Syntactic integration of external languages in Prolog

Jan Wielemaker[1] and Nicos Angelopoulos[2]

[1] Web and Media group, VU University Amsterdam,
De Boelelaan 1081a,
1081 HV Amsterdam, The Netherlands,
J.Wielemaker@vu.nl
[2] Netherlands Cancer Institute,
Amsterdam,
The Netherlands,
n.angelopoulos@nki.nl

Abstract. Todays applications are typically programmed in multiple languages, using SQL to access databases, JavaScript to make the (web-based) user interface interactive, etc. Prolog can cooperate to this orchestra using two views: as a logic server component or as 'glue'. In this article we concentrate on the 'glue' view, which implies that we must be able to 'talk' the language of other components. In one scenario, code snippets, such as SQL queries or HTML output, are embedded as strings in the Prolog program. Using strings however is slow and creates insecure programs that are hard to debug. Alternatively, one can use Prolog terms to represent foreign 'objects'. This latter approach has been in use for a long while in the Prolog community. In this article, we give an overview of the design choices that are available and discuss their consequences based on our experience.

1 Introduction

A modern (Prolog) application often has to interact to the external world and this interaction is increasingly based on the exchange of messages with other languages. The oldest and most known example is SQL. To query a relational database we construct the text for an SQL query from a skeleton and additional (parameter) information. This SQL query is sent to the database and the result is translated into Prolog terms using, for example, an ODBC wrapper. Other examples are (a) web-based applications generating HTML and Javascript, (b) generating graphical representations of a graph by creating a description thereof in the *dot* language, using graphviz [3] to render this and (c) interfacing to the **R** language for statistical operations.

In many languages such tasks are performed by creating strings, often concatenated from smaller building blocks or using printf() like skeletons where details are filled from runtime parameters. For example, we can use the C code below to create an SQL query to find all persons that live in a given city.

```
sprintf(cmd, "SELECT * FROM Persons\n"
             "WHERE City='%s'", city);
```

This approach is problematic. It is at the heart of what is commonly referred to as *SQL injection*. Such an injection is established by exploiting syntax features such as quoting rules of the target language. For example, we can use the value `"xxx' OR City LIKE '*"` for city, getting results for all cities rather than just a single city. Another disadvantage of this approach is that creating, but especially parsing textual expressions is a costly process in terms of CPU usage. Both misinterpretation and the CPU overhead can be avoided if the target language provides a structured alternative to the textual interface (e.g., a binding to the C language). Finally, using strings that express messages in another language often leads to inelegant, hard to maintain programs that do not take advantage of the flexibility of Prolog's data representation.

Nevertheless, the text-based approach is popular for at least two reasons. First, it is easy to understand, especially for novices and second, alternative approaches with a user-friendly syntax are hard or impossible in most statically typed languages. A further possible cause for this popularity is that it is more immediate, straight-forward to implement at the start. It is often the case that such integrations are born by the necessity of specific projects, that want the job done in the minimum of time. There is rarely an interest within specific projects to create well functioning interfaces.

However, Prolog is a dynamically typed language and allows for modifying its syntax using operator declarations. These properties allow for representing expressions of target languages as Prolog terms using a syntax that is equal to or closely related to the target syntax.

The described examples have been published elsewhere. The contribution of this article is that it provides an overview of what we consider the main design options and it contains suggestions on how the Prolog syntax could be extended to accommodate a wider range of languages.

In the remainder of this article, we will describe a number of examples and our experience with these. In section 8 we summarise the relevant dimensions to consider when developing a Prolog interface to an external language.

2 SQL

Connecting Prolog to relational databases using SQL is extensively studied. Many Prolog systems contain a low-level database interface that allows for executing an SQL query represented as text and receive the results as a Prolog term, usually row-by-row on backtracking. This interface makes the complete power of SQL available to Prolog, together with the disadvantages of text-based interaction described in the introduction. Some interfaces, e.g., SWI-Prolog's **odbc_prepare/5**[3] allow for *prepared statements*. Prepared statements are executed using their handle and parameters, avoiding SQL injections.

Christophe Draxler [2] designed a still widely used high-level interface to access relational databases. The core idea is to relate database tables to predicates. Next, a normal Prolog body-term created from these predicates and a subset of normal Prolog goals is translated into an SQL query. The advantage of this is that the Prolog programmer

[3] `http://www.swi-prolog.org/pldoc/man?predicate=odbc_prepare/5`

does not even have to know SQL to pose complex queries to the relational database. The disadvantage is that not all SQL features can be expressed in this language, for example it is not possible to create a new table through this interface. Also, Prolog's approach to access tuples by argument position is problematic for accessing tables with many columns.

An alternative approach is NED [5], where the users use Prolog terms as an abstract representation of a query, where the expressivity is tailored to fit context-specific requirements. The disadvantage of this type of approach is that it introduces a complete new language that needs to be documented and studied in detail, both for Prolog and database experts.

3 HTML

HTML (or XML or SGML) is a markup language rather than a programming language, i.e., it cannot be executed. HTML documents are a frequently used component in web-based applications. Although the security consequences of invalid HTML are less severe than invalid SQL, a toolchain that produces guaranteed valid HTML is still valuable.

One of the early implementations for processing HTML documents as Prolog terms is Ciao's PiLLow [4] library. SWI-Prolog's HTML generation library [9] is related, but exploits DCGs that allows the user to define components that can be reused in different HTML pages.

Dealing with these markup languages using Prolog terms works well because the underlying document model is a tree where each node is either plain text or is a node with an element name (e.g., `span`, `div`, `h1`), a set of attributes and an ordered list of child nodes. Although the concrete HTML syntax (e.g., `<span class="c1">Hello World</span>` cannot be turned into valid Prolog, the underlying datamodel can easily be expressed and the translation is so simple that it can be remembered easily by the user.

4 JavaScript and JSON

In addition to HTML, web applications often have to emit JavaScript. Larger and static pieces of JavaScript should probably be written in a separate JavaScript source file and be included using the HTML `script` element. Small code, such as code that initialises JavaScript objects based on the current document context, should be maintained together with the (Prolog) code that generates the web page. This implies we need a solution to embed JavaScript source code into Prolog.

If we could limit the JavaScript to calls with simple parameters, this is quite feasible and comparable to the Java, R and XPCE interfaces described in later sections of this article. Structured objects can be exchanged as JSON[4]. JSON is a simple serialisation syntax for structured objects. Its syntax is valid Prolog, but using this syntax has some drawbacks. All keys and string values are written using double quotes (e.g.,

[4] `http://www.json.org/`

verb"$name$" : "Joe"), where atoms result in a much more compact representation. If we change this, using atoms for keys and string values, JSON constants such as `null` and `true` become ambiguous. JSON support is part of the SWI-Prolog HTTP library and consist of two layers. First, we define a Prolog representation for a JSON objects using the following rules:

- A JSON object is mapped to a term `json`(*NameValueList*), where NameValueList is a list of Name=Value. Name is an atom created from the JSON string.
- A JSON array is mapped to a Prolog list of JSON values.
- A JSON string is mapped to a Prolog atom.
- A JSON number is mapped to a Prolog number.
- The JSON constants true and false are mapped -like JPL- to @(true) and @(false).
- The JSON constant null is mapped to the Prolog term @(null)

Here is a complete example in JSON and its corresponding Prolog term:

```
{ "name":"Demo term",
  "created": {
    "day":null,
    "month":"December",
    "year":2007
  },
  "confirmed":true,
  "members":[1,2,3]
}
```

```
json([ name='Demo term',
       created=json([day = @null,
                     month='December',
                     year=2007]),
       confirmed = @true,
       members=[1, 2, 3]
     ])
```

The predicates **atom_json_term/3**, **json_read/3** and **json_write/3** can be used to (de)serialise Prolog JSON terms. The second layer allows for defining a mapping between JSON terms and more commonly used Prolog representations. For example, a Prolog programmer would typically represent a point in a two-dimensional space using a term `point`(*5,10*). The typical JSON representation is `{"x":5,"y":10}`, possibly extended with a `"type":"point"` property. The `json_convert` library can convert between these two representations based on a declaration like this:

```
:- json_object
        point(x:integer, y:integer).
```

JavaScript allows for lambda functions, and these are commonly used to customise objects defined in reusable libraries. A common features is to register event hooks as lambda functions with more abstract classes. Lambda functions are specified in the full C-style syntax of JavaScript, while at their role in the program commonly asks them to be specified from Prolog where the details (constants) depend on the application context.

Many aspects of this language are currently invalid in Prolog, in particular consider functions without arguments (e.g., `myfunction()`), The relation between function head and body (`function(x, y) {...}`), array subscripts and the ⟨*object*⟩.⟨*method*⟩ notation. Even if such as mapping is feasible, we are unsure of its value considering our experience with expressing imperative code using Prolog syntax in section 5. Possibly the problem is less severe here because the JavaScript code executes in another agent (typically a web browser).

An alternative approach suggested in private conversation by Maarten van den Dungen is to use Prolog directly, using JScriptLog`http://jlogic.sourceforge.net/` to execute code in the JavaScript environment. This solves the syntax issues, but the user must be aware that the other Prolog code executes on a different engine, seeing a different set of predicates, including a different set of built-in features.

5 XPCE

Although not widely known we'll introduce XPCE, SWI-Prolog's graphics subsystem, as an example of a wider class of language interface challenges. XPCE ([8], reworked version in [7]), is an object oriented system, that allows objects to manipulated from Prolog through four basic predicates:

new(*-Reference, +Class(+Arg, ...)*)
send(*+Reference, +Method(+Arg, ...)*)
get(*+Reference, +Method(+Arg, ...), -Result*)
free(*+Reference*)

> In these predicates, *Reference* is a term @/1 that identifies an XPCE object and *Arg* is a reference, number, atom, term `Class`(*+Arg, ...*) or `new`(*Class*). The latter is needed to create an instance from a class without arguments because `Class()` is not valid syntax.

The above predicates allow for manipulating objects. XPCE can, similar to JavaScript, store executable code into object properties. Being designed for use with Prolog, executable code is expressed as normal objects. Code objects include conjunction, disjunction, negation, arithmetic, higher order operations on collections, etc. Below is a simple example of a button that colours a box when clicked. Note that the outer 'message' refers to a property (method) of class button and the inner 'message' to a class name.

```
...
new(B, button('Click for red box')),
```

```
send(B, message(message(Box, colour, colour(red)))),
...
```

XPCE takes the integration one step further by supporting subclassing of XPCE classes from Prolog. To do this, we use a Prolog syntax to describe a new subclass and its methods. The method body can use the full power of Prolog, although it is executed as **once/1**. A class definition is compiled into an XPCE class with proxy methods that call the Prolog implementations of the methods.

XPCE is a special case because the language was designed for the way it is embedded in Prolog. Notably, XPCE does not have a concrete syntax and can only be 'programmed' by managing objects using the four principal interaction functions.

XPCE/Prolog is a powerful framework for building applications, but it has a steep learning curve. In part, this is unavoidable due to the sheer size of graphical libraries. Other parts of the complexity are caused by the fact that you have two ways to store data: the Prolog way and in XPCE objects. Each can be manipulated through a language. Both languages are expressed in the same syntax, but their different semantics is confusing to programmers.

6 Java (JPL)

JPL,[5] developed by Paul Singleton, uses an approach to access Java from Prolog that is similar to XPCE.[6] Java is disclosed through the JNI interface, providing Prolog with means to create Java data structures. Next, the interface uses the Java reflection API to invoke methods on Java objects by name. This can be compared to the four principal predicates of XPCE.

Extending Java from Prolog is not provided by JPL. In theory, the class-extension mechanism used with XPCE/Prolog can be implemented by generating, compiling and loading Java code that implements the required proxy classes. This would allow Prolog programmers to extend Java class libraries without much knowledge of Java. Given that Java is has a widely known syntax, it is doubtful that many people would like to create Java classes through using Prolog syntax. Creating Java (proxy) classes from Prolog, where the methods are executed in Prolog may provide a promising alternative for making Prolog available from Java.

7 R

.

The R language for statistical computing provides a rich computational framework, a programming language with a functional flavour and library of primitives for graphics, mostly diagrams. The r..eal [1] Prolog to R [6] interface went through two iterations. The first realisation used R as a slave process with pure textual exchange of results. The

[5] `http://www.swi-prolog.org/packages/jpl/`

[6] In addition, JPL provides an interface to access Prolog from Java that is loosely based on the SWI-Prolog C++ interface. We consider that irrelevant to the discussion in this article.

second generation interface, r..eal, uses, like JPL and XPCE, a native interface. The system supports three basic interactions on the <− operator, as illustrated in the session below:

```
1 ?- [library(real)].
true.
2 ?- a <- [1,2,3].
true.
3 ?- <- a.
[1] 1 2 3
true.
4 ?- A <- a.
A = [1, 2, 3].
```

The first line assigns a Prolog list as a vector to an R variable. The second prints the result of an R expression and the last converts the result of an R expression into a Prolog term. The interface describes a mapping between R and Prolog terms. Vectors are mapped to flat lists, matrices to singly nested lists, R's named list to Prolog pair lists and function calls to term structures. Communication of large data structures is facilitated via an efficient low level interface based on the C language interface of the two systems. Calls to R functions result in translation of the Prolog term into a string that is evaluated by R.

Unlike with JavaScript, mapping from Prolog terms to R expressions works well due to the simple uniform functional syntax of R. The result is a powerful interface. However, it suffers from several issues, some fundamental and some less so:

- While primitive Prolog data is converted through the R interface to C, arbitrary terms are processed as text. This results in very different performance results. Consider the code below. In the first call we convert a Prolog list to an R array (no text involved) and in the second, we create a complete textual representation (`"mean(c(1,2...,100000))"`)

```
1 ?- numlist(1,100000,L), time((a<-L, M<-mean(a))).
% 60 inferences, 0.006 CPU ...
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
M = 50000.5.

2 ?- numlist(1,100000,L), time((M<-mean(L))).
% 2,088,940 inferences, 1.129 CPU ...
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...],
M = 50000.5.
```

- R variables are global and can be destructively assigned to. While this provides the -often wanted- global variables to the Prolog/R infrastructure, it can be confusing the Prolog programmers.
- The syntax mapping requires some tweaks to avoid ambiguity. These include the use of dot character in names and the empty argument list notation. The interface circumvents these by adding a '.' in both cases. Thus `var.name` becomes

`var..name` and `foo()` translates to `foo(.)`. The integration would be enhanced if the original code can be handled from within Prolog. A third difficulty arises from the syntax commonly used in many languages for accessing arrays. `a[i,j]` is invalid in Prolog. R..eal uses a current operator to legalise the syntax by: `a^[i,j]`.

8 Criteria for designing an interface

Above, we discussed several examples that make external languages accessible from Prolog. Although we cannot claim that the presented solutions are the best possible, all these systems have gone through multiple iterations to reach at their current design.

All the described interfaces represent 'objects' of the target language using Prolog terms. In part, this may be influenced by Prolog's poor abilities to represent text, which is limited by the following:

- Lack of a long string (as, for example, Python's `"""..."""` syntax). This makes it hard embed strings that require quotes in a Prolog text. Note that the ISO Prolog standard allows both for doubling quotes to escape them and the backslash notation, which complicates the introduction of long strings.
- Complicated multi-line string syntax. Line-breaks need to be escaped with a backslash and the next line must start at the left margin to avoid additional white-space in the output.
- No simple concatenation syntax, neither between constant string fragments nor with number and other constructs.
- Representing text as atoms is limited by atom-length in many implementations. Representing text as strings wastes space on most implementations.

8.1 Choosing a Prolog syntax for the target language

We distinguish several options for representing the target language in Prolog:

- If the target language is (almost) valid Prolog syntax, using Prolog syntax that is either equivalent to the target language or introduces only small easy to understand systematic transformations, this is an attractive option. This approach is easy to learn and does not require extensive documentation. The r..eal interface described in section 7 is an example. Additional operator declarations can help bridging the gap. Commonly encountered problematic syntax features are the empty parameter list, represented as "`()`", array subscripts (e.g., `a[1]`), uppercase identifiers, distinguishing quoted strings from symbols (JSON) and the dot (.) being used in identifiers or to separate methods from objects.
- In some cases, the syntax is complicated, but the datamodel is simple and can easily be represented in Prolog. The discussed markup languages (section 3) form a good example.
- In some cases, we can represent a fair deal of the intended semantics using our familiar Prolog language and compile Prolog to the target language. This is the case

for the SQL interface by Christophe Draxler [2] (see section 2). His SQL mapping results in a uniform syntax and semantics at the price of not being able to access *all* functionality of the target language. For example, it is not possible to create a new table through this interface.

Fortunately, many special purpose languages can be handled by one or more of the above options. An exception is JavaScript, which has a complicated C-like syntax. Inventing a Prolog datastructure that captures all functionality of the language and is easy to understand by a Prolog programmer with some JavaScript knowledge is probably not feasible.

8.2 How does the semantics of target language relate to Prolog?

Relation between the semantics of both systems is also important. We see that a close semantic relation (SQL) can be used to find an alternative to a syntactic mapping. We also see that strong syntactic integration between two systems that have large semantic differences (R, XPCE) can lead to confusion.

8.3 Technical integration

In many of the above described examples (SQL, HTML, JavaScript), we must ultimately produce a string serialisation for the target language. In others (XPCE, Java, R) generation of text can be avoided. In other words, the target language can be linked into the same process and the systems can communicate using low-level communication that typically uses the C language as intermediate.

9 Discussion

Many of today's applications are built from components that use different programming languages and markup languages. There are roughly two ways in which we can make Prolog play in this orchestra: (1) as a logic component or (2) as 'glue', making Prolog talk to multiple different components. Some systems (e.g., Amzi!) concentrate explicitly on a minimal role as 'Logic Server'. Most system make no clear choice and SWI-Prolog probably represents the most extreme view in pushing Prolog as a 'glue' language.

Composing text of another language using simple string manipulation is often undesirable because the actual goal is to create structured objects in the target environment (let it be data structures or executable objects). Simple composition of text fragments easily leads to syntax errors that are hard to detect in the development environment or, worse, security vulnerabilities (e.g., SQL injections).

As we have seen, Prolog is rather poor in representing (long) textual expressions, but it is good in using concise syntax to build complex datastructures of which the syntax can be optimised using operators. This approach works for languages with a uniform syntax that is easily expressed (R), languages with a simple datamodel (HTML) or languages that are semantically close, such that we can translate a subset of pure Prolog into the target language (SQL).

Avoiding limits in Prolog supports this approach. I.e., unbounded integers can comfortably represent integers from any language. Compound terms with unrestricted arity allows representing any table row or array as a compound term and atoms with unlimited length and Unicode characters can represent any string. These features can all be implemented within the ISO standard.

It is feasible to extend Prolog syntax to improve the transparency of these language mappings, avoiding the need for work-arounds such as discussed in section 7. In particular, we lack the following constructs:

name()
: This is currently a syntax error in Prolog. Adding it introduces some ambiguities in e.g., **functor/3**. Otherwise, **atom/1** can fail on this and **compound/1** can succeed.

a[x]
: I.e., array syntax. This is already supported by e.g., B-Prolog.[7] In B-Prolog, `X[I1,...,In]` is a shorthand for `X^[I1,...,In]`.

name(arg...) { **body** }
: It is unclear how to turn this into valid syntax, but currently it is always a syntax error because neither a compound term, nor a {...} term can be an operator and the Prolog syntax does not allow for a sequence of two non-operator terms. This implies that it is possible to make this term legal and, like the array subscript notation discussed above, map it to a Prolog term.

a.b
: I.e., '.'-separated sequences. These can in part be supported by defining '.' as an infix operator, but this creates ambiguous syntax with floating point numbers. Handling the dot as an operator causes such identifiers to show up as lists, which may easily lead to ambiguities. Possibly, `a.b` can be handled at the tokenization level. For example, it could be a valid unquoted atom.

Acknowledgements

[Acknowledgements]

References

1. Nicos Angelopoulos, Vitor Santos Costa, Jan Wielemaker, Joao Azevedo, Rui Camacho, and Lodewyk Wessels. R..eal: A library for statistical AI. Technical report, Netherlands Cancer Institute, 2012.
2. Christoph Draxler. Accessing relational and higher databases through database set predicates in logic programming languages. Phd thesis, Zurich University, 1991.
3. John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, and Gordon Woodhull. Graphviz open source graph drawing tools. In Petra Mutzel, Michael Jnger, and Sebastian Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 594–597. Springer Berlin / Heidelberg, 2002. 10.1007/3-540-45848-4_57.
4. Daniel Cabeza Gras and Manuel V. Hermenegildo. Distributed WWW programming using (ciao-)prolog and the piLLoW library. *TPLP*, 1(3):251–282, 2001.

[7] `http://www.probp.com/manual/node47.html`

5. Frederick Maier, Donald Nute, Walter D. Potter, Jin Wang, Mayukh Dass, Hajime Uchiyama, Mark J. Twery, Peter Knopp, Scott Thomasma, and H. Michael Rauscher. Efficient integration of prolog and relational databases in the ned intelligent information system. In Hamid R. Arabnia, editor, *IKE*, pages 364–369. CSREA Press, 2003.
6. R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Found. for Stat. Comp., Vienna, Austria, 2012.
7. Jan Wielemaker. *Logic programming for knowledge-intensive interactive applications*. PhD thesis, University of Amsterdam, 2009. http://dare.uva.nl/en/record/300739.
8. Jan Wielemaker and Anjo Anjewierden. An architecture for making object-oriented systems available from prolog. In *WLPE*, pages 97–110, 2002.
9. Jan Wielemaker, Zhisheng Huang, and Lourens van der Meij. Swi-prolog and the web. *Theory and Practice of Logic Programming*, 8(3):363–392, 2008.