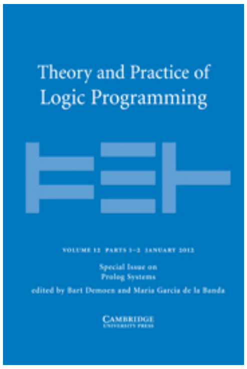


Theory and Practice of Logic Programming
VOLUME 12 PARTS 1–2 JANUARY 2012
Special Issue on Prolog Systems
edited by Bart Demoen and Maria Garcia de la Banda
CAMBRIDGE UNIVERSITY PRESS

SWI-Prolog

JAN WIELEMAKER, TOM SCHRIJVERS, MARKUS TRISKA and TORBJÖRN LAGER

Theory and Practice of Logic Programming / Volume 12 / Special Issue 1-2 / January 2012, pp 67 - 96
DOI: 10.1017/S1471068411000494, Published online: 12 September 2011

Link to this article: http://journals.cambridge.org/abstract_S1471068411000494

How to cite this article:
JAN WIELEMAKER, TOM SCHRIJVERS, MARKUS TRISKA and TORBJÖRN LAGER (2012). SWI-Prolog. Theory and Practice of Logic Programming, 12, pp 67-96 doi:10.1017/S1471068411000494

TLP **12** (1–2): 67–96, 2012. © Cambridge University Press 2011
doi:10.1017/S1471068411000494 First published online 12 September 2011

SWI-Prolog

JAN WIELEMAKER

Faculty of Sciences, Computer Science, VU University Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands
(*e-mail:* `J.Wielemaker@cs.vu.nl`)

TOM SCHRIJVERS

Department of Applied Mathematics and Computer Science, Ghent University, Krijgslaan 281 S9, 9000 Gent, Belgium
(*e-mail:* `Tom.Schrijvers@ugent.be`)

MARKUS TRISKA

Institut für Informationssysteme 184/2, Technische Universität Wien, Abteilung für Datenbanken und Artificial Intelligence, Favoritenstraße 9, A-1040 Wien, Austria
(*e-mail:* `triska@dbai.tuwien.ac.at`)

TORBJÖRN LAGER

Department of Philosophy, Linguistics and Theory of Science, University of Gothenburg, Box 200, S-40530 Göteborg
(*e-mail:* `lager@ling.gu.se`)

submitted 5 October 2009; revised 24 February 2010; accepted 22 November 2010

Abstract

SWI-Prolog is neither a commercial Prolog system nor a purely academic enterprise, but increasingly a community project. The core system has been shaped to its current form while being used as a tool for building research prototypes, primarily for *knowledge-intensive* and *interactive* systems. Community contributions have added several interfaces and the constraint (CLP) libraries. Commercial involvement has created the initial garbage collector, added several interfaces and two development tools: PlDoc (a literate programming documentation system) and PlUnit (a unit testing environment).

In this article, we present SWI-Prolog as an integrating tool, supporting a wide range of ideas developed in the Prolog community and acting as glue between *foreign* resources. This article itself is the glue between technical articles on SWI-Prolog, providing context and experience in applying them over a longer period.

KEYWORDS: Prolog, logic programming system

1 Introduction

SWI-Prolog was started as a recreational program for personal understanding and enjoyment after reading Bowen et al. (1983). This toy landed in fertile soil. Our university department was involved in Shelley (Anjewierden et al. 1990) a workbench for knowledge engineering that was to be implemented using Quintus

Prolog (version 1) and PCE, an in-house developed object-oriented graphical library written in C. Frustrated by the inability of Quintus Prolog version 1 to make recursive calls between C and Prolog, which seriously complicated our integration between Prolog and the graphical system, we demonstrated the much cleaner integration achievable with the small SWI-Prolog system.

While SWI-Prolog was clearly inferior to Quintus in terms of robustness and execution speed, it quickly replaced Quintus, not only at our site, but also at the two overseas sites where modules for the workbench were being developed. Why did this happen? We think this was due to three small, yet much valued features: (1) the **make/0** predicate that reloads all files that have been edited since last loaded (see Section 3.2); (2) the ability to run **make/0** on a restarted saved state dramatically reduced the application restart-time; (3) using the *auto-loader* the initial image could be restricted to the core parts of the application, which again reduced memory usage and startup times. The faster and bi-directional interface to the graphics libraries made the application much more responsive for typical interaction.

Because the market for commercial Prolog systems was by then (late 1980s) already crowded, we decided to make the system available for free to the community through anonymous ftp. Initially, this did not create a *developer* community, but it did create a *user* community. The user community consisted mostly of universities that used this small, free, and portable Prolog system for education. This unforeseen development was strengthened when SWI-Prolog was ported to MS-Windows as part of another research project.

SWI-Prolog's development has been guided by internal projects, external (including commercial) users, and increasingly by developers from the community. This article summarizes the distinguishing features. Because most of the technical details are dealt with in other articles, we concentrate on providing an overview, motivating our decisions and describing our experiences.

In Section 2, we describe the development of SWI-Prolog, with a particular emphasis on how it was embedded in research projects. The paper continues with a user-oriented description, covering the environment (Section 3), the constraint libraries (Section 4), interfaces to the outside world (Section 5), and finally web-applications (Section 6). The next part of the paper targets the Prolog developers community and addresses the language properties and regression testing. We end with a brief discussion and a description of future work.

2 Developing Prolog in the context of applications

In the 1990s, before SWI-Prolog attracted a wider network of developers, it took form in the SWI department at the University of Amsterdam. There, it was used as an in-house (or *in-project*) tool for the development of proof-of-concept software prototypes, rather than an objective in its own right. This supporting role has very much influenced both the development process and design decisions of SWI-Prolog. New architectures, features, and deployment strategies for SWI-Prolog were explored in the course of new research projects and as demanded by every new prototype. Lessons learned influenced subsequent extensions of the core and its libraries.

2.1 Using Prolog as glue

SWI-Prolog's supporting role in the academic research projects is primarily to function as a tool for rapid development. It provides a uniform programming environment for accessing a range of resources. Using multiple environments requires interfaces that are generally hard to program due to differences in datatypes, control (e.g., Prolog non-determinism), and organization (e.g., object-oriented versus functional).

The overall approach we have followed over the years is to bring the required resources to Prolog, either as Prolog resources (e.g., our HTTP client and server libraries) or as encapsulated foreign resources (e.g., our RDF store, Wielemaker et al. (2003)). The latter makes up for the wealth of resources that are not natively available in Prolog, either for lack of development time or because Prolog is not a suitable language (e.g., libraries that require significant destructively updatable state such as a graphics library).

In general, the way to deal with the complexity of an environment that offers many different resources is to define stable and, preferably, small interfaces. While, in our opinion, this is a boon in large top-down designed software systems, it does form a significant burden for developing the medium-scale applications (e.g., 50,000 lines) that we target. Our projects consist of small teams (two to seven programmers) that quickly evolve interfaces and incorporate new insights.

2.2 Application-driven requirements

SWI-Prolog's supporting role in the application-oriented research projects helped setting its main requirements:

- In its role as uniform platform, the system must be able to encapsulate foreign resources in a flexible and transparent way.
- To support rapid and incremental development, the system must load large programs rapidly. After editing, a program must be synchronized quickly without losing state stored in dynamic predicates and foreign resources.
- Decent tools are essential to facilitate (rapid) development. In particular, it provides good editor support, a source-level debugger, an execution profiler, and a cross-referencer.
- In order to qualify as a web-based application platform, SWI-Prolog must run as a server, 24×7. As a server, it must be stable and free of resource leaks. Moreover, to provide scalable request handling, it must exploit concurrent (multi-core) hardware.
- Reflexiveness is desirable for application-specific program analysis and transformation, and supports the debugger. Regardless of the compilation mode, good debugger support and the ability to inspect code (e.g., **listing/1**) should be provided .
- In order to support all major desktop and server systems for applications, SWI-Prolog should be portable. Currently, SWI-Prolog is portable across

platforms that provide a C99-compliant C-compiler and implement either the POSIX/X11 or the Win32/Win64 APIs.

Main Application Influences The directly supported research prototypes were (and in part still are) interactive applications for the management of knowledge models through graphical user interfaces. This has prominently resulted in both the XPCE library and the early adoption of RDF.

- XPCE (Wielemaker and Anjewierden 2002) provides an—for those days—advanced and tightly integrated object-oriented framework for the development of graphical user interfaces.
- RDF (Resource Description Format, Lassila and Swick 1999) provides a widely accepted and extensible infrastructure for representing knowledge. RDF also facilitates exchange of knowledge-bases between SWI-Prolog and external tools.

Support for networking and subsequently concurrency started when we used Prolog programs as an "intelligent agent" in an FIPA-based agent framework (Bellifemine et al. 2001). After a while, we realized that our job would become much easier if Prolog also provided support for concurrency. Support for HTML was added when we used Prolog to partition text into logical segments and classify these segments with terms from an ontology. It also turned out that full support for Unicode is necessary to deal with character-entities in HTML.

Both HTML and HTTP support went through a number of iterations, where the code was part of projects, but not of the SWI-Prolog libraries. The core infrastructure for these and many other extensions reached the system libraries in the MultimediaN project around 2005 (see Section 6).

Often one thing leads to another. To support good performance for arbitrary reasoning patterns, our RDF infrastructure has to be main-memory based. An unfortunate consequence is that applications with significant amounts of data take relatively long to start. In order to use such applications effectively as web services and avoid a prohibitive start-up cost at every request, the application has to be able to run continuously. Moreover, to promptly serve simultaneous clients, such applications must also be concurrent.

3 The SWI-Prolog development environment

The development environment is a crucial part of a Prolog system that aims at prototyping large applications. SWI-Prolog's user-friendliness stems from three sources: (1) command-line interaction, (2) graphical tools, and (3) design decisions for the compiler and extensions to the language. The last category is described in Section 7. In this section, we take a closer look at the command-line and graphical tools.

3.1 Prolog top-level interaction

Originally, the top-level interaction of SWI-Prolog was based on the Edinburgh tradition, prompting for alternatives if and only if the query contains variables and

```
1 ?- [library(clpfd)].
% library(clpfd) compiled into clpfd 0.12 sec, 612,984 bytes
true.                   % command: deterministic: no prompt

2 ?- A is 1+1.
A = 2.                  % deterministic: does not prompt

3 ?- member(x, [a,x,y]).
true ;                  % prompts: non-deterministic success
false.

4 ?- A #> 10.
A in 11..sup.           % CLP(FD): deterministic constraint
```

Fig. 1. Top-level interaction in SWI-Prolog.

printing *Yes* or *No*, the only small difference being that user replies were processed on single-keystrokes (i.e., without using "return"). This approach suffers from three problems: (1) the top-level syntax was not suitable for copy/paste into the top level, (2) there is no way to deal with non-deterministic goals that have no variables, and (3) there is no clean way to represent residual constraints. We have revised the SWI-Prolog top level based on one simple principle: "The answer substitution is a valid Prolog goal that returns the same answers as the original query." Starting from this principle, the rest follows naturally. Figure 1 illustrates some typical cases.

- An answer substitution is a conjunction of equalities of the form *Var* = *Value*. If there are no variables, the answer is simply `true`. (1 in Fig. 1).
- If variables in the answer carry constraints, **copy_term/3** is used to create a copy without constraints and goals to reinstate the constraints. These goals are printed after the variable-bindings. (4 in Fig. 1).
- A query that succeeds deterministically writes its answer substitution followed by a full-stop and prompts immediately for the next query. (2 in Fig. 1).
- A query that fails writes `false.` The predicate **false/0** is a built-in.
- A query that succeeds non-deterministically waits at the end of the printed answer substitution. If the user types ';', this is echoed and the system returns the next substitution. (3 in Fig. 1). If the user hits RETURN, the system prints a full-stop.

3.1.1 Command line editing

During system development, developers spend a considerable amount of time entering commands, specially writing test-queries to assess correctness for parts of the application being developed. SWI-Prolog provides the following features to support this development mode:

- *Using (GNU-)readline for the top-level input*
  Completion of the library is extended with completion on alpha-numerical atoms which enables faster input of long predicate identifiers and atomic arguments, as well as inspection of the possible alternatives using Alt-?. The completion algorithm uses the built-in completion of file names if no atom matches.

- *Command line history*
  SWI-Prolog provides a history facility resembling the corresponding facilities in the Unix `csh` and `bash` shells. Viewing the list of executed commands (using `?- h.`) is a particularly valuable feature.
- *Top-level bindings*
  The facility to reuse answer substitutions through copy/paste is useful, but limited to bindings that have been printed recently, have not been modified using the **portray/1** hook, and are short. For this reason, SWI-Prolog stores the variable-bindings from top-level queries in the database under the name of the used variable. Top-level query expansion replaces terms of the form \$*Var* (\$ is a prefix operator) with the last recorded binding for this variable. New bindings due to backtracking or new queries overwrite the old value. Typical example: by using `$X`, the user avoids typing or copy/paste of the object reference returned by a call to XPCE:

```
1 ?- new(X, picture).
X = @12946012/picture.  % /picture is added by portray/1

2 ?- send($X, open).
true.
```

3.2 Supporting the edit cycle

There are two simple but 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 with information about every loaded file: the pathname of the file, the time of the most recent modification of the file (time stamp) that was valid when the file was loaded, and the context module from which it was loaded. The **make/0** predicate checks whether the modification time of any of the loaded files has changed and reloads these file into the proper module contexts. After updating the running program, **make/0** lists undefined predicates as described in Section 3.3.

edit(*+Specifier***)**
Finds all entities with their location that match *specifier*. If there are multiple entities related to different source files, then it asks the user for the desired one and calls the user-defined editor, placing the cursor at the location of the selected entity. The predicate searches for (loaded) files, predicates, and modules. The interface can be customized in two ways: by extending the entities searched for (e.g., XPCE classes, see Section 5), and by changing the editor that is called. 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
```

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

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:standard/4, which is referenced by
Warning:        chat:i_adj/9 at /home/jan/lib/prolog/chat/slots.pl:128
...
```

Fig. 2. Using **list_undefined/0** on chat 80 wrapped into the module `chat`. To save space, only the first of the nine reported warnings is included.

3.2.1 DWIM: Do What I Mean

DWIM (*Do What I Mean*) is implemented at the top level to quickly fix mistakes and allow for under-specified queries. It corrects the following errors: simple spelling errors, different word-order (e.g., *exists_file* matches *file_exists*), arity mismatches, and wrong module.

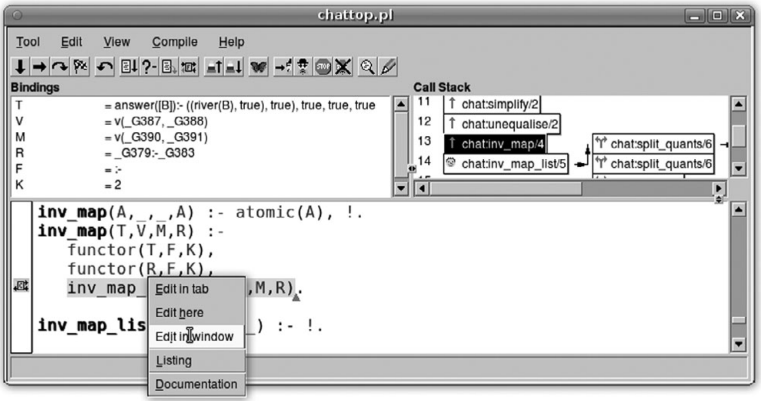
DWIM is used in three areas. First, queries typed at the top level are checked, and if there is a unique correction the system prompts the user whether the corrected query is to be executed instead of the original one. Especially adding the module specifier improves interaction from the top level when using modules. If there is no unique correction, the system reports all close candidates. Second, Predicates such as **spy/1** act on the named predicate in any module if the module is omitted. Third, if a predicate existence error is not caught, the DWIM system is activated to report likely candidates.

3.3 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 shows the partial output of running **list_undefined/0** on the *chat 80* (Pereira and Shieber 1987) program.

3.4 Printing log messages

The library *debug* is a lightweight infrastructure that handles printing debugging messages (logging) and assertions. Each debug message is associated with a *Topic*. A *Topic* is an arbitrary Prolog term that identifies a class of debug messages. Using compound terms such as `http(connection)` and `http(query)`, all debugging messages related to HTTP can be enabled with `?- debug(http(_))`.



```
T    = answer([B]):- ((river(B), true), true), true, true, true
V    = v(_G387, _G388)
M    = v(_G390, _G391)
R    = _G379:-_G383
F    = :-
K    = 2
```

```
11  chat:simplify/2
12  chat:unequalise/2
13  chat:inv_map/4          chat:split_quants/6
14  chat:inv_map_list/5     chat:split_quants/6
```

```
inv_map(A,_,_,A) :- atomic(A), !.
inv_map(T,V,M,R) :-
   functor(T,F,K),
   functor(R,F,K),
   inv_map_         ,M,R).

inv_map_lis          _) :- !.
```

Fig. 3. The Source-level Debugger. The source code is rendered using an embedded version of PceEmacs.

debug(+*Topic*, +*Format*, +*Arguments***)**
  Prints a message using **format**(*Format*, *Arguments*) if *Topic* unifies with a topic enabled with **debug/1**.

debug/nodebug(+*Topic* [>*file*]**)**
  Enables/disables messages for which *Topic* unifies. If >*file* is added, the debug messages are appended to the given file.

assertion(:*Goal***)**
  Assumes that *Goal* is true. Prints a stack-dump and traps to the debugger otherwise. This facility is derived from the **assert()** macro as used in C, renamed for obvious reasons.

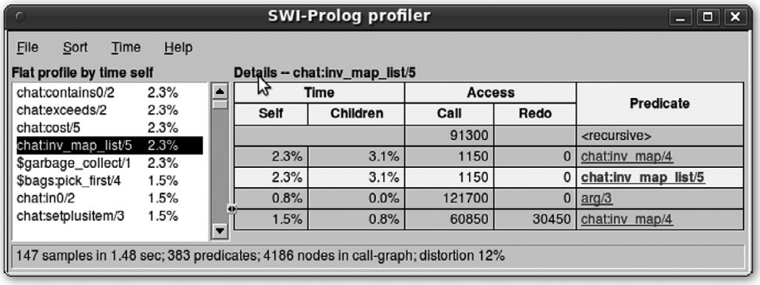
Calls to **debug/3** and **assertion/1** are replaced with `true` using *goal-expansion* if optimization is enabled.

3.5 The built-in editor

PceEmacs is an Emacs clone written in XPCE/Prolog. It has full access to the application by means of the reflexive capabilities of Prolog. On each key-stroke, Prolog opens the edit buffer as a stream and tries to read the current clause. If there is a syntax error, it displays unobtrusive information about the location. If the syntax is valid, the clause is colored based on information from the latest cross-reference analysis. Goals are given a menu that provides access to the source, documentation, and listing. Singleton variables are highlighted. If the cursor appears inside the name of a variable, all other occurrences of this variable in the clause are underlined. Whenever the user pauses for 2 seconds, Prolog opens the edit buffer as a stream and performs a full cross-reference of the edit buffer. Figure 3 shows PceEmacs embedded in the debugger.

3.6 The source-level debugger

The hook-predicate **prolog_trace_interception**(+*Port*, +*Frame*, +*Choice*, -*Action*) can be implemented to realize an alternative debugger such as the source-level debugger



Flat profile by time self

chat:contains0/2	2.3%
chat:exceeds/2	2.3%
chat:cost/5	2.3%
chat:inv_map_list/5	2.3%
$garbage_collect/1	2.3%
$bags:pick_first/4	1.5%
chat:in0/2	1.5%
chat:setplusitem/3	1.5%

Details -- chat:inv_map_list/5

Time		Access		Predicate
Self	Children	Call	Redo	
		91300		<recursive>
2.3%	3.1%	1150	0	chat:inv_map/4
2.3%	3.1%	1150	0	**chat:inv_map_list/5**
0.8%	0.0%	121700	0	arg/3
1.5%	0.8%	60850	30450	chat:inv_map/4

147 samples in 1.48 sec; 383 predicates; 4186 nodes in call-graph; distortion 12%

Fig. 4. The Profiler showing information about CHAT80. To profile a goal, run it using `?- profile(Goal).`

described below. The source-level debugger provides three views (Figure 3):

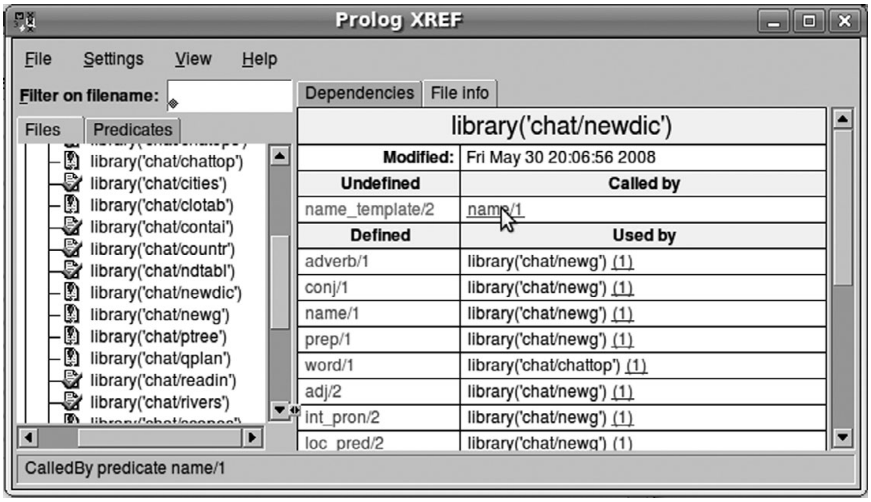
- *The source*
  An embedded *PceEmacs* (see Section 3.5) shows the current location, indicating the current port using color and icons. *PceEmacs* also allows setting a *breakpoint* on an arbitrary location in a clause. Breakpoints are realized by replacing a virtual machine instruction with a *break* 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 names and current bindings in the top-left window (see Section 7.1). The representation of values can be changed using the familiar **portray/1** hook. Double-clicking the displayed value of a variable opens a separate window showing the variable binding with additional layout to clarify the structure of a term.
- *The stack*
  The top-right window shows the recursion stack as well as the recent outstanding choicepoints. Any node can be selected to examine the context of that node. The stack window allows one to quickly examine choicepoints left after a goal succeeds. Clicking a choice-point shows the clause that last succeeded. Using the *up* command shows the source of the calling context.

3.7 Execution profiler

The *Execution Profiler* builds a call-tree at runtime and counts the number of calls and redos for each node in this call-tree. The time spent in each node is established using stochastic sampling. Prolog primitives are provided to extract all information from the recorded call-tree. A graphical Prolog profiling tool presents the information interactively, similarly to the GNU gprof (Graham et al. 1982) tool (see Figure 4).

3.8 Graphical cross-referencer

Figure 5 shows the output of running **gxref/0**, which shows the dependencies between source files based on cross-reference analysis for all files loaded into the running



library('chat/chattop')
library('chat/cities')
library('chat/clotab')
library('chat/contai')
library('chat/countr')
library('chat/ndtabl')
library('chat/newdic')
library('chat/newg')
library('chat/ptree')
library('chat/qplan')
library('chat/readin')
library('chat/rivers')

library('chat/newdic')

Modified:	Fri May 30 20:06:56 2008
Undefined	**Called by**
name_template/2	name/1
Defined	**Used by**
adverb/1	library('chat/newg') (1)
conj/1	library('chat/newg') (1)
name/1	library('chat/newg') (1)
prep/1	library('chat/newg') (1)
word/1	library('chat/chattop') (1)
adj/2	library('chat/newg') (1)
int_pron/2	library('chat/newg') (1)
loc_pred/2	library('chat/newg') (1)

CalledBy predicate name/1

Fig. 5. The cross-referencer.

system. The analysis is provided by a separate public library called `prolog_xref.pl`, which is also used by PceEmacs. The (red) exclamation-mark indicates that there is at least one warning in the file or directory. In addition to the functionality exemplified by Figure 5, the tool can show the dependencies between sources as a graph and it can generate module declarations to help transform non-modular code into modular code.

3.9 Discussion

Many of the current tools are built in SWI-Prolog's proprietary graphics system (XPCE), though the underlying computation for coloring, cross-referencing, profiling, and tracing is accessible from Prolog through public APIs. In an ideal world, these tools would be neatly integrated in open IDEs such as Eclipse. In practice, such an integration is hard to achieve. Bendisposto et al. (2009) have implemented a Prolog parser in Java for this purpose. Still, the result may be incorrect, e.g., because the IDE does not know which operators are visible.

4 Constraint libraries

Constraint Logic Programming functionality came rather late in the lifetime of SWI-Prolog, because it lacked the basic support. This changed early in 2004 when *attributed variables* were added to the language (see Section 7.5). The Leuven CHR library was then the first CLP library to be ported to SWI-Prolog. Later came a port of Christian Holzbaur's CLP($\mathbb{Q}\mathbb{R}$) library, and a finite domain CLP(FD) solver. Finally, we mention SWI-Prolog's INCLP($\mathbb{R}$) library (De Koninck et al. 2006), which provides non-linear constraints over the reals, and was implemented on top of CHR.

4.1 CHR

The Constraint Handling Rules (CHR) language was created about 20 years ago (Frühwirth 1998). Since then, CHR has proved its merit as a powerful complement to Prolog. Both have a firm basis in logic, but whereas Prolog is about single-headed rules, backward chaining, and backtracking, CHR has multi-headed rules, forward chaining, and committed choice.

CHR's features support the modeling and implementation of constraint solvers. CHR has also turned out to be very useful for applications of (Term) Rewriting Systems and Production Rule Systems, as well as for expressing imperative algorithms in a high-level manner.

CHR is usually embedded in Prolog as an add-on library and for a long time the SICStus implementation by Christian Holzbaur (Holzbaur and Frühwirth 2000) has been the standard implementation. This library was also available in YAP (da Silva and Costa 2007); two different, older implementations come with ECLiPSe (Wallace and Schimpf 1999). Neither of these constraint systems was under active development in the last decade. In the last 5 years, the K.U. Leuven CHR system (Schrijvers and Demoen 2004) has come to replace these older systems.

4.1.1 The K.U. Leuven CHR system

The K.U. Leuven CHR system started out as a small practical project, to obtain a benchmark for Bart Demoen's new implementation of dynamic attributed variables for hProlog (Demoen 2002). The output of SICStus' CHR was reverse engineered to obtain a base system. It became clear very soon that there was much potential for improving the system, and gradually the system diverged from its roots. It became the central topic of Tom Schrijvers' Ph.D. thesis (Schrijvers 2005), and the starting point for subsequent theses by Jon Sneyers, Leslie De Koninck, and Peter Van Weert.

In July 2004, the system was ported to XSB and integrated with tabled execution (Schrijvers and Warren 2004). Half a year later, Jan Wielemaker was looking for an easy way to provide general constraint solving capabilities to SWI-Prolog. The smoothness of the port to XSB convinced him that the K.U. Leuven CHR system was what SWI-Prolog needed (Schrijvers et al. 2005). From then on, SWI-Prolog has been the K.U. Leuven CHR system's main supported platform, and Jan Wielemaker has taken care of tight integration with the rest of the system, notably with the debugger. Gradually, other open source Prolog systems became interested in this new CHR system, and today you can find K.U. Leuven CHR in hProlog, XSB, SWI-Prolog, YAP, Ciao, B-Prolog, and SICStus. Simplifying the porting process of the CHR library remains one of the key challenges. Standardization of the core language features in K.U. Leuven CHR would be a good step in this direction.

4.2 CLP(FD)

Finite domain constraint solvers are almost a standard component in modern Prolog environments. SWI-Prolog's solver is implemented completely in Prolog. We aimed at providing a solver that is very reliable, rather than exceedingly fast. Users can easily modify and extend the solver by following a few simple conventions that are explained in the solver's user manual. Because finite domain constraints are also used in introductory Prolog courses, we have implemented several features that make SWI-Prolog's finite domain constraint solver suitable as a teaching aid for beginners. In the following subsections, we discuss these features in more detail.

4.2.1 Extending traditional finite domain solvers

The need for arbitrary precision integer arithmetic is widely recognized, and many common Prolog systems provide transparent built-in support for arbitrarily large integers.

It thus seemed natural to enhance a constraint solver over finite domains with the ability to reason over arbitrarily large integers. SICStus Prolog already goes in that direction, using the symbolic constants `inf` and `sup` to denote default domain limits, but internally, they still correspond to quite small integers – the system yields *representation errors* when these limits are exceeded.

We have implemented a new constraint solver over finite domains, in which big integers are transparently used. We accept the `inf/sup` notation of SICStus Prolog, but these atoms now denote the actual infinities instead of abbreviating underlying finite limits.

4.2.2 Ensuring terminating propagation

By allowing unbounded domains, we gain expressibility at the price of potentially non-terminating propagation. For example, queries like `X#>abs(X)` or `X#>Y,Y#>X, X#>=0` do not terminate in many existing constraint solvers. Triska et al. (2009) describe how to guarantee terminating propagation.

4.2.3 Uniform arithmetic

Prolog's built-in arithmetic predicates are *moded* – at evaluation time, expressions must be ground. Finite domain constraint solvers remove this restriction. A significant generalization was achieved in 1996 with the first release of a finite domain system for SICStus Prolog. The equality relation `#=/2` could now be used in place of **is/2** for integers. In our solver, we generalize this to all constraints, such as `#=</2` and `#>/2`, which can be used instead of the built-ins at all places. At compile time, these constraints are specialized to fall back to moded built-in arithmetic, reducing the overhead of using CLP(FD) over native arithmetic.

5 SWI-Prolog interface libraries

As mentioned in Section 2.1, we believe that entire applications should be written in a single language, and that Prolog is well-suited to the task. To do this, we have to provide support for document formats, protocols, etc., from Prolog. This is opposite to the position taken, e.g., by the developers of Amzi! Prolog + Logic Server.[1] In Amzi!'s view, a logic program is comparable to a database and accessed from

[1] `http://www.amzi.com`

procedural languages: "Amzi! moves you toward a unique view of its positioning in the Prolog market. It aims to be a component of an application written in other languages."[2]

Using Prolog for what it is good at and embedding it in a conventional procedural environment has clear advantages, because it does not require so many developers familiar with Prolog, and makes it possible to implement a large part of the system in accordance with "industrial standards." However, we believe it is not the most productive approach for a large class of projects. Accessing Prolog from an imperative language as a (logical) database engine suffers from what is known as the "object/relational impedance mismatch" (Ambler 2002; Oren et al. 2007, Section 6.1). However, Prolog can provide natural APIs to web document formats (HTML, XML), relational databases, and, especially, schema-less semantic web data (RDF). In addition, embedding Prolog in traditional procedural languages makes interactive development more difficult, while careful encapsulation of software developed in other languages preserves most of the interactive development features.

Our experience shows that embedding Prolog in modern environments (such as Java or .NET) is particularly painful. Such environments typically provide threads, automatic memory management using garbage collection, and (in POSIX systems) signal handling. Although the C-interface (Section 7.7) does provide primitives to manage Prolog threads, proper management of resources is far from simple. Creating and destroying an instance of a Prolog engine for each call to Prolog is generally too expensive. The API also allows for using a pool of Prolog engines, but allocating appropriate resources to this pool is a non-trivial problem. Synchronizing object lifetimes for objects that are referenced from Prolog is complicated. POSIX defines process-global asynchronous signal handling, which is used by both JVMs and the Prolog engine. In short, it is difficult to combine Prolog with such languages within a single process. Debugging interface problems is particularly hard.

More promising interaction is achieved by using network-based communication mechanisms, such as those provided by InterProlog (Calejo 2004) or HTTP. However, communication overhead is then much larger, which limits the usability of such an approach. But such separation also has significant advantages: it becomes much easier to isolate and locate problems; moreover, if various services are provided by separate Prolog threads, one can still carry out traditional interactive development. See also Section 6.

5.1 Provided interfaces

This section provides a brief overview of the external interfaces that are supported by SWI-Prolog. We distinguish two types of interfaces: document formats that can be read, written, and processed (e.g., XML) and supported protocols (e.g., HTTP).

XML, SGML, and HTML (SWI-Prolog and the web, Wielemaker et al. 2008)
These core languages of the web are supported through a C parser library

[2] PC AI Review, September/October '95.

that is also used by XSB Prolog. The library works in two modes: parsing a document into a ground Prolog term and using call-backs (the *event model*). Beside parsing, we provide a library called `html_write.pl` that is used to output HTML. The library provides a concise and extensible mechanism for producing syntactically correct (X)HTML, including a modular mechanism for managing required JavaScript and CSS resources.

RDF (Wielemaker et al. 2003) The RDF support consists of parsers and writers for the RDF/XML and Turtle serializations of the RDF data model, and an RDF-storage module that is written in C and designed to be tightly connected to Prolog. The storage module provides fully indexed lookup, statistics to support a query optimizer, reliable persistent storage, transaction management, and full-text search.

JSON (see also Section 6.1) JavaScript Object Notation is a popular serialization format for structured data.

HTTP (SWI-Prolog and the web, Wielemaker et al. 2008) Supports both clients and servers. We currently see the HTTP server as one of the fundamental libraries. See Section 6 for an example server.

ODBC Provides low-level access to ODBC databases. SWI-Prolog still lacks high-level support, such as the one described by Draxler (1991).

TCP, UDP, SSL, TIPC These libraries provide basic network communication.

XPCE (Wielemaker and Anjewierden 2002) XPCE provides native and portable (X11 and MS-Windows) graphics. As described in the introduction, XPCE was part of SWI-Prolog from the beginning. It is still the basis for many applications, including the development tools described in Section 3.

6 Prolog as a web server

The web has become our most important application domain for SWI-Prolog. A significant part of the interfaces described in Section 5 have been influenced by the development of ClioPatria (*Thesaurus-based search in large heterogeneous collections*, Wielemaker et al. 2008). The user interface of ClioPatria is based on AJAX using the YUI Widget set. In Section 6.1, we describe the implementation of a small interactive web application that uses AJAX/YUI. We present our experiences with hosting SWI-Prolog in Prolog in Section 6.2

6.1 AJAX N-Queens – how to web-enable Prolog

When trying to adapt Prolog to the functioning of the Web, we encounter what is often referred to as an “impedance mismatch problem”: Prolog is relational in that a query may map to more than one result, but HTTP is essentially functional in that one query/request should map to exactly one result/response. Sometimes, this can be solved by using **findall/3**, but this only works for a finite number of solutions and only if there are not too many. Besides, we may *prefer* to generate and present the solutions “a-tuple-at-a-time,”, sometimes because it is much cheaper in terms of memory requirements on both server and client, and sometimes because we want to be able to decide, after having seen the first couple of solutions, whether we want to see more.

Instead of wrapping queries in **findall/3**, we choose to work with a *virtual index* to the solutions that a query has. Each solution in the sequence of *m* solutions to a query receives an integer index in the range 1..*m*. This makes a query for the *i*th solution of a goal functional, and thus solves the impedance mismatch problem. So, to retrieve the first two solutions to an N-Queens solver, i.e., to do what corresponds to the command-line session,

```
?- queens(8, L).
L = [1, 5, 8, 6, 3, 7, 2, 4] ;
L = [1, 6, 8, 3, 7, 4, 2, 5] .
?-
```

we need to make *two* HTTP requests: `/queens?n=8&i=1` followed by `/queens?n=8&i=2`. Implementing this API in standard Prolog is trivial if each HTTP request runs the query and returns the *i*th solution. In order to make this *efficient*, we must make sure that the system remembers the state. We achieve this by using a Prolog thread (see Section 7.6) and message queues to communicate between the HTTP server threads and the *solver* threads. We use a high-level abstraction for creating efficient a-tuple-at-a-time web APIs to Prolog and programs written in Prolog:

thread_call(*+ID, :Goal, +I, +Bindings, -Result, +Options***)**
Computes the *I*th solution to the (possibly) non-deterministic *Goal* in a thread uniquely identified as *ID*. Succeeds exactly once and binds *Result* to a list of *Name=Value* pairs that provides information about the result, including the bindings specified in *Bindings* if *Goal* succeeded.

The idea is that when a call to **thread_call/6** exits, the thread referenced by *ID* may still be available and be ready to backtrack and compute more solutions for indices *greater* than *I*. For indices *smaller* than *I*, **thread_call/6** restarts the enumeration of solutions. Below is an example session. `new= @false` indicates that the second call reuses a state.

```
?- thread_call(t1, queens(8, L), 1, ['L'=L], R, []).
R = [success= @true, error= @false, errormessage= @null,
     bindings=['L'=[1, 5|...]], more= @true, new= @true, time=0.0].

?- thread_call(t1, queens(8, L), 2, ['L'=L], R, []).
R = [success= @true, error= @false, errormessage= @null,
     bindings=['L'=[1, 6|...]], more= @true, new= @false, time=0.01].
```

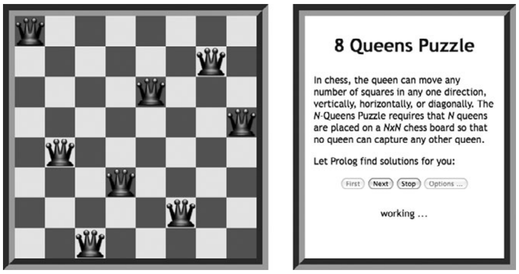
Building an N-Queens web application server is just a matter of setting up an HTTP server and declaring a handler which calls **thread_call/6** and outputs the result as JSON. The code for the handler is shown below:

```
:- http_handler(root(queens), queens, []).

queens(Request) :-
    http_parameters(Request, [n(N,[integer]), i(I,[integer])]),
    http_session_id(ThreadID),
    thread_call(ThreadID, queens(N, L), I, [queens=L], Result, []),
    term_to_json(Result, JsonTerm),
    reply_json(JsonTerm).
```

Table 1. *Average daily traffic from* `www.swi-prolog.org`*, September 2009*

Machine	2× AMD Opteron 2356 (quad core), 32 Gb memory
Hits per day	22,117
Visits per day	2,759
Traffic per day	3,376 Mb
CPU per day	1,632 seconds
Memory usage	25 Mb



8 Queens Puzzle

In chess, the queen can move any number of squares in any one direction, vertically, horizontally, or diagonally. The *N*-Queens Puzzle requires that *N* queens are placed on a *NxN* chess board so that no queen can capture any other queen.

Let Prolog find solutions for you:

working ...

Fig. 6. The N-Queens demo.

The module `term_to_json` provides the means to convert any Prolog term into a JSON structure. Above, it is used for converting the result from a call to **thread_call/6** into a JSON term which is then written to output using **reply_json/1** from library `http/http_json`. For example, the HTTP request `/queens?n=8&i=4` results in the response depicted below:

```
{ "success":true, "more":true, "error":false, "errormessage":null,
  "bindings": {"queens": [1,7,5,8,2,4,6,3]}, "new":false, "time":0.01
}
```

On the client side, nothing is new or in any way peculiar to the use of Prolog. Thanks to the pure JSON interface provided, the client side is a just an ordinary AJAX application that can be written by any programmer familiar with the languages and techniques involved. For our demo, we chose to work with YUI, but we could just as well have used any of the many alternatives available. The GUI to our N-Queens demo is shown in Figure 6.

6.2 The SWI-Prolog website

Since February 2009, SWI-Prolog's website is implemented using the SWI-Prolog HTTP server library. The basis is formed by PlDoc (Wielemaker and Anjewierden 2007), the SWI-Prolog literate programming system that provides a web-interface for the documentation of loaded code and the system manuals. SWI-Prolog hosting itself has two advantages: (1) it provides a realistic environment for testing the HTTP libraries and (2) PlDoc provides a uniform interface for all documentation and automatically creates links from documents in Wiki format to the documentation. Table 1 gives some statistics on the traffic that is handled by the site.

Provided services The server provides several different services: dynamically generated pages (serving from PlDoc and Wiki pages), small static pages (css, JavaScript, images, etc.), large static pages (downloads of binaries, sources, and PDF documentation), and CGI for supporting gitweb, the web frontend of the GIT source-code management system.

Stability and scalability The server uses the core parts of the SWI-Prolog (web) infrastructure: the multi-threaded HTTP libraries and XML parsing. Running this server 24 × 7 on the web has revealed four critical bugs (three of which were related to concurrency) and several memory leaks. Currently, the server runs without problems.

7 Language properties and extensions

SWI-Prolog is first of all a system for prototyping medium-scale (50–100 K-lines) applications, where Prolog is used as glue to unite external resources such as graphical libraries and (RDF) datastores. This idea has shaped the implementation.

7.1 The compiler and program loading

The compiler is distantly based on the ZIP abstract machine (Bowen et al. 1983; Neumerkel 1993). The SWI-Prolog VM is a structure copying machine that passes arguments through the environment and addresses arguments using an *argument pointer*. The argument pointer is also used to read and write arguments in lists and compound terms. The C-based emulator currently implements 145 instructions. The garbage collector is a mark-and-sweep collector that closely follows Appleby et al. (1988).

While extending the instruction-set, we ensured that both compilation and decompilation remain simple tasks. The compiler is written in C and used both for compiling source files and adding dynamic code using **assert/1**. As a result, static code and dynamic code have the same reflexive properties (e.g., **clause/2** can be used for both).

Our VM allocates all variables on the stack and passes all arguments over the stack. Where the WAM loses access to arguments and temporary variables (variables that only appear in two adjacent calls in the body), we allocate all these variables. An advantage of this is that the source-level tracer can display the value of all variables in a clause (see Figure 3). As a consequence, however, our environments are larger and without precautions, more data remain accessible through the environments. In Wielemaker and Neumerkel (2008), we show that the reachability problem can be solved at marginal costs by scanning the virtual machine code to determine which variables are initialized and still reachable.

The time needed for loading a large program (and for updating it after some of its source files are modified) must be short, lest the process of developing a prototype become too cumbersome. In particular, we are interested in two features: (1) it must be possible to compile files clause-by-clause, so there is no need for buffering, and (2) it must be possible to make code available *on-demand*, so there is no need to compile the whole program before execution can start. Starting with version 5.7, this is realized by the predicate *supervisor-code*. A predicate contains a linked list of clauses compiled to ZIP VM code and a supervisor. Execution of a predicate starts at the entry-point of the supervisor. Every predicate has the same initial supervisor, consisting of the single instruction `S_VIRGIN`. If this instruction is executed, it searches for the implementation of the predicate using this sequence:

(1) If already defined, we are done.
(2) If the predicate is in one of the *import* modules (see Section 7.2.2), import it.
(3) If the predicate is in the auto-load (lazy load) index, compile the source.
(4) If the predicate is still undefined, replace the supervisor with `S_UNDEF`, an instruction that handles calls to undefined predicates depending on the `unknown` flag.

After establishing the clauses for the predicate, `S_VIRGIN` examines the clause-heads by decompilation and generates the clause-indexing code.

Note that the supervisor only deals with lazy loading, lazy generation of indices, calling non-Prolog (i.e., foreign) predicates, and clause-*selection*. In particular, it carries no information that is not available in the predicate attributes and clause-list. This approach allows for *hotspot* compilation and other optimizations without complicating the reflexive features of SWI-Prolog (da Silva and Costa 2007).

7.2 The module system

Compatibility requirements during early development (see Section 1) have caused SWI-Prolog to adopt the Quintus Prolog predicate-based module system. However, we have made several modifications to this model to make it more suitable for rapid development and lazy loading of code. Using modules in Prolog has great practical value for two reasons: modules help avoid name-conflicts, especially for local *helper* predicates, and they define the public interface of a file. This section describes and motivates the modifications we made to the original Quintus Prolog model.

7.2.1 Meta-predicate handling

Meta-predicates are predicates that refer to other predicates. For example, **findall/3** takes a *goal* as argument. With a module system, there can be multiple predicates with the same name and arity, and a predicate must refer to the correct one: the one that appears in the same lexical context. For example, given the following program, **findall/3** (which is defined in the `system` module) must call **child/2** in the module `family`.

```
:- module(family, [ child/2, children/2 ]).

child(bob, jane).
child(bob, peter).

children(Parent, Children) :- findall(Child, child(Parent,Child), Children).
```

Predicate-based module systems solve this problem by declaring **findall/3** as a meta-predicate: `:- meta_predicate findall(?,0,-)`. An argument that has passed module-sensitive information (e.g., a goal) is specified by ':' or by an integer. An integer specifies that the argument is a goal and this goal will be called with *N* additional arguments (e.g., **maplist**(*2,?,?*)). This declaration is processed when the compiler compiles *a call to* a meta-predicate and causes the compiler to embed the argument in a term ⟨*module*⟩:⟨*plain*⟩ (i.e., to *qualify* the argument; `family:child(Parent,Child)` in the example above).

This approach is used by many Prolog systems, but it comes with two drawbacks: (1) the compiler must have access to information about whether any given predicate is a meta-predicate, and (2) modifying the meta-predicate declarations requires all code that *calls* this predicate to be recompiled. The first requirement either puts ordering constraints on the location of meta-predicate declarations or requires multi-pass compilation. With lazy loading, the index of predicates that can be loaded must be known at compile-time and must include meta-argument information. The second requirement complicates resynchronizing the Prolog database if one or more source files have changed (see the discussion of **make/0** in Section 3.2).

SWI-Prolog supports the **meta_predicate/1** directive without changing the compilation of code *calling* a meta-predicate. This is achieved by adding a *context module* to each environment. If an environment is created, the context is copied from the parent. Next, the virtual machine resolves the predicate (possibly through lazy loading). If the predicate is not a meta-predicate, the context is set to the module in which the predicate is defined. Next, the virtual machine starts executing the supervisor (see Section 7.1) of the predicate. The supervisor of a meta-predicate qualifies all meta-arguments and then sets the context to the module of the predicate.

This implementation satisfies our requirement (of being able to autoload meta-predicates and update meta-declarations dynamically) at the cost of some space and runtime overhead in managing the context module in each environment.

7.2.2 Import modules

The notion of import modules generalizes the distinction between built-in and user-defined predicates found in other Prolog systems. If a predicate is not locally defined, the system first tries to import it silently from the modules' import modules. In the normal setup, each user module imports from the `user` module, which, in turn, imports from `the system`. The `system` module contains the built-in predicates. The underlying machinery allows for multiple import modules per module and an arbitrary acyclic module-dependency graph. This mechanism is used to create unit-tests as isolated modules importing from the module-to-be-tested in PlUnit (see Section 8.2).

Especially for rapid development, programmers may choose to import utilities that are used at many places in an application into the `user` module. This makes these utilities available from the top level for debugging and avoids the need to import them in every application module. Note that, while a definition that is visible in the `user` module *can* be used in a module without explicit import, it is still *allowed* to import explicitly.

Import modules allow for different reuse schemes. SWI-Prolog supports Prolog de-facto standard import of the predicates **use_module/[1,2]**. It can support modularity similar to C by loading all modules into module `user` and omitting explicit import relations between the modules. Finally, it supports modules that inherit from their context (as used in PlUnit). Future versions are likely to split the `system` module into multiple modules to accommodate subsystems such as `iso`.

7.2.3 Operator handling

Most Prolog systems, even those that provide modules, use globally scoped operators. However, integration of large programs that feature programmer-defined operators is likely to fail due to operator conflicts. In the most common case, this results in syntax errors. In other cases, it results in different interpretation of terms that cause different behavior of the program. Such cases are hard to diagnose.

Therefore, we decided to make operators local to the module in which it is declared. The system searches for operators in the same order as it searches for predicates (see Section 7.2.2). This scheme allows for defining globally used operators in the `user` module. Support for global operators is needed for compatibility. Operators can be exported and imported. Here is an example from the library `record.pl`, which provides named access to fields in structures.

```
:- module((record), [ (record)/1, op(1150, fx, record) ]).
```

7.3 Avoiding limitations

We have attempted to avoid hard limits that could complicate the task of writing an application. In particular:

Atoms There is no limit on the length of atom names, which can be written in Unicode and include null characters. If atoms are used for processing input (text), this is needed to avoid representation errors, either on the length or the represented characters. Allowing for null characters allows representing arbitrary data (e.g., *image data*) as atoms. Processing input using atoms calls for atom garbage collection.

Integers Having no bounds to integers is not only meaningful to mathematicians, but it also allows representing integers in, e.g., XML documents as Prolog integers, without worrying about overflows. When building interfaces to foreign resources, it covers all limited integer types in a clean and uniform way.

Terms Compound terms have unbounded arity, which makes them particularly suitable for implementing arrays. SWI-Prolog supports rational trees (also called cyclic terms). Although we are not convinced that rational trees have much practical value in Prolog, crashing or looping on them is not acceptable. This is particularly the case for servers (DoS attacks) and education because students frequently create rational trees unintentionally (e.g., `List = [Head|List]`). SWI-Prolog can run in three modes, causing the unification above to succeed with a rational tree (default), to fail or to throw an error, depending on the Prolog flag `occurs_check`.

Stacks Unfortunately, SWI-Prolog's stacks are limited to 128 Mb per stack on 32-bit hardware. Given that 64-bit systems are now widely available, we do not plan to raise the limits on 32-bit hardware.

7.4 Global variables and destructive assignment

A global variable associates an identifier (we only allow atoms) with a term on the heap. We provide two types of assignment to global variables: backtrackable (**setval/2**) and non-backtrackable (**nb_setval/2**). The naming and implementation is based on hProlog (Demoen 2002).

Assigning a value to a global variable is a destructive operation. The same implementation can be used to facilitate destructive assignment of arguments of compound terms. Global variables and destructive modification of compound terms are useful in combination, for example, to implement a global array, as shown below:

```
new_global_array(Name, Size) :-
        functor(Array, array, Size),
        setval(Name, Array).

global_array_set_element(Index, Name, Value) :-
        getval(Name, Array),
        setarg(Index, Array, Value).
```

The backtrackable **setarg/3** is supported by many Prolog implementations. Non-backtrackable assignment as implemented in **nb_setarg/3** is less widely supported. GNU-Prolog supports it using **setarg/4**, but the argument value must be atomic. Backtrackable assignment is based on two-cell entries in the trail that maintains the old value.

Non-backtrackable assignment of a value that lives on the heap is more complicated. It is achieved by maintaining a global pointer (called *frozen_bar*) to the top of the heap at the moment of assignment. Backtracking never resets the top-of-heap below this mark. This implies that data that is older than the global term must be discarded by the garbage collector instead of by resetting the top-of-heap in backtracking. Backtracking may reset trailed bindings inside the value if the value is compound. This is indeed the case in hProlog. SWI-Prolog avoids this by making a copy of the value if the value is compound.

Sometimes, it is necessary to preserve state over backtracking. A clean solution to that are the all-solution predicates (e.g., **findall/3**). In standard Prolog, one can only use dynamic predicates if the all-solution predicates are not appropriate. The global nature of dynamic predicates makes it hard to implement re-entrance, thread-safety, and clean up in the event of an exception. One solution to this problem is given by Tarau (2008), by introducing explicit interaction with Prolog engines. We support this style of programming by using threads. Non-backtrackable assignment in compound terms provides another solution. The example below counts proofs for a goal. It is fast, safe, and runs in constant space.

```
proof_count(Goal, Count) :-
        State = count(0),
        (   Goal,
            arg(1, State, C0), C1 is C0 + 1, nb_setarg(1, State, C1),
            fail
        ;   arg(1, State, Count)
        ).
```

7.5 Attributed variables and coroutining

Attributed variables (Holzbaur 1992) were added in early 2004 to allow for constraints and coroutining. For SWI-Prolog, we chose to use the dynamic interface for attributed variables that was developed by Bart Demoen (Demoen 2002). This interface does not require attributes to be declared, and represents them with a linked list associated with the variable. This interface (which is currently available in hProlog, SWI-Prolog, XSB, Ciao, YAP, and SICStus) is the basis of the portable constraint libraries discussed in Section 4. Attributed variables are also used to implement the common coroutining predicates: **freeze/2**, **when/2**, and **dif/2**.

7.6 Multi-threading

Multi-threading was initially introduced to support scalable web servers in Prolog. The design and implementation is described in Wielemaker (2003) and is the basis for the ISO WG17 work on threading in Prolog as well as multi-threading support in YAP and XSB. In our design, each thread comes with an independent set of stacks. This implies that threads cannot share terms and therefore unification, backtracking, and garbage collection in each thread can be done independently from other threads. The key features are:

- Static predicates are fully shared. SWI-Prolog provides both shared and non-shared dynamic predicates.[3]
- Communication is primarily achieved by using message queues (*ports*) holding Prolog terms.
- There is a signaling interface that allows a thread to interrupt the execution of another thread. This interface was initially intended to support the debugger, but is also used to abort threads by injecting an exception into their control flow.
- Threads are layered on top of the POSIX thread primitives, providing smooth integration with thread-safe foreign code and taking full advantage of multi-core hardware.

We see two major application areas for concurrency in Prolog: (1) solving a large problem, and (2) solving many, mostly independent, tasks. Supporting problems of the first class is problematic because such problems often require intensive

[3] In SWI-Prolog, the default is to provide shared dynamic predicates. In XSB, dynamic predicates are by default non-shared. This is a safer choice because sharing dynamic data almost always calls for additional synchronization.

communication for which copying terms is too expensive or for which the copying semantics is inappropriate. Another problem is that when a thread encounters a serious error (e.g., a resource error), this may affect the entire computation (which may have to take appropriate action).

The design is successfully applied for the second class of problems, dealing with many, mostly independent, tasks. The SWI-Prolog HTTP libraries (Section 6) have grown into a mature multi-threaded web server.

7.7 The C-interface

As we have seen in Section 1, the C-interface was one of the success factors of SWI-Prolog in its early days. Mutually recursive calling between C and Prolog is now commonly supported. Below, we describe the more distinguishing features of the current C-API.

Non-determinism C-API supports non-deterministic foreign predicates by adding a context argument that provides the type of call. We distinguish three reasons to call the C function:

`PL_FIRST_CALL` The first call is the same as for deterministic foreign code. The function can return with one of three values: `FALSE` to indicate failure; `TRUE` to indicate deterministic success; or `RETRY` with an integer or pointer context. The context must carry enough information to compute the next solution. For example, when enumerating values from an array, this could be the array index. Often, the foreign predicate allocates a context structure and returns a pointer to this structure.

`PL_REDO` This call is issued by the kernel upon backtracking to an invocation of a foreign predicate. All Prolog arguments are guaranteed to be the same as for the initial goal. The implementation uses the previously returned context to compute the next result and returns with one of the three values, just as with the `PL_FIRST_CALL` call.

`PL_PRUNED` Called if the choice-point is pruned as a result of executing a cut or handling an exception. Only the context value is valid. The foreign implementation must clean up side-effects (e.g., free memory allocated for preserving the context).

Foreign context frames The C-API refers to Prolog terms through explicitly allocated *term-handles*. Because Prolog keeps track of the allocated handles, it knows which terms are references from C-code and can perform heap and atom garbage collection transparently. This mechanism was first introduced by Quintus Prolog and is now used in many modern Prolog implementations. Term-handles that are used in the implementation of predicates in C are discarded when the C-implementation returns to Prolog. Quintus does not provide an API to deallocate term-handles.

Setting up a call from C to Prolog involves allocating term-handles for constructing the arguments and processing the resulting bindings. If the overall application control is in C and the C-code makes multiple calls to Prolog, we need some way to discard term-handles. SWI-Prolog implements this by means of **PL_open_foreign_frame()** ... **PL_close_foreign_frame()**. All term-handles created between these two matching

```
#include <SWI-Prolog.h>
#define MKFUNCTOR(name, arity) PL_new_functor(PL_new_atom(name), arity)
static functor_t FUNCTOR_error2, FUNCTOR_existence_error2;

static int existence_error(term_t missing, const char *what)
{ term_t ex;
  if ( (ex = PL_new_term_ref()) &&
       PL_unify_term(ex, PL_FUNCTOR, FUNCTOR_error2,
                           PL_FUNCTOR, FUNCTOR_existence_error2,
                             PL_CHARS, what, PL_TERM, missing,
                           PL_VARIABLE) )
    return PL_raise_exception(ex);
  return FALSE;
}

static foreign_t pl_getenv(term_t name, term_t value)
{ char *ns, *vs;
  if ( !PL_get_chars(name, &ns, CVT_ATOM|CVT_EXCEPTION|REP_MB) )
    return FALSE;
  if ( !(vs=getenv(ns)) )
    return existence_error(name, "environment_variable");
  return PL_unify_chars(value, PL_ATOM|REP_MB, -1, vs);
}

install_t install_getenv()
{ FUNCTOR_error2           = MKFUNCTOR("error", 2);
  FUNCTOR_existence_error2 = MKFUNCTOR("existence_error", 2);
  PL_register_foreign("my_getenv", 2, pl_getenv, 0);
}
```

Fig. 7. Foreign wrapper for **getenv()**.

calls are invalidated.[4] In addition, **PL_rewind_foreign_frame()** rewinds (i.e., backtracks) the heap to the state at **PL_open_foreign_frame()**. Rewinding can be used to try an alternative if a sequence of **PL_unify_*()** calls fails (see next bullet).

Hand-crafted wrappers The Quintus C-API is designed to describe the C-code from Prolog and automatically generate a wrapper for it. The generated API is typically not Prolog-friendly. Values returned by C functions must be carefully mapped to Prolog success/failure or an exception. Using the Quintus approach, the final mapping to a natural Prolog API must be done in Prolog. This is particularly cumbersome when dealing with enum types or `#define` constants.

The SWI-Prolog C-API concentrates on passing Prolog terms and supporting Prolog success, failure, and errors. This implies that the *wrapper* is hand-crafted[5] and therefore we provided a third family of functions: PL_unify_⟨*type*⟩(Term, C-Value), which unifies a Prolog argument with a converted C-value and returns `TRUE` or `FALSE`.

We illustrate our approach in Figure 7. The actual wrapper is the function **pl_getenv()**. The first call extracts the name of the requested variable if it is an atom.

[4] SWI-Prolog also provides **PL_reset_term_refs()** to discard the argument-handle and all term-handles created afterwards. This function was copied by SICStus as **SP_reset_term_refs()** when porting XPCE to SICStus.

[5] The library qpforeign.pl provides a Quintus-compatible wrapper generator

The text is extracted as a C-string in the native locale.[6] If the extraction fails, it leaves an appropriate exception in the environment. If the **getenv()** API fails, we raise an exception.[7] Finally, if all goes well, we unify the extracted string with the second argument and return the success of this unification. The module can be compiled, loaded, and used as shown in the example below. The swipl-ld utility is a wrapper around the C compiler that hides platform-specific details.

```
% swipl-ld -shared -o getenv getenv.c
% swipl
1 ?- load_foreign_library(getenv).
true.
2 ?- my_getenv('HOME', X).
X = '/home/janw'.
3 ?- my_getenv(notavar, X).
ERROR: environment_variable ‘notavar’ does not exist
```

8 The development model for SWI-Prolog

A language like SWI-Prolog is only taken seriously by its users if its implementation is stable and dependable. This desire for stability is at odds with the academic desire for exploration and development of new language features. In order to reconcile these two desires, the development of SWI-Prolog proceeds in branches according to well-known Open Source practices described in Raymond (1999). In particular:

- Even-numbered branches are stable and odd-numbered branches are for development.
- Create a new branch when (1) the development branch appears to be stable (i.e., there are few reported stability issues), and (2) new developments are about to aversely impact stability. The stable branch is supposed to provide a stable core, but newer libraries may be unstable.
- Release often. The typical release cycle on the active development branch is 2 weeks, but can be shorter if the users require particular fixes or functionality. It can be longer if the current state is considered too unstable or there are no changes of interest.
- Respond quickly. When possible, provide a fix or work-around for problems communicated on the mailing list or posted on the bug-tracking system. If the problem is expected to affect many users, make a release. Otherwise, the patch is only available through the GIT[8] repository. Professional users are expected to be able to build the system from the GIT distribution.
- Maintain a regression test suite and run it frequently. We do not have the resources to create a comprehensive test suite. As an in-between solution, we typically create test-cases from bug-reports. This scheme avoids bugs from reappearing.

[6] Internally all text is stored as Unicode.
[7] We are planning to provide the functionality of **existence_error()** in the Prolog C-API.
[8] `http://git-scm.com/`

8.1 The regression test suite

The regression test suite is activated through the GNU Makefile standard target `check`. For historical reasons, the tests are built according to two different test paradigms:

- Single-clause tests that are expected to succeed. The predicate name indicates the set of tests and the first argument is a ground term that identifies the test. The test driver enumerates the tests using **clause/2**.
- Scripts. A script is a Prolog file, typically named test_⟨*topic*⟩.pl. It defines a module test_⟨*topic*⟩ that exports test_⟨*topic*⟩/0. The test driver enumerates all files in a directory, loads them, and runs the exported goals. Scripts are used to test larger programs (e.g., solve a constraint problem). More recently, they also contain unit-test suites as described in Section 8.2.

Currently, the test suite for the core system contains 411 tests of the first form and 58 test scripts. Thirty-five of the 58 test scripts are *PlUnit* suites, providing another 353 individual tests.

8.2 PlUnit: the test driver framework

PlUnit[9] is the test driver framework of SWI-Prolog. Unlike the driver framework outlined above, *PlUnit* is targeted at users of SWI-Prolog. It is based on the idea of single-clause tests, but uses a slightly different way to identify test-clauses and adds a second argument that specifies properties of the body, such as expected bindings, non-determinism, failure, or raised exception. The second argument can also specify a condition to run the test, and a goal to setup and clean up the environment to run the test. Below is a simple example:

```
:- begin_tests(aggregate).
test(aggregate_count, Count == 2) :-
        aggregate(count, X^between(1,2,X), Count).
:- end_tests(aggregate).
```

A description of the expected behavior of the body allows the test driver to give a more descriptive report if a test fails (contrasting the expected and actual behavior).

A test-block (`begin_tests`..`end_tests`) is compiled to a module that inherits from its lexical context (see Section 7.2.2). This allows test units to be embedded in the actual source code: the tests have access to the internals of the module to be tested but do not pollute the namespace of this module. The *PlUnit* driver can be asked whether or not embedded tests should be compiled and whether or not they should be run automatically by **make/0** (see Section 3.2) after the module has been modified.

9 Discussion and future work

SWI-Prolog has become a comprehensive and mature implementation of the Prolog language. Its focus is on integrating technology from the logic programming

[9] `http://www.swi-prolog.org/pldoc/package/plunit.html`

community and interfacing to external resources to provide a platform for prototyping and development of fairly large applications. The system is widely used in educational, research, and commercial environments.

Prolog still has a difficult marketing position. It is generally perceived as hard-to-learn, lacking ready-to-use resources and a good Integrated Development Environment (IDE). Nevertheless, the Prolog language is being used in new projects. In such projects, we typically find a mixture of four components: (1) application-specific high-level languages, and (2) rule-based reasoning, and (3) constraint handling, and (4) Semantic web (RDF) data.

Prolog is well equipped to compile high-level application-specific descriptions into programs that combine the other three components. Using application-specific descriptions is particularly suitable for domains that face frequent changes in the rules and procedures. For example, SecuritEase[10] uses CHR for transforming their Constraint Query Language into SQL. Although we lack hard evidence, we think that many commercial users deploy SWI-Prolog as a multi-threaded server component.

In many universities, Prolog is now taught as part of a course on programming paradigms. After a few weeks, students often come to the conclusion that logic programming is a neat idea, but it is not useful for anything practical. Possibly, Prolog should be taught in the context of, e.g., the Semantic Web, where students can create applications using real data that are readily available on the web, and can compare writing Prolog rules over this data with querying this data in an imperative language through a SPARQL interface.

For a long time, the Prolog community was divided into many isolated islands. Adoption of the ISO-standard, although not perfect (Szabó and Szeredi 2006), and the development of larger portable resources such as Logtalk, Leuven CHR, CLP(FD), and CLP(Q,R) have built bridges between these islands. Developers of portable resources persuade Prolog system developers to resolve incompatibilities. At the same time, the existence of portable resources makes the logic programming community more credible.

Future SWI-Prolog development will concentrate on the following aspects:

- Improving compatibility, notably with systems with a similar module system.
- Improving stability, scalability, and performance.
- Improving support for rule-based programming by providing tabling.
- Providing more libraries, notable for RDF and web programming.
- Improving the development environment, notably by adding a type-checker (Schrijvers et al. 2008), adding (style) warnings, and adding tools that support refactoring of programs.

Acknowledgements

Continuous development of SWI-Prolog has been made possible by the SWI department of the University of Amsterdam. Currently, the development is hosted at the web and media group of the VU University Amsterdam. An important

[10] `http://www.securitease.co.nz/`

factor in the success of SWI-Prolog is the community that provides the motivation, challenges, and bug reports, as well as code, patches, and descriptions of how to fix errors in the implementation. In particular, we would like to acknowledge Paulo Moura for Logtalk, his effort in pointing out incompatibilities between Prolog implementations and his work on the MacOS port; Richard O'Keefe and Bart Demoen for background knowledge of libraries, standards, and implementation details, and Ulrich Neumerkel for testing and code.

References

Ambler, S. W. 2002. Mapping objects to relational databases: What you need to know and why IBM developerWorks. Accessed 16 August 2011. URL: `http://www.ibm.com/developerworks/library/ws-mapping-to-rdb/`.

Anjewierden, A., Wielemaker, J. and Toussaint, C. 1990. Shelley – computer aided knowledge engineering. In *Current Trends in Knowledge Acquisition*, B. Wielinga, J. Boose, B. Gaines, G. Schreiber and M. van Someren, Eds. IOS Press, Amsterdam, 41–59.

Appleby, K., Carlsson, M., Haridi, S. and Sahlin, D. 1988. Garbage collection for Prolog based on WAM. *Communications of the ACM 31* (6), 719–741.

Bellifemine, F., Poggi, A. and Rimassa, G. 2001. Developing multi-agent systems with a FIPA-compliant agent framework. *Softw: Practice and Experience 31* (2), 103–128.

Bendisposto, J., Endrijautzki, I., Leuschel, M. and Schneider, D. 2009. A semantics-aware editing environment for Prolog in Eclipse. *CoRR*. Accessed 17 August 2011. URL: `http://arxiv.org/abs/0903.2252`.

Bowen, D. L., Byrd, L. M. and Clocksin, W. 1983. A portable Prolog compiler. In *Proc. Logic Programming Workshop '83*, L. M. Pereira, Ed. Universidade Nova de Lisboa, Lisbon, Portugal.

Calejo, M. 2004. Interprolog: Towards a declarative embedding of logic programming in Java. In *Proc. JELIA*, J. J. Alferes and J. A. Leite, Eds. Lecture Notes in Computer Science, vol. 3229. Springer-Verlag, Berlin, Germnay, 714–717.

da Silva, A. F. and Costa, V. S. 2007. Design, implementation, and evaluation of a dynamic compilation framework for the YAP system. In *Proc. ICLP*, V. Dahl and I. Niemelä, Eds. Lecture Notes in Computer Science, vol. 4670. Springer-Verlag, Berlin, Germany, 410–424.

De Koninck, L., Schrijvers, T. and Demoen, B. 2006. INCLP($\mathbb{R}$) – Interval-based nonlinear constraint logic programming over the reals. In *WLP '06: Proc. 20th Workshop on Logic Programming*, M. Fink, H. Tompits and S. Woltran, Eds. T.U.Wien, Austria, INFSYS Res. Rep. 1843-06-02. Vienna, Austria, 91–100.

Demoen, B. 2002. *Dynamic Attributes, Their hProlog Implementation, and a First Evaluation.* Rep. CW 350. Department of Computer Science, K.U. Leuven, Leuven, Belgium. October. Accessed 16 August 2011. URL: `http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html`.

Draxler, C. 1991. Accessing relational and NF^2 databases through database set predicates. In *ALPUK91: Proc. 3rd UK Annual Conference on Logic Programming,* Edinburgh, UK, G. A. Wiggins, C. Mellish, and T. Duncan, Eds. Workshops in Computing. Springer-Verlag, Berlin, Germany, 156–173.

Frühwirth, T. 1998. Theory and practice of constraint handling rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming 37* (1–3), 95–138.

Graham, S. L., Kessler, P. B. and McKusick, M. K. 1982. gprof: A call graph execution profiler. In *Proc. SIGPLAN Symposium on Compiler Construction.* ACM, New York, USA, 120–126. Accessed 17 August 2011. URL: `http://doi.acm.org/10.1145/800230.806987`.

Holzbaur, C. 1992. *Metastructures vs. Attributed Variables in the Context of Extensible Unification*. Tech. Rep. TR-92-23, Austrian Research Institute for Artificial Intelligence, Vienna, Austria.

Holzbaur, C. and Frühwirth, T. 2000. A Prolog constraint handling ules compiler and runtime system. *Journal of Applied Artificial Intelligence, 14*(4), , 369–388.

Lassila, O. and Swick, R. R. 1999. Resource description framework (RDF) model and specification. Recommendation, W3C Consortium. 22 February. URL: `http://www.w3.org/TR/1999/REC-rdf-syntax-19990222`.

Neumerkel, U. 1993. *The Binary WAM, a Simplified Prolog Engine*. Tech. Rep., Technische Universität Wien, Wien, Germany. URL: `http://www.complang.tuwien.ac.at/ulrich/papers/PDF/binwam-nov93.pdf`.

Oren, E., Delbru, R., Gerke, S., Haller, A. and Decker, S. 2007. Activerdf: Object-oriented semantic web programming. In *WWW '07: Proc. 16th International Conference on World Wide Web*. ACM, New York, NY, USA, 817–824.

Pereira, F. C. N. and Shieber, S. M. 1987. *Prolog and Natural-Language Analysis*. Number 10 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, CA. Chicago University Press, Chicago, IL.

Raymond, E. S. 1999. *Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media, Sebastopol, CA.

Schrijvers, T. 2005. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. Ph.D. thesis, K.U. Leuven, Leuven, Belgium.

Schrijvers, T., Costa, V. S., Wielemaker, J. and Demoen, B. 2008. Towards typed Prolog. In *Proc. ICLP*, M. G. de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer-Verlag, Berlin, Germany, 693–697.

Schrijvers, T. and Demoen, B. 2004. The K.U.Leuven CHR system: Implementation and application. In *Proc. CHR '04, Selected Contributions*, T. Frühwirth and M. Meister, Eds. Universität Ulm, Fakultät für Informatik, Ulm, Germany, 8–12.

Schrijvers, T. and Warren, D. S. 2004. Constraint handling rules and tabled execution. In *Proc. ICLP '04*, Saint-Malo, France, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer-Verlag, Berlin, 120–136.

Schrijvers, T., Wielemaker, J. and Demoen, B. 2005. Poster: Constraint handling rules for SWI-Prolog. In *W(C)LP '05: Proc. 19th Workshop on (Constraint) Logic Programming*, A. Wolf, T. Frühwirth, and M. Meister, Eds. Ulmer Informatik-Berichte, vol. 2005-01. Universität Ulm, Ulm, Germany.

Szabó, P. and Szeredi, P. 2006. Improving the ISO Prolog standard by analyzing compliance test results. In *Proc. ICLP*, S. Etalle and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 4079. Springer-Verlag, Berlin, Germany, 257–269.

Tarau, P. 2008. Logic engines as interactors. *CoRR*. Accessed 17 August 2011. URL: `http://arxiv.org/abs/0808.0556`.

Triska, M., Neumerkel, U. and Wielemaker, J. 2009. Better termination for Prolog with constraints. *CoRR*. WLPE 2009, Udine, Italy. Accessed 17 August 2011. URL: `http://arxiv.org/abs/0903.2168`.

Wallace, M. and Schimpf, J. 1999. Eclipse: Declarative specification and scaleable implementation. In *Proc. PADL*, G. Gupta, Ed. Lecture Notes in Computer Science, vol. 1551. Springer, Berlin, Germany, 365–366.

Wielemaker, J. 2003. Native preemptive threads in SWI-Prolog. In *Practical Aspects of Declarative Languages*, C. Palamidessi, Ed. Lecture Notes in Computer Science, vol. 1551. Springer-Verlag, Berlin, Germany, 331–345.

Wielemaker, J. and Anjewierden, A. 2002. An architecture for making object-oriented systems available from Prolog. In *Proc. WLPE*. 97–110. Accessed 17 August 2011. URL: `http://lanl.arxiv.org/abs/cs.SE/0207053`.

Wielemaker, J. and Anjewierden, A. 2007. PlDoc: Wiki style literate programming for Prolog. In *Proc. 17th Workshop on Logic-Based Methods in Programming Environments*, P. Hill and W. Vanhoof, Eds. 16–30. Accessed 17 August 2011. URL: `http://arxiv.org/abs/0712.3116v1`.

Wielemaker, J., Hildebrand, M., van Ossenbruggen, J. and Schreiber, G. 2008. Thesaurus-based search in large heterogeneous collections. In *Proc. International Semantic Web Conference*, A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, and K. Thirunarayan, Eds. Lecture Notes in Computer Science, vol. 5318. Springer-Verlag, Berlin, Germany, 695–708.

Wielemaker, J., Huang, Z. and van der Meij, L. 2008. SWI-Prolog and the web. *Theory and Practice of Logic Programming 8*(3), 363–392.

Wielemaker, J. and Neumerkel, U. 2008. Precise garbage collection in Prolog. In *Proc. CICLOPS '08*. Manuel Carro Linares and Bart Demoen, Eds. Udine, Italy. Accessed 12 and 13 December 2008. URL: `http://www.clip.dia.fi.upm.es/Conferences/CICLOPS-2008/`.

Wielemaker, J., Schreiber, G. and Wielinga, B. 2003. Prolog-based infrastructure for RDF: Performance and scalability. In *Semantic Web – Proc. ISWC '03*, Sanibel Island, FL, D. Fensel, K. Sycara, and J. Mylopoulos, Eds. Lecture Notes in Computer Science, vol. 2870. Springer-Verlag, Berlin, Germany, 644–658.