Foreign Language Interfaces for Prolog: A Terse Survey

Roberto Bagnara
Department of Mathematics
University of Parma
Italy
bagnara@cs.unipr.it

Manuel Carro
School of Computer Science
Technical University of Madrid (UPM)
Spain
mcarro@fi.upm.es

Version 1

Abstract

The availability of good foreign language interfaces is fundamental for the interoperability of different programming languages. While this observation is true for any language, it is especially important for non-mainstream languages such as Prolog, since they are likely to be discarded when shortcomings of the interfaces suggest the adoption of just one implementation language. In this paper we review some existing Prolog foreign language interfaces, trying to highlight both the important characteristics they share and the features that distinguish them from one another. We argue that even a basic, but *standard* Prolog foreign language interface could significantly contribute to increasing the adoption of Prolog for those subsystems where it is "the right language". Finally we suggest which steps could be taken in this direction.

Note

This document and its previous versions may contain imprecisions and inaccuracies that we will try to correct. Moreover, the authors reserve the right to extend and modify the material presented here as new information is gathered and as old information becomes out of date. The latest version of this documents will always be available at `http://www.cs.unipr.it/~bagnara/Papers/Prolog-FLI-survey` and `http://www.clip.dia.fi.upm.es/papers/Prolog-FLI-survey.ps.gz`.

1 Introduction

Given the wide range of goals and environments a programmer has to deal with nowadays, it is difficult (not to say impossible) to find a single language able to cope satisfactorily with all the heterogeneous needs appeared in the last years. While it is perfectly possible to write expert systems in Fortran and perform massive numerical computations in Prolog, choosing an inadequate language for the task at hand can significantly increase the costs of software development and maintenance, and even compromise the successful end of a project.

There are in practice several reasons why a language seemingly well-fitted for a particular purpose cannot be selected: technically obtuse management, lack of appropriate knowledge or culture (this is especially true for paradigms, such as logic or functional programming, where a particular *forma mentis* may help considerably), lack of appropriate programming environments, etc. Another reason that may prevent choosing the right language is the lack of standard foreign language interfaces, which would be used to patch weaknesses in a host language. Suppose that a complex application, to be written in its largest part in C, has some inference engine as one of its components. In principle, Prolog could be the best language to develop the inference part. However, if the interface between C and Prolog has serious drawbacks (e.g., it does not provide the right functionality, or it is non-standard and/or non-portable, or it is poorly documented), very likely Prolog will succumb and the inference engine will be written in C.

The number of already existing systems with foreign language interfaces witnesses that the technical difficulties of implementing them have been overcome. The right shape of the interface and its functionality is a matter of careful design and refinement; in fact the very basic facilities should be, understandably, much the same in different interfaces. But a widely accepted, homogeneous form of a foreign language interface is still a pending subject. It is not that the very basic features (i.e., accepting Prolog terms as input arguments, traversing these terms, creating new ones, and returning them as a result) are not present, but rather that they come in many different flavors. More advanced characteristics (such as raising exceptions or calling Prolog back from C) are not always implemented, despite that they are needed in many applications.

While Prolog interfaces for different languages may choose to adopt different views (in order to take advantage of the characteristics of each language), there should be a consensus for making the interface to a single language as compatible as possible across different vendors. One of the concerns of software production managers is the maintainability of a product: the question "What would happen if Prolog X disappears in one year? How difficult (and feasible) would it be to port my product to Prolog Y?" will have a more positive answer if there is a common ground about what is provided by those Prolog systems. Note that an unsatisfactory answer may mean not selecting any Prolog system at all rather than opting for a Prolog system instead of for another one. The ISO Prolog Standard has made (and is making) steps toward homogenizing the language, thus increasing the possibility of Prolog being selected. In the heterogeneous world computer science lives now, a similar action should be made with respect to foreign language interfaces.

In this piece of work we will pay attention to some characteristics of foreign language interfaces for C, study what is available in some well-known Prolog systems, summarize the results, and draw some conclusions. We aim, on one hand, at finding, the "average state of the practice", and, on the other hand, at pointing out which features need standardization more acutely. Our choice of C is motivated by the number of Prolog systems that have a C interface and because many of the characteristics to be studied are actually valid for a wide range of languages that are compiled and linked in a similar fashion to C. Moreover, there are several programming languages' implementations that provide a C interface so that they can all be interfaced to Prolog via C.

This paper is necessarily incomplete: real experiences have been taken into account only up to a limited extent. This means that we will not dig into possible implementation bugs: we will just trust the reference manuals of the vendors, and discuss the ideas behind that interfaces.

2 Systems Under Analysis

This survey focuses on a series of well-known Prolog implementations; the authors chose some of those that have free WWW access to the documentation, or that were available to the authors for other reasons. Time restrictions forced us not to be as comprehensive as we would have desired. It may also be the case that the systems under study changed some of the facilities available during the time elapsed between the moment the survey was compiled and the moment it was published. We apologize in advance for inaccurate or out-of-date data, and also for not including all the Prolog environments available, and encourage Prolog system producers to get in touch with the authors to provide data or clarifications about their products. We would like to stress that this survey does not aim at coming up with a winner, but rather with a set of observations on existing foreign interfaces.

All the information pertaining to the Prolog systems we examined was found on the respective Web sites that, in turn, are listed in the `comp.lang.prolog` FAQ, posted once a month to `news:comp.lang.prolog` and currently also available at `http://www.cs.kuleuven.ac.be/~remko/prolog/faq/`. The environments evaluated, and their URLs (which we refer the reader to for further information), are:

- B-Prolog (`http://www.cad.mse.kyutech.ac.jp/people/zhou/bprolog.html`)
- BinProlog (`http://www.binnetcorp.com/BinProlog/`)
- Ciao Prolog (`http://clip.dia.fi.upm.es/Software/Ciao/`)
- Eclipse (`http://www.icparc.ic.ac.uk/eclipse/`)
- GNU Prolog (`http://gnu-prolog.inria.fr/`)

- SICStus Prolog (`http://www.sics.se/sicstus/`)
- SWI Prolog (`http://www.swi-prolog.org/`)
- XSB (`http://xsb.sourceforge.net/`)
- YAP (`http://yap.sourceforge.net/`)

3 Features Taken into Account

Identifying which characteristics are those more relevant from a user point of view was perhaps the hardest part of all the survey. Besides some subjectiveness in this task, there are a number of facilities that almost every user would find desirable in order to ease the interfacing task. We will review here the points we have paid attention to, and why. We also mention some characteristics which, not having made it into the final table, are worth pointing out.

3.1 Data Interface

Prolog systems have their own internal (tagged) data representation; this has the implication that passing around any data types needs a conversion. Simple data types, such as integers, that exist both in Prolog and C make no exception. A Prolog system should provide convenient conversion operations. These are usually implemented by fragments of code (known as *glue code*) that are also in charge of testing types, checking the instantiation state, raising runtime errors, etc. Glue code can be hand-written or automatically generated; in any case it is mainly composed of calls to interface operations.

Simple data types can be given a C-level representation (e.g., C integers for Prolog integers, C strings for Prolog atoms) or passed to C as references to the corresponding Prolog object (e.g., an atom as a pointer to the atom storage in Prolog, where the corresponding string is to be found). More general data types cannot be given a straightforward C representation, and thus the commonly used approach is to offer an interface to construct and consult such data. Figures 1, 2, and 3 give an idea of how C functions to construct a list of integers from m to n could look like. The integers m and n are supposed to have been previously *unboxed* out of their Prolog representation.

Besides the names given to the operations in the interface, there are different flavors in which they may come. A *functional* interface (we apologize for this abuse of terminology) returns the data as function results, and often helps to have compact C code (see Figure 1). A *procedural* interface usually needs intermediate variables and more lengthy code, but it offers a way to return error codes very similar to what a C programmer would expect (Figure 2). Both can however be reconciled by selecting a special, non-valid term to denote an error situation (Figure 3). While comparable, they are clearly incompatible.

In addition to the functional/procedural difference, the interfaces can have been designed to be used top-down or bottom-up when constructing terms. That

```
Prolog_term from_to_list(int n, int m) {
  int i;
  Prolog_term tail = nil();

  for (i = m; i >= n; --i)
    tail = make_cons(make_integer(i), tail);

  return tail;
}
```

Figure 1: Constructing a Prolog list from C: A functional view.

is, if `f(g(a))` is to be created on variable `X`, should the order be `Y = g(a), X = f(Y)`, or is the other way around acceptable? While from the point of view of a Prolog programmer this order is immaterial, some C interfaces might not accept the seemingly dangling pointer introduced by issuing first the code for `X = f(?)` and then filling in the *hole* — or would need to set up an explicit free variable first as an argument to `f(?)`.

To summarize this section, although the general lines followed by the different interfaces are the same, there are several variations regarding how they are actually implemented and what they offer:

Automatic Type Conversion: Is it possible to specify cleanly which C data types are expected, so that glue code can be generated automatically? What language is used to specify it? How rich is set of allowed types?

Level: Does it provide an abstraction of the structures, or does it have a more *low level* flavor, perhaps with explicit reference to the *tags* usually associated to data inside the Prolog engine?

Functional or Procedural: Is the interface functional or procedural?

Top-Down or Bottom-Up: Do subterms have to be created prior to be used in term construction?

3.2 Control Interface

The quite generic term *control interface* actually groups several different issues: non-determinism, exceptions, the ability to call back Prolog from C, and automatic initialization and deinitialization.

Non-determinism, implemented as backtracking, is not directly supported by the C language. If the C code to be interfaced is to offer the possibility of giving different solutions for a call, the foreign interface should provide primitives to that effect — basically mimicking what a Prolog choice-point does. We have to point out that this possibility is not the more acutely needed in practice, since often existing C code does not exhibit such behavior. Moreover, non-determinism in C can be simulated by wrappers at the Prolog level.

```
#define CHECK(f) if (f == 0) return 0
int from_to_list(int n, int m, Prolog_term res) {
  int i;
  Prolog_term head, tail;
  new_Prolog_term(head);
  new_Prolog_term(tail);
  CHECK(put_atom(tail, nil()));

  for (i = m; i >= n; --i) {
    CHECK(put_integer(head, i));
    CHECK(put_cons(tail, head, tail));
  }

  CHECK(put_term(res, tail));
  return 1;
}
```

Figure 2: Constructing a Prolog list from C: A procedural view. In fact error returning should be more complicated and make untrailing, etc. This can however be left to the general *fail* mechanism of Prolog.

While a program whose control relies heavily on exceptions is arguably a badly designed one in any language (*exceptions should be exceptional*), error recovery by means of exceptions is a widely adopted technique. In order to take full advantage of it, C code should be able to raise exceptions on the Prolog side. Of course, a Prolog wrapper to throw exceptions upon receiving a designated result can be written, but the programmer may prefer not to put additional effort in designing an *artificial* interface.

Calling Prolog from C shares some common points with the Prolog-to-C interface: goals and arguments can be created and inspected using the same primitives. However a way to interpret a term as a goal, hand it to the Prolog engine, and recover the different solutions, is needed. Note that there are conceptually two levels here: one is just calling Prolog from a C application, and the other is calling Prolog from C when the C part has in turn been called from Prolog; the latter needs the Prolog system to be reentrant.

Some pieces of code may need explicit initialization and deinitialization routines, intended to be automatically called whenever foreign code is loaded and just before it is unloaded (or execution finishes for any reason). This is especially important in connection with interfaces to object-based languages, but it is useful in general: foreign code may allocate resources that ought to be deallocated, open files that must be properly closed, initialize transactions that need to be finalized for proper operation. Providing an elegant way to call these routines will help in developing certain types of applications in a robust way.

The control interface features to look at are thus:

```
#define CHECK(f) if ((f) == Prolog_term_error) return Prolog_term_error;
Prolog_term from_to_list(int n, int m) {
  int i;
  Prolog_term tail, head;
  CHECK(tail = nil());

  for (i = m; i >= n; --i) {
    CHECK(head = make_integer(i));
    CHECK(tail = make_cons(head, tail));
  }

  return tail;
}
```

Figure 3: Constructing a Prolog list from C: Functions with designated errors.

Non-determinism: Can non-deterministic C code be easily written with the interface provided?

Exceptions: Can Prolog exceptions be directly (and nicely) generated from C?

Calling Prolog from C: Can Prolog code call C? Can Prolog code call C in a reentrant fashion?

Init/DeInit: Can explicit initialization/deinitialization procedures be clearly marked?

3.3 Compiling and Linking Procedures

Different variations of the mechanism for compiling and linking will offer the programmer several degrees of automatization in the development environment. We believe that the system behavior should be as close as possible to what a programmer would perform by hand, in order for her/him to be able to arrange custom compilations for special cases. This does not mean that tasks to be done routinely (as recompiling old files) should be left to the programmer, but rather that they must be automated sensibly (e.g., *not* recompiling always everything from scratch).

In fact, a minimal (but usable) foreign language interface boils down to having a set of interface operations and header files with which the programmer can generate (by hand) an object file to be linked with already existing Prolog machinery. Starting from this basic scheme, there are several intermediate steps toward a fuller integration and a more programmer-friendly behavior. Care should be taken, however, that friendliness does not turn out to hinder flexibility.

Compiling Source Code: Does compiling of source code to object code have to be done manually, or is it automated? Is an additional tool used to take care of that task? In the case of automatic compiling, how does the Prolog system know where the source files are located? Is recompilation minimal? How is type conversion communicated to the glue code generator?

Linking Object Code: Once object/library files for the C code are available, how are they linked with the Prolog system? Three possibilities appear:

1. The source code of the Prolog engine has to be recompiled and linked with the new C object files to give a new engine.
2. As above, but the Prolog engine is also provided in the form of an object file. Recompiling is then avoided.
3. The generated C object files can be dynamically linked and loaded by the Prolog system. This facilitates program development.

From a practical aspect, these three differ in how much the development process is hindered. Considerations regarding the location of object code, similar to those for source code in the compilation stage, can be applied.

3.4 Miscellaneous

There are other points that, while not being crucial for the development of an application, can make that task easier. These belong perhaps to a more pragmatic classification because they do not affect the basic functionality of the system, but a sizable amount of time can be invested in a workaround for the lack or presence of some unexpected features. Apart from the points mentioned in the previous sections, we would like to highlight the following ones:

Naming Conventions: How is the mapping from C functions to Prolog predicates expressed? Is it tailorable, in order to take advantage of, e.g., predicates with different number or arguments? Assigning names, while not crucial in principle for the usefulness of an interface, can be of importance, if only because one might need to avoids name clashes between already provided C objects and existing Prolog code.

Final Application Deployment: How can a mixed-language application be packaged for distribution? Is there a relatively stable ABI so that binary distributions of packages can work reliably with different versions of the Prolog system?

O.S. Support: Which operating systems are supported? Reconciling and supporting different operating systems may be a technical issue, but it is certainly an issue to be solved for a supposedly platform-independent language.

Syst./Charac.	B-Prolog	Bin	Ciao	Eclipse	GNU	SICStus	SWI	XSB	YAP
Glue code	m	m	b	m	b	b	m	b	m
Int. style	f	l	f	f	f	p	p	p	f
Term constr.	t	b	b	b	b	b	b	t	b
Non-det.	n	n	s	n	y	n	y	n	y
Exceptions	n	n	y	n	y	y	y	n	n
C to Prolog	r	r	r	y	r	r	r	y	r
(De)Init	n	n	i	n	n	b	n	n	n
Compilation	m	m	a	m	m	e	m	a	m
Linking	s	s	d	b	s	b	b	s	d

Table 1: Summary of gathered information

4 Analysis of Some Systems

Table 1 summarizes our understanding of the C foreign language interfaces of the systems mentioned in Section 2.

Here is, for each of the features corresponding to rows in the table, the meaning of the symbols used in the table:

Glue code generation: **A**utomatic, **M**anual, **B**oth.

Interface style: **L**ow-level, **F**unctional, **P**rocedural.

Complex term construction: **T**op-down, **B**ottom-up.

Non-deterministic calls: **N**o, with **S**ome work, **Y**es.

Throw Prolog exceptions from C code: **Y**es, **N**o.

Calling Prolog from C: Yes and **R**eentrant, **Y**es but non-reentrant.

Initialization/Deinitialization: only **I**nitialization, **B**oth initialization and deinitialization, **N**one of them.

Compilation: **A**utomatic recompilation, **E**xternal tool used, **M**anual recompilation.

Linking: **D**ynamic, **S**tatic, **B**oth.

5 Some Preliminary Conclusions

The work presented here is very preliminary. Nonetheless, from this survey and from the work done and being done on interfacing the Parma Polyhedra Library (see `http://www.cs.unipr.it/ppl/`) with several Prolog systems, some considerations can safely follow.

5.1 Data Interface

In what concerns terms as abstract data types, there is a (natural) convergence, with two approaches: top down and bottom up. The bottom up approach is more widely adopted, presumably because it is more natural for interface users with an imperative programming background, and because it more easily allows for efficient implementations.

However, the semantics of the access functions is often different from one Prolog system to another. Those trying to interface the same application with different Prolog systems must be prepared to surprises. Moreover, same (or very similar) names in different Prolog systems are used to denote different things. For instance, there are foreign language interfaces where a *list* is a synonym for a term whose principal functor is *cons* (`'.'/2`), others where a *list* is either a *cons* structure or *nil* (`[]/0`), others that reserve the word *list* for properly terminated Prolog lists.

The automatic generation of glue code is interesting for simple interfaces involving simple data types. When things gets more complicated glue code must be written by hand since complete control is what is needed. This is the case, for instance, when foreign predicates acquire resources (e.g., they allocate memory) that must be released in case of failure: this cannot be done if output unifications are delegated to the automatically generated glue code.

5.2 Control Interface

Most of the characteristics of the foreign interface reviewed so far do not affect the Prolog side — i.e., how the C part is written should be, to a large extent, immaterial. However, some features of procedural (and OO languages) might call for a special Prolog support. In particular, initialization (and deinitialization) of, e.g., static data, might be automated and done more cleanly by stating which are associated functions with Prolog declarations. We note that the ISO Prolog standard only supports `:- initialization`, and a `:- deinitialization` whose associated goal is called when the module is unloaded would be useful in several cases — especially if a normal (i.e., not an abortion) program shutdown also unloads the modules. In that case, if the C code is seen at the same level as Prolog, foreign code initialization and deinitialization is just a particular case which can be handled by the Prolog mechanism.

5.3 Compiling and Linking

On the one hand, making the tasks of compiling and linking as automatic as possible may reduce the stress on the programmer and, in some cases, the time to catch bugs. On the other hand, it is important that the programmer is not trapped into the automatic mechanisms as this may be the source of even more frustration. There are foreign applications that may require particular tools for compiling and linking the C part, and each tool may need a particular set of options for the operation to be done exactly as the user wants it. As a consequence, there is no way an automated mechanism can do for all (present, let alone future) possibilities. For instance, it is inadvisable to restrict the choice of the C compiler to the one that was used to compile the Prolog system: that compiler may be unavailable (e.g., because the Prolog system was distributed in binary form) or it may have a bug affecting the C resource the user wants to interface. Given that the C compiler may change completely, the options it handles and the way it handles them as far as their relative ordering is concerned can also change completely: this reduces the things the Prolog system can count on to just anything. It is thus important that the automated mechanism that helps the user during the more routine work can be skipped altogether so as to allow the user complete control, that is, the same degree of control that can be exercised in developing the C part of the application. This is possible if, among other things,

- no assumption on the compiler is done besides that it follows the standard C ABI on the platform considered;
- standard C header and library file are provided *in standard places* so as to facilitate the production of the foreign objects.

6 Time to Standardize

As we have argued in the introduction, a well-designed standard is a good way for the work of a community to be accepted outside the community. Partly due to the expected application of a foreign interface, giving a homogeneous, well-known, widely accepted shape to it is a sign of technical maturity that will be appreciated by many people outside the Prolog fellowship. However, the reviewed systems are far from being compatible, although the basic characteristics are roughly the same — there are minor differences in the big picture, but they are incompatible after all. While it is perfectly understandable that implementors want to highlight a particular advantage or feature not offered by the rest of the developers, it seems to make sense that common services should be available using the same interface: that is, there should be a well-established minimal set of interface operations that ought to be adhered to by any vendor who wants to abide by some *de facto* (if not yet *de jure*) standard. We summarize some of the components of such a set.

6.1 Automatic Type Conversion

While it is true that automatic type conversion only works fully for simple cases, it is also true that many cases are simple. A conceptually easy to understand scenario should not need complex machinery to be dealt with. Points to be agreed upon in this respect are:

- Set of types to be reflected in each language.

- Which conversions are available (e.g., a list of character codes in Prolog can be seen as a C string on the C side, or also as a general term, depending on the purpose. The same holds for a C string, which can be converted into Prolog as an atom or as a list of character codes, depending on the final application).
- Language to express type conversions.
- Which type and mode checking is performed, and how should the system react when these are violated (related to this, see the **Exceptions** paragraph).

6.2 Data

In the more general case it is necessary to traverse and construct the data in the format used internally by Prolog, using an API interface. A minimal set should include: bi-directional conversion for basic data types (atoms, integers, and floating point numbers) in a way compatible with (a subset of) the automatic type conversion, type checking, and construction and traversal of complex structures. Error conditions should be detectable and recoverable.

6.3 Control

Calling Prolog from C (or from any other language) is in practice as relevant — and maybe more demanded— than the other way around. The data construction primitives are a preliminary need, to which transformation of terms into goals and a `call/1` counterpart in C is added, plus facilities to get additional solutions and to detect final failure and to discard further solutions. Also, if the runtime system cannot have the Prolog program linked and an external Prolog program is to be loaded, additional primitives to this effect are necessary.

6.4 Exceptions

These are the standard way to denote run-time errors, and should therefore be raised by the C part whenever an abnormal situation is detected. The API should therefore include a way to raise exceptions, and also a way to *easily* construct the different exceptions that appear in the Prolog Standard.

To Conclude

It is true that, today, there seems to be little agreement as to what is the best way to accomplish the minimal set of functionalities a foreign language interface should provide. However, the difference among current implementations is not insurmountable, and we hope that, if a consensus is reached, the effort to adapt existing Prolog systems to it will be, in the worst case, moderate and, in most cases, small.

Acknowledgments

Roberto Bagnara has been partially supported by project Adela (Italian-Spanish *Azione Integrata* IT 229) and by the University of Parma's FIL scientific research project (ex 60%) "Pure and Applied Mathematics".

Manuel Carro has been partially supported by projects Adela (Spanish-Italian *Acción Integrada* MCYT HI2000-0043), Amos (EU Project IST-2001-34717), and EDIPIA (Spanish Project MCYT TIC 99-1151).