SWI-Prolog: history and focus for the future

Jan Wielemaker

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

Abstract. In this article, we provide a historical overview of the development of SWI-Prolog. We use this historical perspective to identify what we perceive as primary reasons for the popularity of the system. We use recent developments within and outside the Prolog community to comment on possible future directions for sharing more resources, such as applications and libraries, between Prolog developers. We argue that sharing and compatibility across engines go hand-in-hand and we conclude with our vision of immediate future steps towards these goals.

1 In the beginning

Considering the rather personal view of this section, it is written in a personal style and uses 'I'.

I came into contact with Prolog during an apprenticeship at the University of Edinburgh, visiting the mathematical reasoning group let by Alan Bundy. This is also where I met Richard O'Keefe, although he worked mostly at night and I used somewhat more traditional office hours. Coming from an engineering background Prolog came as a shock, but quickly I became hooked on the language. After my studies, in 1985, I became a researcher at the SWI department of the University of Amsterdam. SWI was led by Bob Wielinga and SWI had just acquired one of the early European (Esprit) projects, called KADS: Knowledge Acquisition and Documentation System. One of the tasks was the development of a 'workbench for knowledge engineering'. The official development platform was Quintus Prolog (still in release 1.x) and Anjo Anjewierden's PCE library for graphical user interfaces developed for C-Prolog. This was an inspiring place for learning about programming and software architecture.

The workbench had to become an interactive program, where knowledge engineers could analyse diverse textual data sources, including transcripts of interviews with domain experts and structure this information using the (graphical) modelling languages of the KADS methodology. Quite a challenge in the days where state-of-the-art was programming in C using the SunView graphical libraries.[1]

While happily programming on the project, I read "A portable Prolog compiler" [1] and decided to implement the described minimal runtime support system for Prolog,

[1] The LISP community already had InterLisp, but this was incompatible with the hardware and software platform of choice in the project.

just for the fun. The first version had its compiler written in C-Prolog, but the tiny system worked and performed similar to C-Prolog. I added a Prolog parser written in C and rewrote the simple compiler in C. With these two additions, the system no longer needed an external Prolog system to bootstrap itself and compile programs. In addition, because the compiler was small and fast and decompilation was nearly trivial, the same compiled format could be used for dynamic predicates. A debugger could easily be hooked into the virtual machine, leading to one representation for debug/nodebug code as well as static/dynamic code. These properties are still present in today's SWI-Prolog. Of course, it was still just a toy system. But, it performed like C-Prolog and compiled *much* faster then both C-Prolog and Quintus. It started to become fun.

In the meanwhile, the connection between PCE and Prolog became a serious bottleneck. The initial connection, using pipes, was replaced by an interface based on the foreign language (C) interface. This provided the required performance, but mutually recursive calls between C and Quintus Prolog were not supported in those days.[2] This complicated flow-control, especially for *modal* dialog objects. Also, lack of access to the toplevel forced running a loop for processing user events, loosing access to the interactive Prolog toplevel.

I started to see opportunities for my (still unnamed) toy. I interfaced it with PCE, showing the benefits of mutually recursive calls between C and Prolog and integration of the PCE event processing into the Prolog toplevel such that it was possible to interact with the GUI and interact with the Prolog text console at the same time. Next step, I added enough built-in predicates to run the workbench prototype. Further steps to convince the project included adding a module system that was closely compatible to Quintus and optimizing it so carefully on CHAT80 ([3]) that it outperformed Quintus on this program.

2 Yet another Prolog

Suddenly, we had "Yet Another Prolog", not to be confused with YAP. It was simple and small, and served our purposes. The KADS project partners had ensured they had access to it. But, what else to do with it? Commercial exploitation didn't seem attractive, as there were already enough commercial Prolog implementations and its value was not in its general quality, but (1) in some specific aspects (bi-directional C interface, a toplevel hook and some development goodies like **make/0**) that were easy enough for commercial competitors to add and (2) a simple system over which we had full control.

We started to get requests for the system from people who heart about it through the project partners. As a growing number of universities was connected to the Internet, we added a license modelled after what was common practice for academic software in the late eighties, providing free usage for research and education and uploaded the sources to the departmental FTP server. The PCE graphics system was perceived to have more potential and was sold for a fee for both academic and commercial use.

Because the system demanded very little resources, was easy to install, required no license forms to be exchanged and provided all functionality required by Prolog textbooks, it quickly became one of the popular Prolog systems in education. Other than

[2] This was added in Quintus Prolog version 3.

that, it was a simple minded system lacking fashionable features and involvement in research topics, such as garbage collection, concurrency, constraints, more advanced resolution techniques, etc. Its position as an educational system was strengthened further when it was ported to Microsoft Windows 95 because of the requirements of another European project.

3 Growing up

For a long time, the status of SWI-Prolog didn't change much. That is, the system got a lot more functionality, but the development was mostly done by the main author with small contributions, dealing mostly with portability aspects. Development was primarily driven by internal projects. Key to the growing popularity was the open source model with a short release cycle and short response times, both for resolving bugs and adding new functionality. The system was still slow and was definitely not leading innovation in the logic programming community.

With SWI-Prolog 4 and PCE 5, in the meanwhile renamed to XPCE after it was ported from SunView to X11 and Microsoft Windows, the license was simplified to LGPL (Lesser General Public License) and a modified version of the GPL for the Prolog code that removes the 'viral' aspect of the GPL.[3] A simplified and in the meanwhile widely accepted license, together with a modernised website hosted on a stable address (`http://www.swi-prolog.org`) were important steps for SWI-Prolog. This was boosted by the success of the GNU toolchain and Linux. The open source model, where software was not backed up by a commercial organisation, was acceptable for academia and research software, but also for commercial software by a growing number of IT professionals.

As a result, in addition to internally driven development, commercial users started to sponsor development and contribute components. Examples of sponsored development are the initial garbage collector, unbounded integer and rational number support, the SSL (Secure Socket Layer) interface, the unit test (PlUnit) and literate programming libraries (PlDoc). The main example of a package developed for commercial use and contributed is JPL, the Java interface written by Paul Singleton.

Education was (and still is) responsible for the vast majority of the installations. The educational needs were served by people writing beginners guides targeted on getting SWI-Prolog installed and using it to do the programming required in courses. Gerhard Röhner developed SWI-Prolog Editor[4], an easy to use frontend for Microsoft Windows.

At the ICLP 2003 (Mumbay), Bart Demoen and Tom Schrijvers proposed to port the Leuven constraint libraries to SWI-Prolog. With help from Bart and Tom we implemented attributed variables using the API of hProlog, as well as partial support for cyclic terms (rational trees). This suddenly opened SWI-Prolog for CLP (Constraint

[3] The LGPL is specifically targeted at dynamic linking, and cannot be applied easily to Prolog code. The license for the Prolog part was established in direct cooperation with Richard Stallman from the FSF, based on the license for the C code of the GCC runtime library.

[4] `http://lakk.bildung.hessen.de/netzwerk/faecher/informatik/swiprolog/indexe.html`

Logic Programming), a key technology in the LP community. Markus Triska used these new facilities to provide a portable pure Prolog implementation of CLP(fd).

In the meanwhile, internal project needs had added support for multi-threading, Unicode, HTTP services and support for HTML/SGML/XML and RDF documents. This powerful basis for deploying Prolog in fashionable client/server architectures did not pass unnoticed. It was first pickup by Solvo (Russia). Without the help of Sergey Tikhonov, multi-threading, and notably its implications on atom-garbage collection, would probably never have passed the prototype status. For SSS (New Zealand), Mike Elston, Keri Harris and Matt Lilley facilitated and helped maturing the Windows port, ODBC interface and much more.

4 Success factors

It is hard to measure success. The distribution model does not involve sales that can be counted. We count downloads, but many installations are from external redistributions, such as Linux distributions, media distributed with textbooks and PortableApps[5]. Even if we could count installations, it says little about usage. What comes for free is easily discarded, notably after the Prolog course has finished. Additionally, even if we could measure the number of used installations, it would still be hard to know the reason.[6]

All we have are some anecdotal evidence. First, we identify the reasons that are in our own subjective opinion are the main contributors to SWI-Prolog's success in education. None of these features are (any longer) unique to SWI-Prolog. SWI-Prolog merely happened to have them at the right time.

- No administrative work involved.
- Easy installation, providing a compatible environment on Unix, Windows and MacOS.
- Small in the days that resource usage mattered.
- Provides all basics required to run programs from textbooks. No need to load libraries to run the examples thanks to autoloading. It 'just works'.
- Comes with reasonable development tools, while there are also alternative independent tools that work well (SWI-Prolog Editor, PDT, GNU-Emacs).
- Mailing list where a good mix of novice and expert postings co-exist.

For research and commercial usage, the above is not enough. Here, typically robustness, performance, scalability, functionality, (backward) compatibility and (long time) support become important additional requirements. For most of these factors, hard evidence is hard to give, but we will give an educated guess. As with the key features that enabled widespread use in education, SWI-Prolog does not have unique features. Possibly, the key is to have enough features in combination with robustness and active response to bug reports and feature requests.

[5] `http://portableapps.com/`

[6] We ran a questionnaire years ago, but there was not enough feedback to draw any sensible conclusion.

- *Robustness* it good when compared to free systems and probably at least average when compared to commercial systems. Robustness issues in SWI-Prolog, are generally fixed quickly and due to the open source and quick release cycle, their impact on development and release plans for applications is usually small.
- The opinions on *Performance* vary. SWI-Prolog scores low on almost all (small) benchmarks, but the picture becomes less clear on large applications due to good *scalability* and good performance in aspects that do not appear much in benchmarks but apparently more so in applications, such as use of the dynamic database (assert/retract) and meta-calling. Also, many of the built-in predicates are coded in C and perform well. In [4], we claim performance equal to SICStus version 3 on Alpino, a parser for the Dutch language.
- *Scalability* is good, especially since adding JIT indexing on multiple arguments. Recent versions have much improved memory and thread management, accommodating thousands of threads.
- *Functionality* is a-typical, aiming at using Prolog as 'glue' in large applications rather than concentrating on the logical aspects of Prolog ('logic server' model). The a-typical functionality becomes clear if we consider the many interfaces, multi threading and unicode support in contrast to the relatively late addition of constraints and the still lacking support for tabling.
- *(Backward) compatibility* is fair, but not outstanding. Autoloading allows for moving built-ins to the library or renaming them while providing the old definition through the library `backcomp`. This causes such changes not to be noticed by the large majority of the users, but applications using e.g., explicit module qualification will break.
- There is no formal *support*. Formal support used to be perceived as a vital requirement for commercial use. Given the current role of open source software that is supported by a community, what matters is that the project is being actively developed for a long period and questions are being answered.
- *Familiarity*. This is particularly hard to judge, but the extensive use of SWI in education might well have been a contributing factor to each deployment in both research and business.
- *Forward looking*. The fact that during the past decades the development has capitalised on user and project requirements to modernise the system give confidence that new ideas, technologies and requirements are competently incorporated.

5 Portability, Standards and LP community

Portability and standards compliance across Prolog implementations is generally perceived as poor, and one of the main problems with the acceptance of Prolog language as a major general programming language. This is particularly true when compared to single-vendor languages like Python or Perl or strongly controlled languages like Java. When compared to C or C++ (still major languages, see e.g., `http://langpop.com`), these languages may be better standardised, but their libraries are often highly platform dependent. Here, we also see that major players like Microsoft do not implement the C99 standard or the POSIX thread standard in their current products.

In [4] we described the status of porting and writing portable Prolog code in more detail. In a separate recent effort, we established an IF/Prolog emulation layer that can deal with a large part of IF/Prolog 5, including its module interface that is similar to the ISO part II standard and the **block/3** construct from IF/Prolog 4 (which is available as an undocumented feature in IF/Prolog 5). Development of the emulation layer together with testing it on approximately 1,000,000 lines of Prolog code took 3 weeks with two persons. In the meanwhile, the code was migrated from Windows using an MFC based GUI to Linux using a web-based GUI. The SWI-Prolog/YAP compatibility layer described in [4] was at the heart of this success. Enhancement to term-expansion in dealing with modules and the **include/1** directive as well as two minor extensions to the kernel to support the **@/2** and **block/3** construct completed the work. The IF/Prolog emulation library is now part of the SWI-Prolog release.

Such enterprises become increasingly more ease to program thanks to convergence we have seen over the last years. This convergence was initiated by the ISO standard and Paulo Moura's work on getting Logtalk to run on all major and many minor Prolog implementation. During the past few years, vendors communicate directly using the prolog-standard mailing list. While not perfect, issues are resolved there, creating convergence rather than divergence.

Ideally, one would wish for a slightly more formal mix of users and developers that drive new ideas springing from individual engines into useful re-usable code that runs on a number of systems.

6 Towards sharable resources for Prolog

We believe that one of the major drawbacks of the Prolog community with respect to many other languages is the lack of ready-to-use libraries. The library included with a particular Prolog implementation often include parts that decent from a common ground such as the DEC-10 library, resources that are shared by multiple systems, such as CHR and CLP(FD) and a large diversity of proprietary code. The library is compiled by the vendor as a compromise between the number of users interested and required resources, both in terms of development, documentation, (in)coherence with the rest of the system size of the distribution. In other words it serves the majority, but it does not serve the 'long tail'.

Application programmers often need functionality that is in the 'long tail'. Most likely some user wrote a library that either suits your need or is a good starting point. Even if this code was made available, it is hard to find and porting it to the target Prolog environment may not be trivial. If the code has dependencies to other libraries it quickly becomes infeasible to reuse it.

One of the enabling factors for sharing code is a language with good portability and a system to specify dependencies. An example from the Java world is Maven[7]. Given the current state of portability within the Prolog world, a similar solution seems infeasible. A more realistic scenario might be the scenario that what is currently being used for open source applications, were applications such as Emacs or many of our Prolog systems are available for a wide range of operating systems. The challenge for making such

[7] `http://maven.apache.org`

systems portable is in dealing with differences in the C/C++ compilers, different runtime libraries, different GUI systems, different toolchains for building applications and finally different packaging and dependency tracking systems for all these ecosystems. Dealing with this diversity is in part resolved using tools (e.g., GNU autoconf, cmake), portable libraries (e.g., wxWindows, the Boost infrastructure for C++, Cygwin bringing POSIX to Windows). Remaining issues are typically resolved using project-specific abstraction layers and/or conditional statements. Portability is typically established in an interaction between developers, maintainers and users.

The natural next step to a packager with capabilities of downloading, installing and resolving dependencies, is a public repository of open source contributed code. Other communities have thrived when they created such repositories and have demonstrated that publicly available code enhances acceptability and publishing chances for research (for instance, the R language [2]).

Prolog has a lot to offer as a general programming language and as a vehicle for prototyping in research. We argue that further penetration to research and general use will depend on expanding the non-core parts of the language and the existing systems and a focus on re-usable code.

7 Conclusions

We have described the history of the development of SWI-Prolog. In our perception, the major factors in its success are partly technical (especially stability, portability and a wide coverage of features), partly organisational (early adoption of open source model with early and frequent releases combined with short response times on bug reports and feature requests, and a lively mailing list) and partly timing (open source release and port to Microsoft Windows).

Building on the small but sound foundation of the ISO standard and community initiated de-facto standardisation, resulting from portable applications (in particular Logtalk), YAP and SWI established a portability framework. This framework has been used both to make large applications portable (i.e., run unmodified on multiple Prolog systems) as well as being the basis for porting such applications.

We believe that the next challenge, in addition to extending and improving our Prolog implementations and building applications that convince people of the capabilities of the Prolog language, is to realise a framework for sharing code in the Prolog community. The open source community provides applications that are portable over different operating systems based on portability tools and conditional compilation. We have an initial compatibility framework that is proven on some large real-world applications. A growing number of Prolog implementations provide conditional compilation. If Prolog systems can complete this infrastructure with a package installation system that can deal with downloading and installation of dependencies, we have usable infrastructure for sharing Prolog resources.

Acknowledgements

SWI-Prolog is a community. Continuous development has been facilitated by the University of Amsterdam for a long time and is now continued at the VU University of Amsterdam. Many people and projects have contributed to the system.[8]

Core to the portability and cooperation work described here are Vitor Santos Costa (YAP/SWI integration layer), Paulo Moura for pointing implementors of Prolog systems at incompatibilities and propose solutions, Richard O'Keefe for "An Elementary Prolog Library" [9], Ulrich Neumerkel for his work on extending the ISO standard and Nicos Angelopoulos for commenting on this article and initiating new cooperation work. Also the one-day 'Prolog Commons' project hosted by Tom Schrijvers and Bart Demoen in Leuven has enabled some more integration.

References

1. D. L. Bowen, L. M. Byrd, and WF. Clocksin. A portable Prolog compiler. In L. M. Pereira, editor, *Proceedings of the Logic Programming Workshop 1983*, Lisabon, Portugal, 1983. Universidade nova de Lisboa.
2. R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Found. for Stat. Comp., Vienna, Austria, 2012.
3. David H. D. Warren. Efficient processing of interactive relational data base queries expressed in logic. In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 272–281. VLDB Endowment, 1981.
4. Jan Wielemaker and Vítor Santos Costa. On the portability of prolog applications. In Ricardo Rocha and John Launchbury, editors, *PADL*, volume 6539 of *Lecture Notes in Computer Science*, pages 69–83. Springer, 2011.

[8] `http://www.swi-prolog.org/Contributors.html`
[9] `http://www.cs.otago.ac.nz/staffpriv/ok/pllib.htm`