SWAPP: A Lightweight Semantic Web Application Platform Based on Prolog

TORBJÖRN LAGER
University of Gothenburg
Sweden
(*e-mail:* lager@ling.gu.se)

JAN WIELEMAKER
VU University Amsterdam
The Netherlands
(*e-mail:* J.Wielemaker@cs.vu.nl)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

We describe the design and development of a Semantic Web application platform in which the Prolog programming language plays no less than four different roles: as a query language, as a rule language, as a server-side web application programming language, and as the language in which the platform as a whole is implemented. The platform is prolog-ish also in that its RESTful web API supports the a-tuple-at-a-time retrieval of solutions to a query typical of Prolog and in that solutions are given in the form of Prolog variable bindings, encoded as JSON. A client may furthermore POST Prolog clauses to the service to be stored in a scratch module associated with the current session and used in subsequent queries, something which greatly increases the power of the querying facilities. The platform thus provides a way to work with RDF and friends which is both powerful and practical, and which also allows developers to easily define proprietary extensions to overcome current limitations of the RDF framework.

KEYWORDS: Prolog, Semantic Web

1 Introduction

Prolog has a number of traits that distinguishes it from most other programming languages:

- Declarativeness, inherited from its roots in logic
- Inference capabilities, inherited from its roots in theorem proving
- No separation between data and programs (reflexivity)
- The lazy a-tuple-at-a-time generation of solutions to queries

Its declarative nature and its inference capabilities places Prolog closer to the semantic web than most other general programming languages. In this paper we

intend to show that the closeness between data and programs as well as the a-tuple-at-a-time generation of solutions to a query also make a lot of sense in the context of a web application platform.

We have developed a semantic web application server featuring

- A RESTful web API for querying and managing data
- Querying of RDF graphs through Prolog queries
- Efficient a-tuple-at-a-time generation of query results
- Query results as Prolog variable bindings encoded in JSON
- Rule-based inferencing well-integrated with application code
- Rule-based ad-hoc inferencing in the context of querying
- Simulated server-push of events for the purpose of monitoring etc.

To some extent, we have been inspired by *Persevere*[1], described as a platform supporting rapid development of data-driven JavaScript-based rich internet applications. Persevere is a JSON storage and JavaScript application server, offering schema-free persistent storage, a secure RESTful JSON interface for data interaction, and JSONQuery/JSONPath for fast ad-hoc querying. A rich interactive server side JavaScript environment (running on Java/Rhino) is accessible via JSON-RPC, meaning Persevere works great for serving both *thin* and *thick* AJAX front ends.

However, Persevere is not a *semantic* web application platform, since it uses JSON for storage, rather than RDF, and since no inference machinery whatsoever is available, over and above what JavaScript provides. What makes Persevere inspiring is its lightness and flexibility, its support for recent standards and practices such as JSON and RESTfulness, as well as its focus on easy deployment.

Platforms that *do* describe themselves as semantic web application platforms, such as the Talis Platform and Freebase(?), are typically hosted RESTful web services which provide RDF data management, SPARQL querying and search features, but not a general server-side application development programming language. This only makes them suitable as storage backends to *thick* AJAX applications, where the application code runs on the client.

We see many potential application areas for a lightweight Semantic Web and AJAX application server based on Prolog. Natural language parsing web services for example, with parsers implemented in Prolog and lexica stored in RDF. Or expert system web applications, representing domain data in RDF. In section ?? we describe one particular use: a corpus workbench representing and storing corpus data as well as other linguistic resources in RDF and using Prolog rules and queries for the navigation and analysis of data.

TORBJORN: Comparison with Cliopatria – explain in what sense SWAPP is more lightweight than Cliopatria (no templating, leaves the choice of Javascript library to the user, etc.) and what it has that Cliopatria does not have (a-tuple-at-a-time generation of solutions, etc.) Explain also why Cliopatria is superior for the things that it was built for.

[1] http://www.persvr.org/

2 Components

SWAPP is built from components most of which are shared with Cliopatria. We mention the most important ones here:

- *RDF (?)*
  The RDF support consists of parser 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.
- *HTTP (?)*
  The HTTP server handles concurrent requests and includes authorisation and session management.
- *JSON*
  Provides a bidirectional generic translation between a JSON string and a ground Prolog term.

3 Query language

Prolog provides a natural interface to schemaless semantic web data in the form of RDF. The central predicate is `rdf/3`, with the obvious interface `rdf(Subject, Predicate, Object)` which matches an edge in the RDF graph.

Finding a subgraph with certain properties is now easily expressed as a Prolog conjunction, for example

```
rdf(Author, ex:fullName, literal(Name)), rdf(Report, ex:author, Author).
```

We do not use SPARQL. Instead, queries by the application logic are expressed as Prolog goals on the raw RDF database and/or RDFS/OWL reasoning modules. At places where the order of executing conjunctions is critical and cannot easily be predicted by the application programmer, we use the query optimiser which rewrites a Prolog goal involving multiple calls to `rdf/3` and tests for optimal performance. Semantic Web query languages are not used in the application logic because

- Prolog itself already provides a completely transparent and easy to use API. As the application programmer uses Prolog anyway, Prolog syntax is a natural choice.
- SPARQL takes the "a-set-at-a-time" approach to evaluation but we require "a-tuple-at-a-time" evaluation.
- SPARQL lacks expressiveness to construct complex path expressions. For example, SPARQL does not support regular expressions in query paths, therefore, there exists no query that gets the root of a resource given a transitive property.
- For our purpose we often need specific RDFS/OWL reasoning support. Partial reasoning that fulfil our requirements is easily implemented and performs well.

4 Query API

Initially we designed and implemented, on top of HTTP, a simple protocol for communicating with a Prolog process running on the server. We referred to it as the *first-next*-stop* protocol since the exploration of a Prolog goal had to start with a `/first?query=` request, be followed by zero or more `/next` requests and end with a `/stop` request. It allowed the client to control the process in a procedural manner, by sending *commands* to it. In recent years, however, the REST philosophy has gained a lot of ground, emphasizing the importance of meaningful URIs and the versatility of HTTP, and correctly pointing out that there usually is no need to build special purpose protocols on top of HTTP. We took this to our hearts and now believe that embedding Prolog in the web demands a RESTful interface and that it would indeed be ironic if Prolog, being essentially a declarative language, only provided a procedural web interface.

Still, the fact remains that Prolog is a relational, nondeterministic programming language, meaning that a query may have more than one solution and frequently very many and sometimes even an infinite number of solutions, which we can lazily iterate over using Prolog's backtracking mechanism. Unfortunately, this does not mesh well with the way the web and its protocols work. We have 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 an all-solutions predicate such as `findall/3`, but this only works for a finite number of solutions and only if they are not too many. Besides, we may *prefer* to generate the solutions one-by-one, sometimes because it is cheaper in terms of memory requirements (on both server and client), and sometimes because we want to decide, after having seen the first couple of solutions, whether we want to see more. If not, we may have saved ourselves some time and some CPU cycles.

We choose instead to work with a *virtual index* to the solutions that a query has, without actually generating the solutions. Each solution in the sequence of n solutions to a query receives an integer index in the range $1..n$. This makes a query for the ith solution of a goal functional and deterministic, and thus solves the impedance mismatch problem.

Presenting this as a RESTful API is of course trivial.

Method	**URI Pattern**	**Semantics**
GET	`/swapp/session/db?query=`q`&i=`i	Retrieve the ith solution to the query q

URIs can now be seen as declarative *expressions* rather than commands, and each solution to a query has become a *resource* (in REST parlance), with the consequence that it is uniquely addressable by a URI, which in turn means that it may for example be *bookmarked* (using the bookmarking mechanism available in browsers), something which certainly isn't possible with URIs such as `/next`.

5 Caching

Implementing the semantics of this API:s is not difficult. The problem is getting an implementation to behave efficiently, in particular for certain repeated requests. Consider the following two requests:

```
GET /swapp/session/db?query=rdf(S,p,O)&i=1
GET /swapp/session/db?query=rdf(S,p,O)&i=2
```

In a naive implementation, work that `GET /swapp/session/db?query=rdf(S,p,O)&i=2` has to perform will have to repeat the work performed by `GET /swapp/session/db?query= rdf(S,p,O)&i=1`. To deal with this, we have developed a technique to preserve the Prolog state (stack, choicepoints, variable bindings) between requests by creating a thread that is associated to the HTTP session, running the state-full computation there, and sending messages back and forth between the HTTP handlers and the session thread to communicate queries and results.

Under the assumption that we only want to manage one Prolog state (per client session) at a time, this raises the problem of when to take advantage of the preserved Prolog state, and when to reset it (by killing the thread in which the query is running). Intuitively, the Prolog state needs to be reset if the query changes, or if the request is for a solution earlier in the sequence of solutions to the query, but that it otherwise may be of use. For example, if the two GET requests above are followed by

```
GET /swapp/session/db?query=rdf(S,p,O)&i=3
```

then it is indeed the case that the Prolog state can be used, and that doing so will save a few logical inferences. But if they are followed by a request such as

```
GET /swapp/session/db?query=rdf(S,p,O)&i=1
```

or by a request with a different query

```
GET /swapp/session/db?query=rdf(S,q,O)&i=1
```

then we should reset the Prolog state before attempting to serve them. We can wrap this up as a rule:

IF the URI of the current request has the form `/swapp/session/db?query=`q_1`&i=`i_1
AND there exists a previous request R (in the current session)
AND the URI of R has the form `/swapp/session/db?query=`q_0`&i=`i_0
AND $q_1 = q_0$
AND $i_1 > i_0$
THEN the current Prolog state is used
ELSE the current Prolog state is reset before the current request is dealt with
ENDIF

Note that the Prolog state plays the role of a kind of *cache*, and that the act of resetting the Prolog state is equivalent to the act of *flushing* this cache. The comparison to a cache seems natural also in light of the fact that all that is does is to (under certain conditions) increase the efficiency of the server. In particular, it does not change the semantics of the API.

To take care of the problem of server threads "just hanging there" indefinitely, wasting valuable resources, we allow the cache to *expire* after a set number of seconds without activity (again by killing the thread in which the query is running). For example, if the set number of second is 60, then for the case of

```
GET /swapp/session/db?query=rdf(S,p,O)&i=1
... 61 seconds passing....
GET /swapp/session/db?query=rdf(S,p,O)&i=2
```

the Prolog state would not be preserved in between the requests. Note that the semantics is preserved though.

6 Output format

There are a number of formats in which solutions to a query could be returned: JSON, XML and Prolog, to mention the most obvious ones. A general web service framework should probably allow the user to select output format on a query-by-query basis (e.g. by specifying a request parameter) but our implemention is currently only able to return solutions encoded as JSON.

We convert any Prolog binding into a JSON term. Prolog lists are treated in a special way. Also, JSON terms are not converted. Here is the mapping:

- Variable → {"type":"var", "name":<string>}
- Atom →{"type":"atom", "value":<string>}
- Integer → {"type":"integer", "value":<integer>}
- Float → {"type":"float", "value":<float>}
- List → JSON array
- Compound → {"type":"compound", "fun":<string>, "args":<array>}

TORBJORN: Here we need an example.

7 Update API

Through library `http/http_session` SWI-Prolog supports the creation and management of sessions based on HTTP cookies. A session provides a session ID, a unique identifier known by the server as well as the client that initialized the session. When a session starts, SWAPP creates a module named by the session ID, and when the session ends, it is destroyed. This session database module thus serves as a scratch area, for the client to store any Prolog clauses deemed useful for subsequent querying or for other kinds of processing. The API is simple:

Method	URI Pattern	Semantics
POST	`/update`	Add a set of clauses to the session database

TORBJORN: PUT and DELETE methods? Safetey?

8 Working with the APIs

A Prolog query by itself is not more (nor less) expressive than a SPARQL query. However, by combining the querying ability with the ability to dynamically update the session database we end up with something quite powerful. Here, for example, is how we could compute the transitive closure of the `p` property. First we POST a definition

```
POST /update
ptrans(S,O) :- rdf(S, p, O).
ptrans(S,O) :- rdf(S, p, X), ptrans(X, O).
```

and then we GET the first solution

```
GET /swapp/session/db?query=ptrans(S,O)&i=1
```

and the second solution

```
GET /swapp/session/db?query=ptrans(S,O)&i=2
```

or all of them in a batch if we prefer:

```
GET /swapp/session/db?query=findall(ptrans(S, O), ptrans(S, O), PTs)&i=1
```

We could also update the session database with a Prolog procedure doing something interesting (not necessarily involving querying data), and call this procedure remotely from the client by sending a GET request. This is where Persevere makes a distinction between retrieving data through a standard JSON HTTP/REST web interface on the one hand, and the remote execution of JavaScript methods on the server through JSON-RPC on the other. Since Prolog does not distinguish data from programs (i.e. clauses are both data and programs), there is no need for an Remote Procedure Call (RPC) interface. The HTTP/REST web interface can be used for both the retrieval of data or for the invocation of procedures, either static procedures or procedures in the dynamic session database.

9 The API Explorers

Our Prolog-based semantic web service, like any HTTP/REST-based web service, may be tried and tested using command line tools such as *cURL*[2] or browser extensions such as the *Poster*[3] Firefox plugin.

[2] http://curl.haxx.se/
[3] http://code.google.com/p/poster-extension/

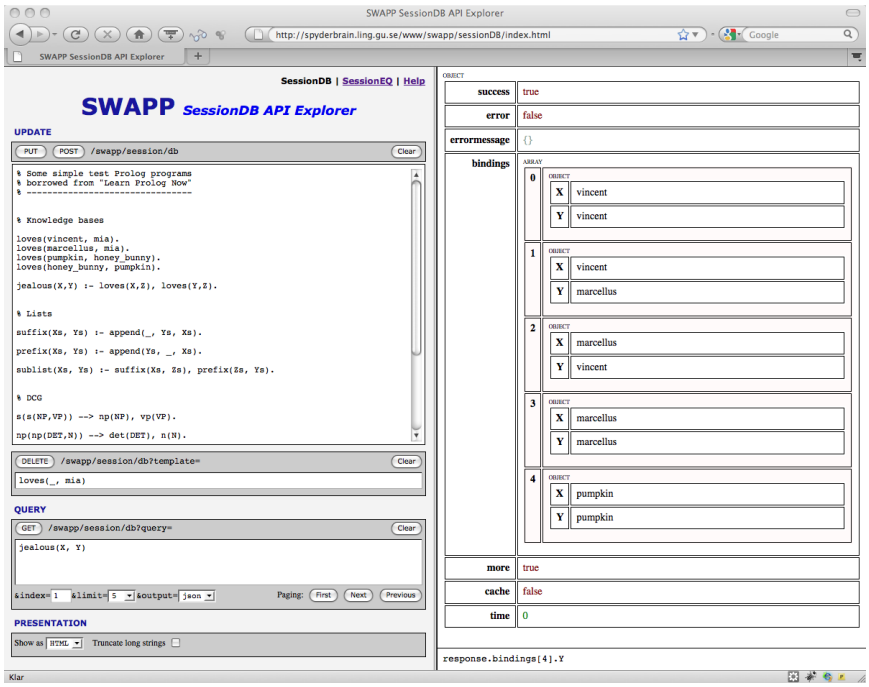

Fig. 1. The SessionDB API Explorer

10 Implementation details

TORBJORN: This may not be needed.

11 Use case: a corpus tool

Our Prolog-based semantic web service, like any HTTP/REST-based web service, may be tried and tested using command line tools such as *cURL*[4] or browser extensions such as the *Poster*[5] Firefox plugin.

Building a web application in a combination of HTML, CSS and JavaScript is straightforward. The GUI to a corpus tool written by the first author is shown in ??. Note the Previous button that allows a user to inspect the previous solution/match, albeit often significantly slower than looking at the next solution/match since the Prolog state caching does not work when moving "backwards".

It is important to note that on the client side, nothing is new or in any way peculiar to the use of Prolog. The client side is a just an ordinary AJAX application that can be written by any programmer familiar with the languages and techniques involved.

[4] http://curl.haxx.se/
[5] http://code.google.com/p/poster-extension/

12 Discussion

SWAPP is all about Prolog in, JSON out, and all about contributing to the "Logic Programmable Web".

Prolog stands for "programming in logic" and thus WebProlog stands for "web programming in logic".

It is time for Prolog programmers to learn JavaScript and for AJAX programmers to learn Prolog. Very few Prolog programs need a front end which isn't browser based, and AJAX applications needs to be smarter.

TORBJORN: To be written.

Acknowledgements