple type inference technique that provides us only with information whether an argument is integer or not. More

refined analysis can be used. For instance, the technique presented in [21] would have allowed us to know whether some numerical argument belongs to a certain interval. Alternatively, the integer intervals domain of Cousot and Cousot [11] might have been used. For our running example type analysis establishes that all calls that will be generated are of the form $q(int, int)$.

Based on the results of the type analysis the system approximates whether termination of the given program can be dependent on the integer computation. It should be noted that there are programs, such as *quicksort*, that use arithmetic but whose termination behaviour is not dependent on their arithmetic part. This is not the case, however, for Example 2, since the integer assignment operator (*is*) is used to produce a value in the recursive call to q.

If Hasta-La-Vista suspects that termination depends on arithmetic, the *adorning* transformation [33] is applied. The aim of the transformation is to split the domain of calls in order to allow each one of the cases to be analysed separately. In this way we discover bounded integer arguments and make the bounds explicit. Intuitively, we are interested in bounded arguments, since if, for example, a variable x is known to be bounded from above by n, then $n-x$ is always positive and thus, it can be used as a basis for a definition of a *level-mapping* (a function from the set of atoms to the naturals). Similarly, if a variable x is bounded from below by n, $x-n$ is always positive and thus, can be used as a basis for a definition of a level-mapping. To illustrate the transformation consider the following example.

Example 3. We are interested in proving termination of the set of queries $\{p(n) \mid n \text{ is an integer}\}$ with respect to the following program:

$$p(X) \leftarrow X > 1,\ X < 100,\ X1\ is\ -X^2,\ p(X1).$$
$$p(X) \leftarrow X < -1,\ X > -100,\ X1\ is\ X^2,\ p(X1).$$

Let arg_1 denote the first argument. The first clause is applicable, i.e., the clause is selected and the test is passed, if the constraint $c_1 \equiv 1 < arg_1 < 100$ holds for $p(X)$. Similarly, the second clause is applicable if $c_2 \equiv -100 < arg_1 < -1$ holds for $p(X)$. Thus, termination of $p(X)$ for $c_3 \equiv (arg_1 \leq -100 \vee -1 \leq arg_1 \leq 1 \vee arg_1 \geq 100)$ is trivial. We call c_1, c_2 and c_3 *adornments* and denote the set of adornments $\mathcal{A}_p$. In general, adornments are constructed as (disjunctions of) conjunctions of comparisons appearing in the bodies of the clauses. Formally, $\mathcal{A}_p$, computed by Hasta-La-Vista, is the set of all conjunctions $\wedge_{i=1}^{n} d_i$, where d_i is either a conjunction of comparisons appearing in the body of a clause or its negation.

It should be noted that if c_1 holds and the first clause is applied, then either c_2 or c_3 hold for the recursive call. We use this observation and specialise the

program with respect to c_1, c_2 and c_3 (cf. [39]). The following program is obtained:

$$p^{c_1}(X) \leftarrow X > 1,\ X < 100,\ X1\ is\ -X^2,\ -100 < X1,\ X1 < -1,\ p^{c_2}(X1).$$
$$p^{c_1}(X) \leftarrow X > 1,\ X < 100,\ X1\ is\ -X^2,$$
$$\quad (X1 \leq -100; (-1 \leq X1,\ X1 \leq 1); X1 \geq 100),\ p^{c_3}(X1).$$
$$p^{c_2}(X) \leftarrow X < -1,\ X > -100,\ X1\ is X^2,\ 1 < X1,\ X1 < 100,\ p^{c_1}(X1).$$
$$p^{c_2}(X) \leftarrow X < -1,\ X > -100,\ X1\ is X^2,$$
$$\quad (X1 \leq -100; (-1 \leq X1,\ X1 \leq 1); X1 \geq 100),\ p^{c_3}(X1).$$

Observe that in our example there are no clauses defining p^{c_3} in the specialised program since c_3 is not consistent with tests in the clauses.

For the specialised program the following holds. In case c_1, arg_1 is bounded by 1 and 100, in case c_2, it is bounded by -100 and -1. This information is essential for proving termination. □

Similarly, Hasta-La-Vista discovers that there are two adornments relevant for q in Example 2: $arg_1 \leq arg_2$ and $arg_1 > arg_2$. In the former case the program clause is not applicable and termination is trivially established. In the latter case termination has to be proved. Observe that the second argument of q is bounded from above by its first argument, and thus $arg_1 - arg_2$ is a natural number. Extending the constraints-based approach of Decorte *et al*. [14] we define a level-mapping for $q(t_1,t_2)$ as $W_{q_1>q_2} \cdot (t_1 - t_2)$ if $t_1 > t_2$ and as zero, otherwise, where $W_{q_1>q_2}$ is a parameter ranging over a finite domain of natural numbers. In general, a level mapping is a linear combination of differences corresponding to inequalities appearing in adornments.

In order to prove termination, the acceptability condition requires the level-mapping to decrease from the call unified with the head of the clause to the corresponding call to the recursive body subgoal. This condition can be translated into a set of constraints over two kinds of variables: parameters and numerical variables. For our example the following constraint is obtained:

$$W_{q_1>q_2} \cdot (X - Y) > W_{q_1>q_2} \cdot ((X - Y) - Y),$$

that is, $W_{q_1>q_2} \cdot Y > 0$ should hold. In [14] variables such as X and Y above should be interpreted as sizes of terms ranging over the naturals. In our case, they correspond to the integer variables of the program, so we do not know *a priori* that they are natural numbers.

This set of constraints is solved with respect to the parameters. If one can find the parameter values such that the set of constraints is satisfied for all possible values of the numerical variables, termination is reported for all queries.

If for some parameter values one can find additional constraints on the numerical variables corresponding to the query arguments such that the original set of constraints is satisfied, we infer termination for queries satisfying these additional constraints. Finally, if none of the previous cases is applicable possibility of non-termination is suspected. In our case, to conclude the proof we observe that since $W_{q_1>q_2} \geq 0$, then we have $Y > 0$ and $W_{q_1>q_2} > 0$. Variable Y appears in the head of the clause, i.e., $Y > 0$ can be viewed as a constraint on the query. This completes the analysis for the case $arg_1 > arg_2$. Recalling our earlier observations we report termination for $(arg_1 \leq arg_2) \vee (arg_1 > arg_2 \wedge arg_2 > 0)$. Formally, the algorithm is sketched in Figure 1.

Let P be a homogeneous program, let S be a single predicate set of atomic queries and let q be $rel(S)$.

1. For each $p \simeq q$ construct a guard-tuned set $\mathcal{A}_p$.
2. Adorn P with respect to q and $\bigcup_{p\simeq q} \mathcal{A}_p$.
3. Let $A = \{c \mid c \in \mathcal{A}_q$, for all p such that $q^c \sqsupseteq p$: p is not recursive in $P^a\}$. Let $cond_1 = \bigvee_{c\in A} c$. Let $cond_2 = \bigvee_{c\in\mathcal{A}_q, c\notin A} c$.
4. Let S' be $\{c(Q) \wedge Q^c \mid c \in \mathcal{A}_q, c \notin A, c(Q) \wedge Q^c \in S^a\}$.
5. Define the symbolic counterparts of level mappings and interargument relations.
6. Let Σ be a set of constraints on the symbolic variables, following from rigid acceptability of S' with respect to P^a and the validity of interargument relations.
7. Solve Σ with respect to the symbolic variables.
   (a) Solution of Σ doesn't produce extra constraints on variables.
       Report termination for *true*.
   (b) Solution of Σ produces extra constraints on integer variables, appearing in the heads of the clauses.
       Conjunct these constraints to termination condition $cond_2$.
       Report termination for $cond_1 \vee cond_2$.
   (c) There are no solutions or integer variables, constrained by the solution of Σ, do not appear in the heads of the clauses
       Report termination for $cond_1$.

Fig. 1. The Termination Inference Algorithm

One can show that for our running example the termination condition inferred is optimal, i.e., if it is not satisfied the computation will proceed infinitely. However, the undecidability of the termination problem implies that no automatic tool can always guarantee optimality.

Example 4. Consider the following program.

$$q(X,Y) \leftarrow X > Y,\ Z\ is\ X - Y,\ Y1\ is\ Y + 1,\ q(Z,Y1).$$

We would like to study termination of this program with respect to $\{q(z_1, z_2) \mid z_1, z_2$ are integers$\}$. Our algorithm infers the following termination condition: $arg_1 \leq arg_2 \vee (arg_1 > arg_2 \wedge arg_2 \geq 0)$. This is a correct termination condition, but it is not optimal as $q(z_1, z_2)$ terminates, in fact, for all values of z_1 and z_2, i.e., the optimal termination condition is *true*. □

3 Experimental evaluation

We have tested our system on a number of examples. First of all, we considered integer examples from two textbooks' chapters dedicated to programming with arithmetic, namely, Chapter 8 of Sterling and Shapiro [37] and Chapter 9 of Apt [1]. These results are summarised in Table 1. We can prove termination of all the examples presented for all possible values of the integer arguments, that is, the termination condition inferred is *true*. Next, we've collected a number of programs from different sources: mostly from textbooks and benchmark collections. Table 2 presents timings and results for these programs. Again, termination of almost all programs can be shown for all possible values of the integer arguments. We believe that the reason for this is that most textbooks authors aim to teach how to write good software and always keep termination in mind. Finally, Table 3 demonstrates some of the termination conditions inferred by our system. We can summarise our results by saying that the system turns out to be powerful enough to analyse correctly a broad spectrum of programs, while the time spent on the analysis never exceeds 0.30 seconds. In fact, for almost 90% of the integer programs results were obtained in less than 0.10 seconds. Observe that for some examples the time needed to perform the analysis was too small to be measured exactly. These cases are indicated by 0.00 in the Time column.

The core part of the implementation was done in SICStus Prolog (version 3.10.0), using its CLP(FD) and CLP(Q) libraries. The type inference of Janssens and Bruynooghe [20] was implemented in MasterProLog (release 4.1i). Tests were performed on Intel®Pentium®4 with 1.60GHz CPU and 260Mb memory, running 2.4.20-pre11 Linux.

In Tables 1–3, the following abbreviations are used:

- Ref: reference to the program;
- Name: name of the program;
- Queries: atomic queries of interest, where the arguments are denoted
  - c, if the argument is a character;
  - i, if the argument is an integer;
  - li, if the argument is a list of integers;
  - lp, if the argument is a list of pairs of integers;
  - t, if the argument is a binary tree, containing integers;

Table 1. Examples from [37, 1]

Ref	Queries	Time	Ref	Queries	Time
Examples of Sterling and Shapiro [37]			*Examples of Apt [1]*		
8.1	greatest_common_divisor(i, i, v)	0.01	between	between(i, i, v)	0.02
8.2	factorial(i, v)	0.01	delete	delete (i, i, v)	0.03
8.3	factorial(i, v)	0.02	factorial	fact(i, v)	0.01
8.4	factorial(i, v)	0.02	in_tree	in_tree(i, t)	0.00
8.5	between(i, i, v)	0.02	insert	insert(i, t, v)	0.01
8.6a	sumlist(li, v)	0.00	length1	length(li, v)	0.00
8.6b	sumlist(li, v)	0.01	maximum	maximum(li, v)	0.00
8.7a	inner_product(li, li, v)	0.00	ordered	ordered(li)	0.01
8.7b	inner_product(li, li, v)	0.01	quicksort	qs(li, v)	0.06
8.8	area(lp, v)	0.02	quicksort_acc	qs_acc(li, v)	0.05
8.9	maxlist(li, v)	0.02	quicksort_dl	qs_dl(li, v)	0.08
8.10	length(v, li)	0.01	search_tree	is_search_tree(t)	0.03
8.11	length(li, v)	0.01	tree_minimum	minimum(t, v)	0.01
8.12	range(i, i, v)	0.02			

  - v, if the argument is a variable;
- Time: time (in seconds) needed to analyse the example;
- T (in Table 2): termination behaviour:
  - T means that the termination condition inferred is *true*, i.e., computation of any query from the specified set with respect to a given program terminates;
  - N+ means that the termination condition inferred is *false* and indeed computation of a query from the specified set with respect to a given program does not necessarily terminate;
  - T* means that the termination condition inferred is *true*, but a transformation was required as a preliminary step (see further);
- Condition (in Table 3): termination condition (other than *true* and *false*) inferred by the system.

A surprising result was the discovery of non-terminating examples in Prolog textbooks. The first one, by Coelho and Cotta [9], should compute an nth power of a number.

$$exp(X,0,1).$$
$$exp(X,Y,Z) \leftarrow even(Y),\ R\ is\ Y/2,\ P\ is\ X*X,\ exp(P,R,Z).$$
$$exp(X,Y,Z) \leftarrow T\ is\ Y-1,\ exp(X,T,Z1),\ Z\ is\ Z1*X.$$

$$even(Y) \leftarrow R\ is\ Y\ mod\ 2,\ R=0.$$

The termination condition inferred by our system is *false*. Indeed, this is the only termination condition possible, since, for any query $exp(arg_1, arg_2, arg_3)$

Table 2. Various examples

Name	Ref	Queries	Time	T
dldf	[3]	depthfirst2(c, v, i)	0.03	T
exp	[9]	exp(i, i, v)	0.05	N+
fib	[5]	fib(i, v)	0.12	T
fib	[12]	fib(i, v)	0.27	T
fib	[29]	fib(i, v)	0.03	T*
forwardfib	[3]	fib3(i, v)	0.02	T
hanoi	[16]	hanoi(i,v,v,v)	0.18	T
interval	[35]	interval(i,i,v)	0.02	T
money	[5]	money(v, v, v, v,v, v, v, v)	0.13	T
mortgage	[24]	mortgage(i,i,i,i,v)	0.02	T*
oscillate	Example 3	p(i)	0.15	T
p	[15]	p(i)	0.01	T
p32	[19]	gcd(i, i, v)	0.03	T
p33	[19]	coprime(i, i)	0.04	T
p34	[19]	totient_phi(i, v)	0.08	T
primes	[7]	primes(i, v)	0.03	T
pythag	[7]	pythag(v, v, v)	0.03	N+
r	[15]	r(i, v)	0.01	T
triangle	[25]	triangle(i, v)	0.02	N+

Table 3. Examples of inferring termination conditions

Name	Ref	Queries	Time	Condition
q	Example 2	q(i, i)	0.01	$q_1 \leq q_2 \vee (q_1 > q_2 \wedge q_2 > 0)$
q	Example 4	q(i, i)	0.02	$q_1 \leq q_2 \vee (q_1 > q_2 \wedge q_2 \geq 0)$
gcd	[3]	*gcd*(i,i,v)	0.05	$\text{gcd}_1 = \text{gcd}_2 \vee (\text{gcd}_1 > \text{gcd}_2 \wedge \text{gcd}_2 \geq 1)$
loop	[35]	*loop*(i,i,i,i,i)	0.03	$\text{loop}_1 > 0$

such that arg_1 and arg_2 are integers, the LD-tree of this program and this query is infinite due to the infinite traversal of the third clause.

In their book [25] McDonald and Yazdani suggest an exercise that can be seen as computing $\sum_{i=1}^{n} i$ for a given parameter n. The next program is the solution provided by the authors:

$$\textit{triangle}(1,1).$$
$$\textit{triangle}(N,S) \leftarrow M \textit{ is } N-1,\ \textit{triangle}(M,R),\ S \textit{ is } M+R.$$

Once more, the termination condition inferred by our system is *false*, and it is the only possible one. This example and the previous one demonstrate that Hasta-La-Vista can be used for error detection.

Unlike these examples, non-termination of *pythag* [7] is intentional. This program is supposed to compute all Pythagorean triples, i.e., triples of positive

integers a, b and c, such that $a^2 + b^2 = c^2$ holds. Since it is well-known that there are infinitely many different Pythagorean triples, computation necessarily will produce infinitely many answers, i.e., it will be infinite.

Marriott and Stuckey [24] proposed the following CLP program, serving since then as a benchmark for different CLP-analyses.

$$\begin{aligned}
&\mathit{mortgage}(P,T,I,R,B) \leftarrow T = 0,\ B = P.\\
&\mathit{mortgage}(P,T,I,R,B) \leftarrow T \geq 1,\ NT = T - 1,\\
&\qquad NP = P + P * I - R,\ \mathit{mortgage}(NP,NT,I,R,B).
\end{aligned}$$

As a preprocessing step, we transform this program into the non-CLP form by replacing the two equality constraints in the second clause with *is*-assignments:

$$\begin{aligned}
&\mathit{mortgage}(P,T,I,R,B) \leftarrow T = 0,\ B = P.\\
&\mathit{mortgage}(P,T,I,R,B) \leftarrow T \geq 1,\ NT\ \mathit{is}\ T - 1,\\
&\qquad NP\ \mathit{is}\ P + P * I - R,\ \mathit{mortgage}(NP,NT,I,R,B).
\end{aligned}$$

Such a replacement can be performed automatically, provided that variables T,P,I,R are ground and variables NT and NP are free prior to constraints application. Termination of the transformed program can be shown by our system.

Finally, O'Keefe [29] suggested a more efficient way to calculate Fibonacci numbers, by performing $O(n)$ work each time it is called, unlike the versions of [5] and [12] which perform an exponential amount of work each time.

$$\begin{aligned}
&\mathit{fib}(1,X) \leftarrow !,\ X = 1.\\
&\mathit{fib}(2,X) \leftarrow !,\ X = 1.\\
&\mathit{fib}(N,X) \leftarrow N > 2,\ \mathit{fib}(2,N,1,1,X).
\end{aligned}$$

$$\begin{aligned}
&\mathit{fib}(N,N,X2,_,X) \leftarrow !,\ X = X2.\\
&\mathit{fib}(N0,N,X2,X1,X) \leftarrow N1\ \mathit{is}\ N0 + 1,\\
&\qquad X3\ \mathit{is}\ X2 + X1,\ \mathit{fib}(N1,N,X3,X2,X).
\end{aligned}$$

Termination of goals of the form fib(i,v) with respect to this example depends on the cut in the first clause of $\mathit{fib}/5$. If it is replaced with $N_0 \neq N$ in the second clause termination can be proved. This fact is denoted T* in Table 2. Note that this replacement does not affect the complexity of the computation.

Clearly, since the halting problem is undecidable we cannot expect our algorithm to compute precise termination condition for any given example in a finite time. Termination proofs of many contrived examples, like Takeuchi's function, require sophisticated argumentation. Moreover, termination of certain numerical computations, such as the $3n+1$ problem attributed to L. Collatz, is still an open problem in mathematics.

4 Further extensions

In this section we review briefly possible extensions of the Hasta-La-Vista system. First of all, as suggested in [32], an adapted version of the technique proposed can be applied to infer termination of programs using floating point numbers. The major difference with the basic Hasta-La-Vista algorithm is that *rounding functions*, specified in IEEE standard 754, should be taken into account. For instance, "*X1 is* $-X^2$" can no longer be understood as $x1 = -x^2$ as in the integer case, but should be interpreted as $x1 = round(-(round(x^2)))$.

Next, while Hasta-La-Vista was originally developed for *definite programs*, i.e., programs without negation, it can be extended to analyse *normal programs*, i.e., programs that contain negation in bodies of their clauses. The simplest way to do is to require that the size of a negative literal measured by a given level mapping should be equal to the size of the corresponding positive literal. This approach would allow us to prove termination of a number of examples, such as *primes* [7] and *Goldbach's conjecture* [19].

We also can use the ideas of adornment to prove termination of symbolical computations, i.e., computations on terms. Two possible directions seem to be promising. First, a number of modern approaches to termination analysis of logic programs [8,27] abstract a program to CLP($\mathbb{N}$) and then infer termination of the original program from the corresponding property of the abstract one. Unfortunately, techniques used to prove termination of numerical programs are often restricted to the use of the identity function as the only level-mapping, which results in failure to prove termination in many interesting examples. In some cases a CLP($\mathbb{N}$) program can be seen as a Prolog program with numerical computation. For these programs our technique can be applied to improve the precision of the analysis. Alternatively, one can try and avoid transformation by using the ideas of adorning directly on the symbolical program [34].

Finally, observe that in real-world programs, numerical computations are sometimes interleaved with symbolical ones, as illustrated by the following example. This example collects leaves of a tree with a variable branching factor, which is a common data structure in natural language processing [30].

Example 5.

$$
\begin{aligned}
&collect(X,[X|L],L) \leftarrow atomic(X).\\
&collect(T,L0,L) \leftarrow compound(T),\ functor(T,\_,A), \qquad (1)\\
&\qquad process(T,0,A,L0,L).\\
&process(\_,A,A,L,L).\\
&process(T,I,A,L0,L2) \leftarrow I < A,\ I1\ is\ I+1,\ arg(I1,T,Arg), \qquad (2)\\
&\qquad collect(Arg,L0,L1),\ process(T,I1,A,L1,L2).
\end{aligned}
$$

To prove termination of $\{collect(t, v, [])\}$, where t is a tree and v is a variable, three decreases should be shown: between a call to *collect* and a call to *process* in (1), between a call to *process* and a call to *collect* in (2) and between two calls to *process* in (2) . The first two can be shown only by a symbolic level mapping, the third one—only by the numerical approach. □

Thus, our goal is to *combine* the existing symbolic approaches with the numerical one presented so far. One of the possible ways to do so is to combine two level mappings, $|\cdot|_1$ and $|\cdot|_2$, by mapping each atom $A \in B_P^E$ either to a natural number $|A|_1$ or to a pair of natural numbers $(|A|_1, |A|_2)$. Then we prove termination by showing decreases via orderings on $(\mathbb{N} \cup \mathbb{N}^2)$ as suggested in [13].

Example 6. Example 5, continued. Define $\varphi : B_P^E \to (\mathbb{N} \cup \mathbb{N}^2)$ as: $\varphi(collect(t, l0, l)) = \|t\|$ and $\varphi(process(t, i, a, l0, l)) = (\|t\|, a - i)$, where $\|\cdot\|$ is a term-size norm. The decreases are satisfied with respect to $>$, such that $A_1 > A_2$ if and only if $\varphi(A_1) \succ \varphi(A_2)$, where $\succ$ is defined as follows: $n \succ m$ if $n >_{\mathbb{N}} m$; $n \succ (n, m)$ if *true*; $(n, m_1) \succ (n, m_2)$ if $m_1 >_{\mathbb{N}} m_2$; $(n_1, m) \succ n_2$ if $n_1 >_{\mathcal{N}} n_2$, where $>_{\mathbb{N}}$ is the usual order on the naturals.

Indeed, $collect(t, l0, l) > process(t, 0, a, l0, l)$, since $\varphi(process(t, 0, a, l0, l)) = (\|t\|, a)$, $\varphi(collect(t, l0, l)) = \|t\|$, and $\|t\| \succ (\|t\|, a)$ by definition of $\succ$. Similarly, $process(t, i, a, l0, l2) > collect(arg, l0, l1)$, since $\varphi(process(t, i, a, l0, l2)) = (\|t\|, a - i), \varphi(collect(arg, l0, l1)) = \|arg\|$, $\|t\| >_{\mathbb{N}} \|arg\|$ by the predefined semantics of the built-in predicate *arg* and $(\|t\|, a - i) \succ \|arg\|$ by definition of $\succ$. Finally, $process(t, i, a, l0, l2) > process(t, i1, a, l1, l2)$, since $\varphi(process(t, i, a, l0, l2)) = (\|t\|, a - i), \varphi(process(t, i1, a, l1, l2)) = (\|t\|, a - i1), a - i >_{\mathbb{N}} a - i1$ (since $i1 = i + 1$) and $(\|t\|, a - i) \succ (\|t\|, a - i1)$ by definition of $\succ$. □

This integrated approach allows one to analyse correctly examples such as *ground*, *unify*, *numbervars* [37] and Example 6.12 in [15].

5 Conclusion

We have presented Hasta-La-Vista, a termination analyser for logic programs with integer computations. This functionality is lacking in currently available termination analysers for Prolog, such as TerminWeb [8], cTI [26], TALP [28], and TermiLog [22]. The main contribution of this work is in integrating termination analysis for numerical computations in the automatic termination analyser of [14]. It was shown that our approach is robust enough to prove termination for a wide range of numerical examples. To the best of our knowledge, Hasta-La-Vista is unique in being able to prove termination of programs depending on integer computations. This work plays a complementary role with respect to [33] as it highlights the implementation aspects of the system and provides detailed

discussion of experimental evaluation. In the subsequent papers [32, 34] we investigate how the adornment technique can be applied to domains other than the domain of the integers, namely, the floating point numbers and the Herbrand domain.

Termination of numerical computations was studied by a number of authors [1, 2, 15]. In [1] it was suggested that arithmetic computations should be simply ignored. Apt *et al*. [2] provided an important theoretical characterisation of strong termination, but it seems to be difficult to integrate the approach with automatic tools. Moreover, there are programs that terminate only for *some* queries, such as Example 2. Alternatively, Dershowitz *et al*. [15] extended the query-mapping pairs formalism of [22] to deal with numerical computations. However, this approach inherited the disadvantages of [22], such as a high computational price, which is inherent to this approach due to the repetitive fixpoint computations. In contrast to their work which was restricted to the verification of termination, Hasta-La-Vista can *infer* termination conditions. It is not clear whether and how [15] can be extended to infer such conditions.

More research has been done on termination analysis for constraint logic programs. Since numerical computations in Prolog should be written in a way that allows a system to verify their satisfiability we can see numerical computations of Prolog as an *ideal constraint system*. Thus, all the results obtained for ideal constraint systems can be applied. Unfortunately, the research was either oriented towards theoretical characterisations [31] or restricted to domains isomorphic to $\mathbb{N}$ [26], such as trees and terms. Recently, in the journal revision [27] of [26] and [31], a possibility is mentioned of using abstraction functions other than combinations of term-size norm, list-length norm, identity function and null-function. However, the question of *how* these functions should be inferred automatically is not considered, and the cTI implementation is restricted to the term-size norm as an abstraction function.

Numerical computations have also been analysed in the early works on termination analysis for imperative languages [17]. However, our approach to automation differs significantly from these works. Traditionally, the verification community considered automatic generation of invariants, while automatic generation of ranking functions (level mappings, in the logic programming parlance) just started to emerge [10]. The inherent restriction of automatically generated ranking functions is that they have to be linear. Moreover, in order to perform the analysis of larger programs, such as *mergesort*, in a reasonable amount of time, the ranking functions were further restricted so that they depend on one variable only. Our approach doesn't suffer from such limitations.

The idea of splitting a predicate into cases was first mentioned by Ullman and Van Gelder [38]. However, neither in this paper, nor in the subsequent

one [36] was the proposed methodology presented formally. To the best of our knowledge, the first formal presentation of splitting in the framework of termination analysis is due to Lindenstrauss *et al*. [23]. Unlike in their work, a numerical and not a symbolic domain was considered in the current paper. Distinguishing different subsets of values for variables, and deriving norms and level mappings based on these subsets, links our approach to the ideas of using *type information* in termination analysis for symbolic computations [4]. Indeed, adornments can be seen as types, refining the predefined type *integers*. Unlike these works, our work does not start with a given set of types, but for each program derives a collection of types relevant to this program.

Acknowledgement. We are very grateful to Gerda Janssens for making her type analysis system available to us.

References

1. K. R. Apt. *From Logic Programming to Prolog*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1997.
2. K. R. Apt, E. Marchiori, and C. Palamidessi. A declarative approach for first-order built-in's in Prolog. *Applicable Algebra in Engineering, Communication and Computation*, 5(3/4):159–191, 1994.
3. I. Bratko. *Prolog programming for Artificial Intelligence*. Addison-Wesley, 1986.
4. M. Bruynooghe, M. Codish, S. Genaim, and W. Vanhoof. Reuse of results in termination analysis of typed logic programs. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis, 9th International Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 477–492. Springer Verlag, 2002.
5. F. Bueno, M. J. García de la Banda, and M. V. Hermenegildo. Effectiveness of global analysis in strict independence-based automatic parallelization. In M. Bruynooghe, editor, *Logic Programming, Proceedings of the 1994 International Symposium*, pages 320–336. MIT Press, 1994.
6. J. Cameron. Terminator 2: Judgment day. 1991.
7. W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer Verlag, 1981.
8. M. Codish and C. Taboch. A semantic basis for termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999.
9. H. Coelho and J. C. Cotta. *Prolog by example*. Springer Verlag, 1988.
10. M. A. Colón and H. B. Sipma. Synthesis of linear ranking functions. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference*, volume 2031 of *Lecture Notes in Computer Science*, pages 67–81. Springer Verlag, 2001.
11. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
12. M. A. Covington, D. Nute, and A. Vellino. *Prolog Programming in Depth*. Prentice Hall, second edition, 1996.
13. D. De Schreye and A. Serebrenik. Acceptability with general orderings. In A. C. Kakas and F. Sadri, editors, *Computational Logic. Logic Programming and Beyond. Essays in Honour*

of Robert A. Kowalski, Part I, volume 2407 of *LNCS*, pages 187–210. Springer Verlag, July 2002.

14. S. Decorte, D. De Schreye, and H. Vandecasteele. Constraint-based termination analysis of logic programs. *ACM TOPLAS*, 21(6):1137–1195, November 1999.
15. N. Dershowitz, N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. A general framework for automatic termination analysis of logic programs. *Applicable Algebra in Engineering, Communication and Computing*, 12(1-2):117–156, 2001.
16. J. Fisher. The Prolog tutorial. Available at `http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/contents.html`, December 1999.
17. R. W. Floyd. Assigning meanings to programs. In J. Schwartz, editor, *Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967. Proceedings of Symposiumsia in Applied Mathematics; v. 19.
18. S. Genaim and M. Codish. Inferring termination conditions for logic programs using backwards analysis. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conferencerence, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 685–694. Springer Verlag, 2001.
19. W. Hett. P-99: Ninety-nine Prolog problems. Available at `http://www.hta-bi.bfh.ch/~hew/informatik3/prolog/p-99/`, July 2001.
20. G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2&3):205–258, July 1992.
21. G. Janssens, M. Bruynooghe, and V. Englebert. Abstracting numerical values in CLP(H, N). In M. V. Hermenegildo and J. Penjam, editors, *Programming Language Implementation and Logic Programming, 6th International Symposiumsium, PLILP'94*, volume 844 of *Lecture Notes in Computer Science*, pages 400–414. Springer Verlag, 1994.
22. N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 63–77. MIT Press, July 1997.
23. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Unfolding the mystery of *mergesort*. In N. Fuchs, editor, *Proceedings of the 7^{th} International Workshop on Logic Program Synthesis and Transformation*, volume 1463 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.
24. K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, Cambridge, MA, USA, 1998.
25. C. McDonald and M. Yazdani. *Prolog programming: a tutorial introduction*. Artificial Intelligence Texts. Blackwell Scientific Publications, 1990.
26. F. Mesnard. Inferring left-terminating classes of queries for constraint logic programs. In M. Maher, editor, *Proceedings of the 1996 Joint International Conference and Syposium on Logic Programming*, pages 7–21, Cambridge, MA, USA, 1996. The MIT Press.
27. F. Mesnard and S. Ruggieri. On proving left termination of constraint logic programs. *ACM Transaction on Computational Logic*, 4(2):207–259, 2003.
28. E. Ohlebusch, C. Claves, and C. Marché. TALP: A tool for the termination analysis of logic programs. In L. Bachmair, editor, *11th International Conference on Rewriting Techniques and Applications*, volume 1833 of *Lecture Notes in Computer Science*, pages 270–273. Springer Verlag, 2000.
29. R. A. O'Keefe. *The Craft of Prolog*. MIT Press, Cambridge, MA, USA, 1990.
30. C. Pollard and I. A. Sag. *Head-driven Phrase Structure Grammar*. The University of Chicago Press, 1994.

31. S. Ruggieri. Termination of constraint logic programs. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *Automata, Languages and Programming, 24th International Colloquium, ICALP'97*, volume 1256 of *Lecture Notes in Computer Science*, pages 838–848. Springer Verlag, 1997.
32. A. Serebrenik and D. De Schreye. On termination of logic programs with floating point computations. In M. V. Hermenegildo and G. Puebla, editors, *9th International Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 151–164. Springer Verlag, 2002.
33. A. Serebrenik and D. De Schreye. Inference of termination conditions for numerical loops in Prolog. *Theory and Practice of Logic Programming*, 2003. accepted.
34. A. Serebrenik and D. De Schreye. Proving termination with adornments. In M. Bruynooghe, editor, *Pre-Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation*, pages 133–148. Katholieke Universiteit Leuven, 2003.
35. K. Sloneger. Lecture notes for course programming language foundations. Logic programming with Prolog. Available at `http://www.cs.uiowa.edu/~slonnegr/plf/`, 2001.
36. K. Sohn and A. Van Gelder. Termination detection in logic programs using argument sizes. In *Proceedings of the Tenth ACM SIGACT-SIGART-SIGMOD Symposium on Principles of Database Systems*, pages 216–226. ACM Press, 1991.
37. L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, MA, USA, 1994.
38. J. D. Ullman and A. Van Gelder. Efficient tests for top-down termination of logical rules. *Journal of the ACM*, 35(2):345–373, April 1988.
39. W. Winsborough. Multiple specialization using minimal-function graph semantics. *Journal of Logic Programming*, 13(2/3):259–290, 1992.

Constructive combination of Crisp and Fuzzy Logic in a Prolog Compiler

Susana Muñoz[1], Claudio Vaucheret[2], and Sergio Guadarrama[2]

[1] Departamento de Lenguajes, Sistemas de la Información e Ingeniería del Software
susana@fi.upm.es
[2] Departamento de Inteligencia Artificial
claudio@clip.dia.fi.upm.es sgaumela@isys.fi.upm.es
Facultad de Informática - Universidad Politécnica de Madrid
Campus de Montegancedo 28660 Madrid, Spain

Abstract. We have presented before a Fuzzy Prolog Language that models interval-valued Fuzzy logic and we have provided an implementation using CLP($\mathcal{R}$). Now, in this work, we describe a sound method for combining crisp and fuzzy logic in one Prolog compiler. The result is a powerful fuzzy Prolog library with an intuitive semantics that works in a constructive way even with negative queries. The implementation is incredibly simple because we are using Prolog's resolution so it is a useful tool for introducing uncertainty in crisp logic programs.

Keywords Fuzzy Prolog, Modeling Uncertainty, Logic Programming, Constraint Programming Application, Implementation of Fuzzy Prolog, Logic Negation, Constructive Negation.

1 Introduction

In [17] we have presented a Fuzzy Prolog Language that models interval-valued Fuzzy logic and we have presented an implementation using CLP($\mathcal{R}$) [5]. This Fuzzy Prolog uses the original inference mechanism of Prolog, and it uses constraints and operations provided by CLP($\mathcal{R}$) to handle the concept of partial truth. In this paper we extend the implementation to combine fuzzy and crisp predicates in the same code. The implementation, as syntactic extension of Prolog, provides a useful tool to allow the mixing of fuzzy predicates with Prolog predicates in the same body clause. We have had to solve some semantics problems using a constructive implementation of *Logic Negation*, which avoids getting unsound results.

The rest of the paper is organized as follows. Section 2 summarizes the syntax and semantics of our fuzzy Prolog system (presented in [17]). Section 3 deals with the new operational semantics considered for this work. The next two sections give us an intuition about the problem. Section 4 introduces the necessity of combining crisp and fuzzy logic in the definition of predicates and how this leads us to fuzzify predicates. In Section 5 we explain the importance of giving constructive answers and how to get them using a constructive implementation of Logic Negation. Finally, we conclude and discuss some future work (Section 6).

2 Fuzzy Prolog

The Language There is no agreement on which fuzzy logic should be used. Most of fuzzy systems use min-max logic (for modeling the conjunction and disjunction operations), but other systems just use Lukasiewicz logic [6]. Furthermore, logic programming is considered a useful tool for implementing methods for reasoning with uncertainty in [19]. There is also an extension of constraint logic programming [1], which can model logics based on semiring structures. This framework can model min-max fuzzy logic, which is the only logic with a semiring structure.

Recently, a theoretical model for fuzzy logic programming without negation, which deals with many-value implications, has been proposed by Votjas in [18]. Over the last few years several papers have been published by Medina et all. ([9,10,8]) about multi-adjoint programming, which describe a theoretical model, but no means of implementation.

In [17], we have proposed another approach more general in some respects:

1. A truth value will be a finite union of sub-intervals on $[0,1]$. An interval is a particular case of union of one element, and a unique truth value is a particular case of having an interval with only one element.
2. A truth value will be propagated through the rules by means of an *aggregation operator*. The definition of *aggregation operator* is general in the sence that it subsumes conjunctive operators (triangular norms [7] like min, prod, etc.), disjunctive operators [16](triangular co-norms, like max, sum, etc.), average operators (like arithmetic average, quasi-linear average, etc) and hybrid operators (combinations of the above operators [14]).
3. The declarative and procedural semantics for Fuzzy Logic programs are given and their equivalence is proven.
4. An implementation of the proposed language is presented.

A *fuzzy program* is a finite set of *fuzzy facts* and *fuzzy clauses* and we obtain information from the program through *fuzzy queries*. They are defined as usual but handling truth values in $\mathcal{B}([0,1])$ (the Borel Algebra over the real interval $[0,1]$ that yields with unions of intervals) represented as constraints. We refer, for example, to expressions as: $(v \geq 0.5 \ \wedge \ v \leq 0.7) \ \vee \ (v \geq 0.8 \ \wedge \ v \leq 0.9)$ to represent a truth value in $[0.5,0.7] \bigcup [0.8,0.9]$.

Examples These definitions of fuzzy sets entail different degrees of fuzziness. Figure 1 shows the concept of *youth* with four different interpretations.

The level of fuzziness increases from the crisp function or the simple fuzzy function, where every age has only one real number representing its youth, to one where an interval represents, for example, the concept of youth of a group of people with slightly different definitions of the borders of the function. However, if we ask two different groups of people, for example, people from two different continents, we might get a representation like the last one. The truth value of youth for 45 years has evolved from the value 0 in the crisp model, to the value 0.5 in the simple fuzzy definition, later to the interval $[0.2,0.5]$ and, finally, to the union of intervals $[0.2,0.5] \bigcup [0.8,1]$.

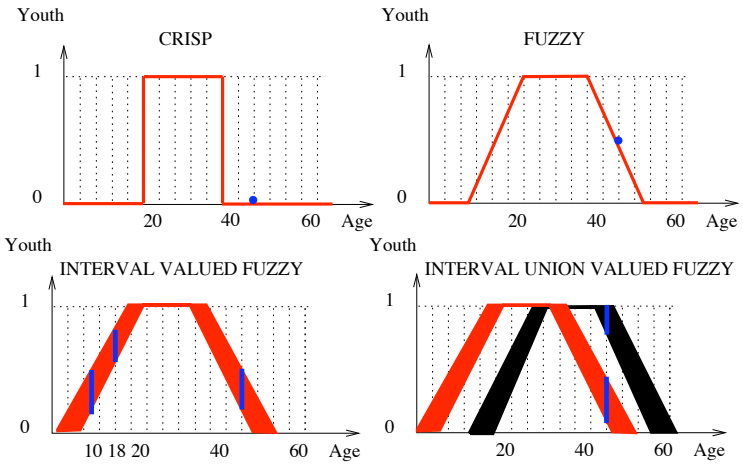


Fig. 1. Uncertainty level of a fuzzy predicate

There are many everyday situations that can only be represented by this general representation of truth value. Here, we provide some simple examples with their representation in our fuzzy language:

- Example 1: My father is 45 years old. If someone asked me how young he was, I would assign $V \in [0.2, 0.5]$, but if someone asked him how young he was, he would assign himself $V \in [0.8, 1]$. So we can say that he is young with $V \in ([0.2, 0.5] \bigcup [0.8, 1])$.
- Example 2: My sons are 10 and 18 years old. My neighbour's daughter, Jane, is the same age as one of my sons, but I cannot remember which one. If I consider the third fuzzy definition of truth, then I can say that Jane is young with a truth value $V \in ([0.3] \bigcup [0.9]) \neq [0.6]$. That is:

```
young(jane) :- [0.3] v [0.9].
```

- Example 3: New Laptop is a computer company producing a laptop model. This model is very slow when running graphic applications, but is very fast when running office applications. If a customer buys a New Laptop computer, the truth value of its speed will be $V \in ([0.02, 0.08] \bigcup [0.75, 0.90])$. Depending on the use to which it is put, however, its speed could be the lowest, the highest or even an average.

```
fast(newLaptop) :~ [0.02, 0.08] v [0.75, 0.90].
```

where each truth value is a union of intervals. The intervals in the first example represent the particular case of intervals consisting of only one element.

Fuzziness versus Uncertainty We can model many real problems thanks to this powerful notation and it is interesting to note that we can handle both uncertainty and fuzziness at the same time with this truth value representation.

Let us go back to Example 2 of section 2. We will represent now the truth value of the concept of youth as an interval as in the third representation in Figure 1, instead of using real numbers. We can say that, in this case, Jane is young with a truth value $V \in$ ([0.2,0.5] $\bigcup$ [0.6,0.8]). It is a union of intervals which represents uncertainty, because we do not know which of the two intervals represents the how young Jane is (we do not know which one of the two possible values, 10 or 18, is her age).

Example 1 presented above shows the truth value of youth as a union of intervals as in the fourth representation in Figure 1 for 45 years of age which is $[0.1,0.4] \bigcup [0.8,1]$. It is a union of intervals, which, in this case, is representing fuzziness, because the concept of youth is represented with the maximum level of fuzziness. We know that the age is 45 (there is no uncertainty about the age) but the truth value that represents its youth is fuzzy (lack of precision).

Although both representations (fuzziness and uncertainty) are semantically different, they are handled using the same syntax in a sound way as it was described in [17].

The Implementation We decided to implement our interpreter as a syntactic extension of a CLP($\mathcal{R}$) system. Particularly, we have written a library (or *package* in the Ciao Prolog terminology) called *fuzzy* which implements the interpreter of our fuzzy Prolog language using the CLP($\mathcal{R}$) library of the Ciao Prolog system[1].

Each Fuzzy Prolog clause has an additional argument in the head that represents its truth value in terms of the truth values of the subgoals of the body of the clause. A fact $A \leftarrow v$ is represented by a Fuzzy Prolog fact that describes the range of values of v with a union of intervals (that can be an interval or even a real number in particular cases). The following examples illustrate the concrete program syntax:

$youth(45) \leftarrow [0.2,0.5] \bigcup [0.8,1]$	`youth(45,V) :~ [0.2,0.5] v [0.8,1].`
$tall(john) \leftarrow [0.8,0.9]$	`tall(john,V) :~ [0.8,0.9].`
$swift(john) \leftarrow 0.7$	`swift(john,0.7) :~.`
$good\_player(X) \leftarrow_{min} tall(X),$ $swift(X)$	`good_player(X,V) :~ min tall(X,V1), swift(X,V2).`

These clauses are expanded at compilation time to constrained clauses that are managed by CLP($\mathcal{R}$) at run time. Predicates $.=./2, .</2, .<=./2, .>/2$ and $.>=./2$ are the Ciao CLP($\mathcal{R}$) operators for representing constraint inequalities. For example, the first fuzzy fact is expanded to these Prolog clauses with constraints

```
youth(45,V):-   V .>=. 0.2,          youth(45,V):-   V .>=. 0.8,
                V .<=. 0.5.                          V .<.  1.
```

and the fuzzy clause

```
good_player(X) :~ min tall(X), swift(X).
```

is expanded to

```
good_player(X,Vp) :-  tall(X,Vq),
                      swift(X,Vr),
```

[1] The Ciao system including our Fuzzy Prolog implementation can be downloaded from http://www.clip.dia.fi.upm.es/Software.

```
minim([Vq,Vr],Vp),
Vp .>=. 0,
Vp .=<. 1.
```

The predicate `minim/2` is included as run time code by the library. Its function is to add constraints to the truth value variables in order to implement the T-norm *min*. We have implemented several *aggregation operators* as `min`, `prod`, `max`,`luka` (Lukasiewicz operator [6]), etc. in a similar way, and any other operator can be added by the user to the system without any effort.

It is possible to fuzzify crisp predicates. For example, to fuzzify $p/2$:

```
p_f :# fuzzy p/2.
```

and the program is expanded with a new fuzzy predicate $p\_f/3$ (the last argument is the truth value) with truth value equal to 1 if $p/2$ succeeds and 0 otherwise. The internal implementation of the fuzzified predicate is discussed in Section 4.1 and at the end of Section 5.1.

We also provide the possibility of having the predicate that is the fuzzy negation of a fuzzy predicate. For this predicate $p\_f/3$, we will define a new fuzzy predicate called, for example, $not\,p\_f/3$ with the following line (note that ": #" is our way of distinguishing a renaming rule from a fuzzy rule,":∼", and from a Prolog rule, ": −"):

```
notp_f :# fnot  p_f/3.
```

that is expanded at compilation time as the common fuzzy negation:

```
notp_f(X,Y,V) :-  p_f(X,Y,Vp),
                  V .=. 1 - Vp.
```

3 Operational Semantics

The procedural semantics is interpreted as a sequence of transitions between different states of a system. We represent the state of a *transition system* in a computation as a tuple $\langle A, \sigma, S \rangle$ where A is the goal, σ is a substitution representing the instantiation of variables needed to get to this state from the initial one and S is a constraint that represents the truth value of the goal at this state.

When computation starts, A is the initial goal, $\sigma = \emptyset$ and S are true (if there are neither previous instantiations nor initial constraints). When we get to a state where the first argument is empty, then we have finished the computation and the other two arguments represent the answer. If a constraint c has solution in the domain of real numbers in the interval $[0,1]$ then we say c is *consistent*, and we denote it as $solvable(c)$.

A *transition* in the *transition system* is defined as:

1. $\langle A \cup a, \sigma, S \rangle \rightarrow \langle A\theta, \sigma \cdot \theta, S \wedge (\mu_a = v) \rangle$
   if $h \leftarrow v$ is a fact of the program P, θ is the mgu of a and h, and μ_a is the truth variable for a, and $solvable(S \wedge (\mu_a = v))$.

2. $\langle A \cup a, \sigma, S \rangle \rightarrow \langle (A \cup B)\theta, \sigma \cdot \theta, S \wedge c \rangle$
   if $h \leftarrow_F B$ is a rule of the program P, θ is the mgu of a and h, c is the constraint that represents the truth value obtained applying the aggregator F on the truth variables of B, and $solvable(S \wedge c)$.
3. $\langle A \cup a, \sigma, S \rangle \rightarrow \langle A \cup a, \sigma, true \rangle$
   if neither of the above are applicable. The constraint is satisfied because there is no limitation over the truth value (we do not have enough information to bound to it).

The success set $SS(P)$ collects the answers to simple goals $p(\overline{x})$ of P. It is defined as follows: $SS(P) = \langle B, V \rangle = \{\langle p(\overline{x})\sigma, v \rangle | \langle p(\overline{x}), \emptyset, true \rangle \rightarrow^* \langle \emptyset, \sigma, S \rangle$ and v is the solution of $S\}$, where B is the set of elements of the Herbrand Base that are instantiated and that have succeeded, and V is the set of truth values of the elements of B, which is the union of truth values that are obtained from the set of constraints provided by the program P while query $p(\overline{x})$ is computed. When S is *true* then the value of v is unlimited, e.g. $v \in \{0..1\}$ that means for us indefinite symbol, $\perp$ (in the sense of lack of information).

The drawback of this representation is that it blocks the truth value propagation in the sense that if v_j is $\perp$ then $V(p(\widehat{x})) = v_1 \cup ... \cup v_j \cup ... \cup v_n = \perp$
but it does not introduce inconsistence, it is only a loss of precision because our main goal is to introduce uncertainty and fuzziness in crisp logic programs and this interpretation is compatible with Prolog as we see in the next section.

4 Combining Crisp and Fuzzy Logic

To use definitions of fuzzy predicates that include crisp subgoals, we must define their semantics properly with respect to the Prolog Close World Assumption (CWA) [4].

Fuzzy clauses usually use crisp predicate calls as requirements to be satisfied by data to verify the definition at a level higher than 0. For example, if we can say that a teenage student is a student whose age is about 15, then we can define the fuzzy predicate *teenage_student*/2 as

```
teenage_student(X):~ student(X),
                     age_about_15(X).
```

In this example, the goal $teenage\_student(X,V)$ should output:

- $V = 0$ if the value of X is not the name of a student.
- The respective truth value V if the value of X is the name of a student and we know that his age is about 15 at some level.
- Unknown if the value of X is the name of a student, but we do not know anything about his/her age.

Note that we run the risk of unsoundness, unless the semantics of crisp and fuzzy predicates is properly defined. CWA means that all information that is not explicitly true is false. For example, if we have the predicate definition of *student*/1 as

```
student(john).
student(peter).
```

then we have that the goal $student(X)$ is successful with $X = john$ or with $X = peter$, but fails with any other value apart from these; for instance:

```
?- student(john).
yes
```

```
?- student(nick).
no
```

which means that *john* is a student and *nick* is not. This is the semantics of Prolog, and it is the one we are going to adopt for crisp predicates because we want our system to be compatible with conventional Prolog reasoning. But what about fuzzy predicates? We have considered two main possibilities. For example, consider the case when we have the definition of *age_about*_15/2 as

```
age_about_15(john,1):~ .
age_about_15(susan,0.7):~ .
```

where the goal *age_about*_15(X,V) is successful for $X = john$ and $V = 1$ or for $X =$ *susan* and $V = 0.7$. If we want to work with CWA, like crisp predicates do, then we will get $V = 0$ for any other value of X apart from *john* and *susan*. The meaning is that the predicate is defined for all values and the membership value will be 0 if the predicate is not explicitly defined with another value. In this example, we know that the age of *john* is 15 and *susan*'s age is about 15 and, using CWA, we are also saying people are not about 15. This semantics is equivalent to crisp definitions, but we think that our meaning is usually different, i.e. in this case, we may mean that we know that *john* and *susan* are about 15 and that we have no information about the age of the other people. Therefore, we do not know whether or not the age of *peter* is about 15; and if we know that *nick*'s age is not about 15, we can explicitly declare

```
age_about_15(nick,0):~ .
```

We are going to work with this semantics for fuzzy predicates, because we think it is the most akin to human reasoning. So, a fuzzy goal (asking for a crisp goal) can be true (value 1), false (value 0) or have another membership value. We have added the concept of `unknown` to represent no explicit knowledge in fuzzy definitions. This new state can be very simply represented, using the common failure of Prolog. We give it the meaning of `unknown`, considering that the meaning that it has in crisp logic is not necessary here because it is represented with membership value 0 in fuzzy logic. For example, with our definition of *age_about*_15/2, we will get

```
?- age_about_15(john,X).
X = 1
```

```
?- age_about_15(peter,X).
no
```

```
?- age_about_15(nick,X).
X = 0
```

This means *john*'s age is about 15, *nick*'s age is not about 15 and we have no data about *peter*'s age.

We expect the fuzzy predicate *teenage_student*/2 to behave similarly, i.e.:

```
?- teenage_student(john,V).
V .=. 1

?- teenage_student(susan,V).
V .=. 0
```

```
?- teenage_student(peter,V).
no
```

as *john* is a "teenage student" (he is a student and his age is about 15), *susan* is not a "teenage student" (she is not a student) and we do not know the maturity value for *peter*, because although he is a student, we do not know whether his age is about 15. The way to do this is by overcoming the CWA behavior of the crisp predicate $student/1$ to get the truth value 0 for $student(susan)$. The solution is to fuzzify crisp predicates when they are in the body of fuzzy clauses.

For each crisp predicate in the definition of a fuzzy predicate, the compiler will generate a fuzzy version to replace the original predicate in the body of the clause. For the above example of crisp predicate $student/1$, the compiler will produce the predicate $f\_student/2$, which is an equivalent fuzzy predicate to the crisp one. So, the definition of $teenage\_student/2$ will be changed to

```
teenage_student(X):~  f_student(X),
                      age_about_15(X).
```

Now the internal fuzzy solution is simple, sound and very homogeneous, because we only consider fuzzy subgoals in the body of the clause. The only problem we now have to solve is how to fuzzify crisp predicates. In the following section, we are going to describe how we have done this.

4.1 Fuzzified Predicates

The first simple approach is to use the *cut* of Prolog to implement the respective fuzzified predicate. So, for a predicate $student(X)$ we have:

```
f_student(X,V):-  student(X),!,
                  V .=. 1.
f_student(X,0).
```

The result is that $V = 1$ if the crisp goal is successful and $V = 0$ otherwise. In this way, we get the expected results, that retain the same meaning as the crisp predicates, for the following calls

```
?- f_student(peter,V).
V .=. 1
```

```
?- f_student(susan,V).
V .=. 0
```

Nevertheless, we come across a problem with goals like:

```
?- f_student(X,1).
X = john
```

where the cut prevents backtracking, and it is impossible to get the entire solutions. This problem is simply sorted out with the alternative transformation:

```
f_student(X,V):-  if(student(X),V=1,V=0).
```

It solves the backtracking problem because of the implementation of the $if/3$ predicate and returns

```
?- f_student(X,1).
X = john ? ;
X = peter ? ;
no
```

This is not only useful for giving constructive answers to goals of fuzzified predicates but it is also the way to get constructive solutions to fuzzy queries of a fuzzy predicate that is defined combining crisp and fuzzy logic. This is illustrated in Section 5.

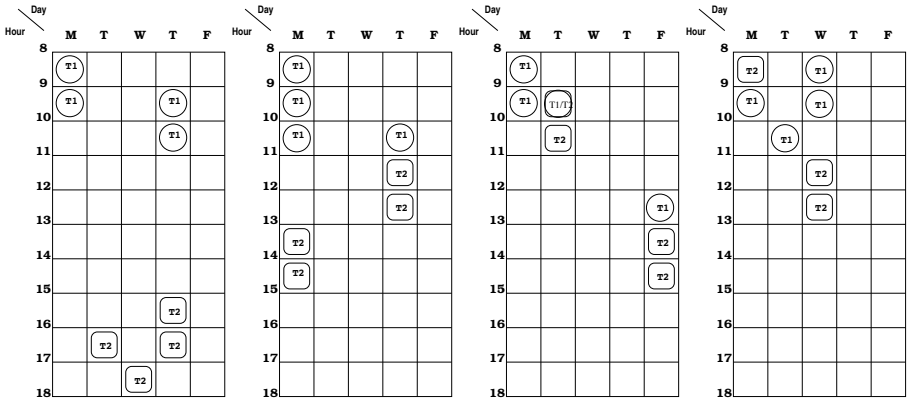


Fig. 2. Timetable 1, 2, 3 and 4

Example Another real example would be the problem of the compatibility of a couple of shifts at a place of work, for example, teachers who work to different class timetables, telephone operators, etc. Imagine a company where the work is divided in to shifts of 4 hours per week. Many workers have to combine a couple of shifts in the same week and a predicate *compatible*/2 is needed to check if two shifts are compatible or to get which couples of shifts are compatible. Two shifts are compatible when both are correct (working days from Monday to Friday, hours between 8 and 18 hours, and there are no repetitions of the same hour in a shift) and, in addition, when the shifts are disjoint.

```
compatible(T1,T2):-  correct_shift(T1),
                     correct_shift(T2),
                     disjoint(T1,T2).
```

But there are so many compatible combinations of shifts that it would be useful to define the concept of compatibility fuzzily instead of crisply, as defined above. This would express that two shifts are incompatible if one of them is incorrect or if they are

not disjoint, but also thatwhen they are compatible, they can be more or less compatible. They can have a level of compatibility. Two shifts will be more compatible if the working hours are concentrated (the employee has to go to work fewer days during the week). Also, two shifts will be more compatible if there are fewer free hours between the busy hours of the working days of the timetable.

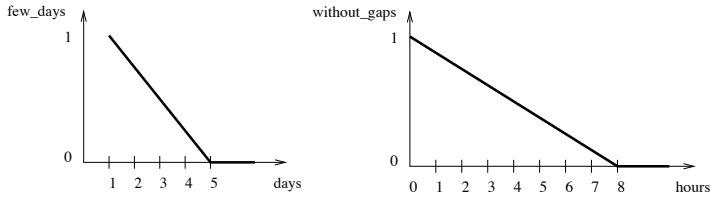


Fig. 3. Fuzzy predicates few_days/2 and without_gaps/2

Therefore, we are handling crisp concepts ($correct\_shift/1$, $disjoint/2$) and fuzzy concepts ($without\_gaps/2$, $few\_days/2$) besides. Their definitions, represented in Figure 3, are simply expressed in our language as follows:

```
few_days :# fuzzy_predicate([(1,1),(2,0.8),(3,0.6),(4,0.3),(5,0)]).

without_gaps :# fuzzy_predicate([(0,1),(1,0.8),(5,0.3),(7,0.1),(8,0)]).
```

A simple implementation combining both types of predicates could be:

```
compatible(T1,T2):~ min  correct_shift(T1),
                         correct_shift(T2),
                         disjoint(T1,T2),
                         append(T1,T2,T),
                         number_of_days(T,D),
                         few_days(D),
                         number_of_free_hours(T,H),
                         without_gaps(H).
```

Whereas $append/3$ gives the total weekly 8-hour timetable as a result of joining two shifts, $number\_of\_days/3$ outputs the total number of working days of a weekly timetable and $number\_of\_free\_hours/2$ returns the number of free one-hour gaps in the weekly timetable of working days. Looking at the timetables in Figure 2. We can get the compatibility between the two shifts, T1 and T2, represented in each timetable using the query $compatible(T1,T2,V)$. The result is $V = 0.2$ for timetable 1, $V = 0.6$ for timetable 2, and $V = 0$ for timetable 3, (because the shifts are incompatible).

5 Constructive answers

Queries of the truth value of a ground argument are important and useful, but the power of a Fuzzy Prolog is more operational, because we can get constructive answers.

Going back to the simple example of $f\_student/2$, we can ask not only whether or not a person is a student but who the students are, as we shall see at the end of section 5. And this constructive development of fuzzified crisp predicates is transmitted to fuzzy predicates defined using these crisp predicates in the body of their clauses. For example:

```
?- teenage_student(X,1).
X = john

?- teenage_student(X,0).
X = susan
```

```
?- teenage_student(X,V).
V .=. 1, X = john ? ;
V .=. 0, X = susan ? ;
V .=. 0, X = nick ? ;
no
```

We can get constructive solutions to a query where we have constraints on the truth value. The problem arises when we try to get constructive information for a query where the truth value is zero. We often want to know which people are not students or which clients give a bank no profits or, generally, which elements, X, do not satisfy a property, p, at all; i.e. the goal $p(X,0)$. If the predicate for which we are asking, $p/2$ in this case, is defined using crisp predicates in its body, $q/1$ for example, then the respective fuzzified predicate, $f\_q(X,0)$ in this case, is going to be queried. The problem with this is that none of the implementations that we have proposed in section 4.1 are able to provide any constructive result. They include cuts in their implementations, and their running, is unsound in Prolog. However, in our Fuzzy System the solutions are sound, because we return only correct results. Also, when we get "no", it means that the system does not know the answer, because it has no information for the query. The problem is that programs do have enough information to provide a result and we would like to get a sound and complete answer even for queries including zero truth values.

The solution is to properly implement the fuzzification of a crisp predicate, instead of using unsound Prolog tools. The sound formal implementation for our example will be:

$$f\_student(X,1) \leftarrow \quad student(X)$$
$$f\_student(X,0) \leftarrow \neg(student(X))$$

where $\neg/1$ is the logic negation that would provide the values of the variables that do not satisfy the predicates that are the argument of the negation in Prolog, $student/1$ in this case. This leads us to the old and complicated problem of how to implement logic negation in Prolog.

5.1 Negation and Implementation

The fundamental idea behind Logic Programming is to use a computable subset of logic as a programming language. Probably, negation is the most significant aspect of logic that was not included in the original design of Logic Programming. This is due to the fact that dealing with negation involves significant additional complexity. However, negation has an important role in knowledge representation, where many of its uses cannot be simulated by positive programs. Declarative modeling of problem specifications typically also include negative as well as positive characteristics of the domain of the problem. Negation is also useful in the management of databases, program composition, manipulation and transformation, default reasoning, etc.

The perceived importance of negation has resulted in significant research and the proposal of many alternative ways to understand and incorporate negation into Logic Programming. The involved problems appears even at the semantic level and the different proposals (negation as failure, stable models, well founded semantics, explicit negation, etc.) differ not only in expressiveness but also in semantics. Presumably as a result of this, implementation aspects have received comparatively little attention. As a consequence, the negation techniques supported by current Prolog compilers are rather limited.

Recently, a negation system for Prolog has been designed and implemented to solve this problem with logic negation. In [11], the most interesting existing proposals, are systematically studied: negation as failure [4], use of delays to apply negation as failure in a secure way , intentional negation [2] and constructive negation [15]. There not exist a negation technique that offers soundness, completeness and efficiency at the same time. A combination of all the above techniques is proposed in [12]. Information provided by a static analysis of the program is used to reduce the cost of selecting among the techniques.

The implementation of these techniques has been developed in the Ciao Prolog System [3] because it has all extensions of Prolog that are needed for the implementation. Nevertheless it can be adapted to any other Prolog compiler.

In [13] we offer a predicate *neg*/1 which is an implementation of the logic negation using a strategy of combination of different techniques. The novelty of this predicate is that it manages disequality constraints and returns answers as equalities plus disequalities as we can expect when working with negation. We will use also the disequality between terms (the predicate =/= /2 that is described in [13].

For example we can obtain the following results using the well-known logic predicate *member*/2:

```
?- neg(member(X,[1,2])).
X =/= 1, X =/= 2 ?;
no
```

```
?- neg((X =/= 0,member(X,[1,2]))).
X = 0 ?;
X =/= 0, X =/= 1, X =/= 2 ?;
no
```

We have taken advantage of the fact that we have implemented Fuzzy Prolog in the Ciao Prolog System too and we have used the predicate *neg*/1 to fuzzify crisp predicates in a sound way. Thanks to this, we have been able to get constructive answers to any fuzzy query. The right implementation of our example would be:

```
f_student(X,V):-
        student(X),
        V .=. 1.
```

```
f_student(X,V):-
        neg(student(X)),
        V .=. 0.
```

Asking for people who are not students and asking for people who are not “teenage students” can be examples of negative fuzzy queries.

```
?- f_student(X,0).
X =/= john, X =/= peter
```

```
?- teenage_student(X,0).
X = nick ? ;
X =/= john, X =/= peter ? ;
no
```

This means that nobody except *john* and *peter*, are students or "teenage students". This includes *nick* and *susan*.

Example Going back to the example 4.1 regarding the compatibility of shifts in a weekly timetable. We are going to ask some questions about the shifts T1 and T2 of timetable 4 in Figure 2. One hour of T2 is not fixed yet.

We can note: the days of the week as mo, tu, we, th and fr; one-hour time slot as its starting time from 8 a.m. to 5 p.m.; one hour of the weekly timetable as a pair of day and hour and one shift as a list of 4 hours of the week.

If we want to fix the free hour of T2 in the 10-11 a.m.slot whose compatibility is not null, we get that only Tuesday is incompatible.:

```
?- compatible([(mo,9),(tu,10),(we,8),(we,9)],
              [(mo,8),(we,11),(we,12),(D,10)], V),  V .>. 0 .
D =/= tu
```

If we want to know how to complete the shift T2 given a level of compatibility higher than 70 %, we get the slice from 10 to 11 a.m. time sloth on Wednesday or Monday morning.

```
?- compatible([(mo,9),(tu,10),(we,8),(we,9)],
              [(mo,8),(we,11),(we,12),(D,H)], V),  V .>. 0.7 .
V = 0.9,  D = we, H = 10 ? ;
V = 0.75, D = mo, H = 10 ? ;
no
```

6 Conclusions and Future work

We have provided in [17] a Fuzzy Prolog Language and we have implemented it over Prolog instead of implementing a new resolution system. The way of doing this extension to Prolog is interpreting fuzzy reasoning as a set of constraints and after that translating fuzzy predicates into CLP($\mathcal{R}$) clauses. The rest of the computation is resolved by the Prolog compiler. The advantage is simplicity and a good potential for efficiency. We have also generality because we work with the definition of a general *aggregation operator* that includes any of the operators used by other approaches and we have got flexibility because new *aggregation operators* can be added with almost no effort.

In this paper we explain how we have combined crisp and fuzzy logic in a Prolog compiler. This is a great advantage because it lets us model many problems using fuzzy programs. We have obtained constructive answers even for negative queries thanks to

constructive logic negation. So we have extended the expressibility of the language and the possibility of applying it to solve real problems.

For this work we have made the decision of using the Prolog's mechanism to be able to integrate our representation of the fuzzy knowledge into a Prolog compiler with the advantages that this involves. But as future work we will change the representation of the indefinite information. In this approach we are using the Prolog failure to represent the total uncertainty, but another possibility is to consider the complete interval $[0,1]$ to represent this truth value, e.g. $v \in \{0..1\}$ or the constraint $0 \leq v \wedge v \leq 1$ that represents the whole interval of possible truth values. The advantage is that it does not stop the truth value evaluation, so if $v_j \in \{0..1\}$ then

$$V(p(\widehat{x})) = v_1 \cup ... \cup v_j \cup ... \cup v_n = v_1 \cup ... \cup v_{j-1} \cup v_{j+1} \cup ... \cup v_n$$

that is, v_j does not affect the evaluation of the truth value of $p(\widehat{x})$.

References

1. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint logic programming: syntax and semantics. In *ACM TOPLAS*, volume 23, pages 1–29, 2001.
2. P. Bruscoli, F. Levi, G. Levi, and M. C. Meo. Compilative Constructive Negation in Constraint Logic Programs. In Sophie Tyson, editor, *Proc. of the Nineteenth International Colloquium on Trees in Algebra and Programming, CAAP '94*, volume 787 of *LNCS*, pages 52–67, Berlin, 1994. Springer-Verlag.
3. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. Technical Report CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997. System and manual at `http://www.cliplab.org/Software/Ciao/`.
4. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322, New York, NY, 1978. Plenum Press.
5. J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The clp(r) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.
6. F. Klawonn and R. Kruse. A Łukasiewicz logic based Prolog. *Mathware & Soft Computing*, 1(1):5–29, 1994.
7. E.P. Klement, R. Mesiar, and E. Pap. Triangular norms. Kluwer Academic Publishers.
8. J. Medina, M. Ojeda-Aciego, and P. Votjas. A completeness theorem for multi-adjoint logic programming. In *International Fuzzy Systems Conference*, pages 1031–1034. IEEE, 2001.
9. J. Medina, M. Ojeda-Aciego, and P. Votjas. Multi-adjoint logic programming with continous semantics. In *LPNMR*, volume 2173 of *LNCS*, pages 351–364, Boston, MA (USA), 2001. Springer-Verlag.
10. J. Medina, M. Ojeda-Aciego, and P. Votjas. A procedural semantics for multi-adjoint logic programming. In *EPIA*, volume 2258 of *LNCS*, pages 290–297, Boston, MA (USA), 2001. Springer-Verlag.
11. S. Muñoz and J. J. Moreno-Navarro. How to incorporate negation in a prolog compiler. In E. Pontelli and V. Santos Costa, editors, *2nd International Workshop PADL'2000*, volume 1753 of *LNCS*, pages 124–140, Boston, MA (USA), 2000. Springer-Verlag.
12. S. Muñoz and J. J. Moreno-Navarro. Intelligent agent to implement logic negation. In J. L. Perez de la Cruz and J. Pavon, editors, *4th Iberoamerican Workshop on Multi-Agent Systems, IBERAGENTS'02*, Malaga (Spain), 2002.

13. S. Muñoz, J. J. Moreno-Navarro, and M. Hermenegildo. Efficient negation using abstract interpretation. In R. Nieuwenhuis and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning*, number 2250 in LNAI, pages 485–494, La Habana (Cuba), 2001. LPAR 2001.
14. A. Pradera, E. Trillas, and T. Calvo. A general class of triangular norm-based aggregation operators: quasi-linear t-s operators. *International Journal of Approximate Reasoning*, 30(1):57–72, 2002.
15. P. Stuckey. Constructive negation for constraint logic programming. In *Proc. IEEE Symp. on Logic in Computer Science*, volume 660. IEEE Comp. Soc. Press, 1991.
16. E. Trillas, S. Cubillo, and J. L. Castro. Conjunction and disjunction on $([0,1], <=)$. *Fuzzy Sets and Systems*, 72:155–165, 1995.
17. C. Vaucheret, S. Guadarrama, and S. Muñoz. Fuzzy prolog: A simple general implementation using clp(r). In M. Baaz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2002*, number 2514 in LNAI, pages 450–463, Tbilisi, Georgia, October 2002. Springer-Verlag.
18. P. Vojtas. Fuzzy logic programming. *Fuzzy sets and systems*, 124(1):361–370, 2001.
19. Ehud Y. and Shapiro. Logic programs with uncertainties: A tool for implementing rule-based systems. In *IJCAI*, pages 529–532, 1983.