Web Prolog and the programmable Prolog Web

An Attempt to Revive and Rebrand Prolog

Torbjörn Lager

Draft printed on June 18, 2018

Contents

Chapter 1

Introduction

In this report, we describe a programming language called *Web Prolog*. We think of it as a *web programming language*, or, more specifically, as a web *logic* programming language. It is based on Prolog, with a pinch of Erlang sprinkled in. We stay really, really close to traditional Prolog, indeed so close that the vast majority of example programs in Prolog text books will run without modification. Towards Erlang we are less faithful, picking only those features from Erlang that we regard as useful in a web programming language. In particular, we pick the features that make Erlang into an *actor programming language*, and on top of these we define the concept of a *pengine* – a programming abstraction in the form of a special kind of actor which closely mirrors the behaviour of a Prolog top-level. However, Erlang has not been the only inspiration. When it comes to the *purpose* of Web Prolog, the inspiration has been JavaScript – a scripting language for the Web that is usually embedded in other programming systems and where processes must be sandboxed in order to allow the execution of untrusted code.

We also propose that the Prolog community should take it upon itself to develop a standardised infrastructure for a *Prolog Web*, a web more firmly based on logic than the traditional Web of Documents. The Prolog Web, as we envision it, will share traits such as distribution, decentralisation and openness with the traditional Web. It will also share some traits with the Semantic Web, but at its core be based on Horn clause logic, a simpler logic than the description logic on which the Semantic Web is based. Compared to the Semantic Web, it will be less pure and have less ambitious goals when it comes to ontology-based semantics. However, in contrast to the Semantic Web languages, Web Prolog is not only a logic language, but also a fairly general-purpose programming language. The Prolog Web consists of a network of Web Prolog programs and processes, and is *programmable* in Web Prolog, but it is in a sense also a *part of* Web Prolog, since the latter contains primitives that allow any program being written in it to access the Prolog Web. One might think of the Prolog Web as the Web raised to a new level of abstraction, a level on which logic has become an integral part, but where programming still counts for something. Another way to put it is to say that we are *wrapping* the Web in Prolog.

Prolog is a mature programming language with unique qualities, and it would be a shame if these qualities never got a chance to play a role on the Web more important than it currently does. Unfortunately, while the paradigms of imperative, functional and object-oriented programming have a vigorous following, the paradigm of logic programming with its flagship Prolog has fallen far behind. Various people both inside and outside the Prolog community have at various occasions voiced their fears about the future of Prolog, noting that there are too many incompatible Prolog systems around, resulting in a fragmented community and an ISO standard that few systems conform to. As the title of this report tries to communicate, we see Web Prolog and the programmable Prolog Web as the two main ingredients in a recipe for creating a new standard, bringing the community back together, and making logic programming and Prolog really relevant again, a challenge that Prolog lovers around the globe hopefully would feel is worth responding to.

There is a third ingredient, not mentioned in the title of the report. We have mentioned it in other places, so it is not a secret. The third ingredient is the Prolog engine, or the *pengine* as we prefer to call it. We like to think of a pengine as a simple kind of *agent*, and of the Prolog Web as the *environment* in which such an agent is born, acts and die. The Prolog Web is populated with lots of pengines, and some of them are talking to each other using Web Prolog as their *agent communication language* (ACL).

During the writing of the report, a considerable amount of thinking has gone into the exposition of the “big picture” and how everything hangs together as a coherent whole. As a first sign of this, we spend a considerable effort on the positioning of Web Prolog, pengines and other actors, in the *really* big picture, which, of course, is the current Web *as such*, be it traditional or semantic, with its architectural design principles, languages, data formats and protocols.

Secondly, our interest in the big picture should also be evident not only from the way we carefully explore and make explicit the relations between some of the programming abstractions offered by the Web Prolog language, but also in the way in which we seek to connect two concepts – *knowledge representation* and *linguistic interaction*, and two application areas where these concepts materialise – the *semantic web* and the so called *conversational systems*. We furthermore relate Web Prolog to web standards in each area – to RDF and SCXML, respectively. [TODO: The relation to SCXML hasn’t really been dealt with yet.]

As a third sign of how big we want the picture to be, we also have something to say about ways to popularise the language, ways to teach it, possible standardisation, and so on. Towards the end of the report, the outcome of our ambitions in this respect will be presented in the condensed format of a *plan*, a partially ordered list of actions that we think that we need to take to reach our goal – to make Prolog stand out as a really relevant again.

1.1 Web Prolog in a nutshell

Starting from traditional Prolog, we are taking what might be called a *shrink-and-extend approach* to the design of Web Prolog. We shrink Prolog by removing primitives that do not seem to “belong” on the Web, and we extend it with carefully crafted primitives heavily inspired by Erlang, as well as with APIs useful for building web applications. We are aiming for a fairly small language, so we will try to extend by less than we shrink.

Web Prolog is a superset of a subset of ISO Prolog. It lacks certain built-in predicates that are present in the ISO standard, but also adds a number of predicates that are not in the standard, but are useful for spawning processes and sending and receiving messages in the style of Erlang. A Web Prolog runtime system – which we will refer to as a *node* –

is equipped with a comprehensive set of web APIs, using both WebSocket and HTTP as transport protocols, over which a client can run Web Prolog programs defined by the owner of the client, the owner of the node, or a combination of both.

Technology-wise this report can be seen as an attempt to rethink, redesign and refactoring our work done on *Pengines* (Lager and Wielemaker, 2014). We bring in a layer of predicates beneath the pengine abstraction in the form of a small set of programming primitives that support *actor programming*. Most of the semantics for the primitives, as well as their names, are borrowed from the Erlang programming language – the oldest and most successful language among those that usually count as actor programming languages. For a number of reasons, mixing in some Erlang-style concurrency with Prolog seems to make a good blend. We therefore take the plunge, and present Web Prolog as a programming language rather than a library for SWI-Prolog, which is the form Pengines currently lives in.

In order to motivate the design decisions made for the language and its environments we will carefully explore the relation between on the one hand Prolog and its execution model, on the other hand the Web and its protocols (HTTP and WebSocket in particular). We think, and will attempt to show, that taking advantage of this relation can be done without compromising the logical foundations of Prolog or introducing insurmountable performance bottlenecks.

We believe that a general feel for a language can be given by showing examples, so this is what we do, a lot. In an appendix the examples are complemented by a documentation of the most important Web Prolog built-in predicates. In between, we discuss various other aspects such as runtime and development environments and security.

We implement the core Web Prolog predicates as well as a minimal node in SWI-Prolog (Wielemaker et al., 2012), effectively using SWI-Prolog as an informal but executable specification language. Efficiency of implementations is not our main concern in this report, but we do suggest a couple of avenues along which efficient implementations may be sought. Caching opportunities, for example, seem to abound.

1.2 Language and environment

> A programming system has two parts. The programming "environment" is the part that's installed on the computer. The programming "language" is the part that's installed in the programmer's head. – *Bret Victor*

Web Prolog can be described as a domain-specific language that both simplifies and specialises (shrinks and extends) the Prolog programming language. Simplifications involve the removal of many I/O operations, foreign-language interfaces, and access to the operative system. Specialisation involves the addition of Erlang-ish primitives which makes our features for web programming possible. Yet, as we wrote above, most Prolog text book examples will run without modifications.

A *node* is an executing Web Prolog runtime environment that has been given a unique name in the form of a URI. A node may also act as a client to another node, and for this reason it makes sense to regard it as a node in a peer-to-peer system rather than as a server in a client-server system. A node thus has a dual identity: as a Web Prolog runtime system, and as a node in the network forming the Prolog Web.

We define several different profiles of Web Prolog (four at the moment), with different language capabilities and different capabilities for communication and other forms of interaction. Some nodes, depending on their profile, can only be queried, others allow a client to hand over a Web Prolog program to be run by the node on behalf of the client. The most advanced profile – which we refer to the ACTOR profile – allows concurrent programming as well as unrestricted two-way communication between a client and a node.

Most programming language standards are aiming at *portability* and the ISO Prolog standard is no exception. For Web Prolog portability is indeed important, but we are aiming also for *interoperability* – the ability of different information technology systems and software applications to communicate, exchange data, and use the information that has been exchanged. Indeed, the level of interoperability that we seek for advanced profiles of Web Prolog does in fact *include* portability. As a concept, interoperability is clearly related to communication and other forms of interaction. Interoperability needs precise and non-ambiguous communication and this is what primitives for sending and receiving messages are for. The predicates that we add to our ISO Prolog subset are more or less all concerned with communication in the service of interoperability. In the context of the Web, interoperability is realised through web services, web APIs, and standards-based protocols such as HTTP and WebSocket. A lot of the effort behind Web Prolog is aiming at putting protocols such as these to work.

Another important information technology concept, clearly related to interoperability, is *integration*, i.e. the process of linking together different computing systems and software applications functionally, in order to make them act as a coordinated whole. Indeed, one of our more ambitious goals is to integrate (at least some of) the many different Prolog systems in existence (and there are many of those), using Web Prolog as an *interlingua* (or, using a related and more popular term, as a *glue language*).

Web Prolog nodes are meant to be implemented on top of other programming platforms. In this report, we sketch an implementation on top of SWI-Prolog, but we also hint at the possibility of building Web Prolog nodes on top of other Prolog systems, as well as on top of systems implementing other programming languages. We shall even hint at the possibility of implementing Web Prolog in JavaScript and thus allow applications running in web browsers to be built using only Web Prolog.

In order to make the installation of the Web Prolog language "in the programmer's head" as painless as possible, we take as our point of departure a subset of the ISO Prolog core predicates. The Prolog dialect defined by ISO is well understood, is reasonably close to most of the dialects used in Prolog text books, and it is a standard, with all that this entails.

Concepts such as files, streams and sockets are too low-level for the Web, and many ISO predicates are therefore simply not appropriate for this domain. Thus we see, in our vision of a viable web logic programming language based on Prolog, no reason to support predicates for file I/O or OS access. This is a consequence of our ambition to allow people (or processes) to create processes on someone else's machine, asking queries and running code that the owner of the machine is not necessarily expecting or knows about. To allow this, the owner must be ensured that safety of the host machine cannot be compromised.

What about other components of the ISO Prolog standard such as Threads and Modules? They simply are not needed. The role of threads will been taken over by actors. That this will work should not be controversial. Since actors also provide isolation, they may be able to take over some of the functionality of modules as well. Other features that could help making modules more or less superfluous comes from the way URIs lend structure to resources on the Web. [Here, Piancastelli and Omicini (2008) can probably provide us with inspiration.]

1.3 Pengines (and other actors) in a nutshell

> In a nutshell, if you were an actor in Erlang's world, you would be a lonely person, sitting in a dark room with no window, waiting by your mailbox to get a message. Once you get a message, you react to it in a specific way: you pay the bills when receiving them, you respond to Birthday cards with a "Thank you" letter and you ignore the letters you can't understand. – *Fred Hebert*

An actor, in the abstract, is the fundamental computational unit in the actor-based model of computation. However, just saying of something that it is an actor does not tell us what it does, what messages it is capable of understanding (and in which order they must arrive), or what messages it is capable of sending (and in which order we can expect them to be sent). In other words, although we can think of an actor as an interaction engine, just saying of something that it is an actor does not necessarily say anything about the specifics of the communication protocol that it acts in accordance with when talking to the world outside it.

Seen from the outside, an actor is a black box – what goes on inside is private to it. In order to do "its thing" it may be talking to other actors, it may even be capable of creating other actors and then act as their supervisors. As long as we know the specifics of the protocol that it speaks, we do not need to know what is going on inside the actor, how it stores data, or if it creates and/or talks to other actors.

The actors being talked to, or first being created and then talked to, do not have to be local, they can be remote. The actor model thus provides us with one single abstraction as a solution to the two problems of concurrency and distribution; it allows applications to scale out not only to multiple cores on one machine but also to multiple nodes running on multiple machines on a network such as the internet.

But can we really have the cake and eat it too? Is the actor programming model compatible with the logic programming model, and does the combination of the two models make sense? Yes, we will try to show that the actor programming model of Web Prolog is able to co-exist harmoniously with its logic programming model, and that both models are usable in the same program, with actors used for modelling the reactive, deterministic interfaces to the user and other external entities, and with the non-deterministic component behaving as an actor, encapsulating the non-determinism. The key abstraction here is the *pengine*.

A pengine is a special *kind* of actor characterised by a particular communication protocol – the Pengine Communication Protocol (PCP). We shall return to the details of the PCP further down in the text, but for now it suffices to say that this protocol is basically what makes a pengine into a first-class Prolog top-level. We can send it queries, and be sent answers in return, answers that follow logically from the "beliefs" that the pengine holds. In this sense, a pengine is an interactive reasoning engine. Similar to a normal Prolog top-level, a pengine is "lazy" in the sense that it will not produce all the answers to a query at once, unless we explicitly ask it to do so. Typically, we send it a query and get one answer back. Then we send it a request for another answer to the same query, and if there is one, we get it too. At any point in the conversation, we may choose to stop asking for more answers to the first query, and to send it a different query instead, and so on. We can think of this as a *ask-next-stop* kind of protocol.

As we shall see, a pengine supports another and very different protocol as well, based on integer offsets into a virtual index of solutions to queries rather than relying on the *ask-next-stop* protocol. Using this protocol, and given a query, we are always able to ask for the solution at offset *i*, regardless of the value of *i*. Using this protocol, we can choose to query a node *directly*, rather than to query a particular pengine running on the node. As it nicely captures the notion of so called *RESTfulness*, this has certain implications for the scalability of the Prolog Web.

Briefly summarising this section, an actor can be seen as the fundamental computational unit in a *combined* logic-based and actor-based model of computation. Since it implements SLD resolution we can think of it as a reasoning engine, and by virtue of its ability to send and receive messages we can think of it as an interaction engine. An actor adheres to the laws of logic and to the norms of its communication protocol – it is both a reasoning engine and an interaction engine.

1.4 Web Prolog is inspired by Erlang

Erlang is a general-purpose, concurrent, functional programming language developed by Joe Armstrong, Robert Virding and Mike Williams in 1986. It started out as a proprietary language within Ericsson, but was released as open source in 1998. A good overview of the Erlang language, its design intent, and the underlying philosophy, is given in (Armstrong, 2003b).

Erlang was designed with the aim of improving the development of telephony applications but has in recent years, thanks to its built-in support for massive concurrency, distribution and fault tolerance, been used as a very capable language for implementing the server-sides of web applications.[1]

Erlang processes are implemented by the virtual machine, not by operating system threads. They are very lightweight, and completely isolated from each other. Creating and destroying processes is very fast, and millions of processes can be created on one machine. Sending messages between processes is also very fast. Therefore, in the context of the Web, Erlang provides the advantage that you can have one server process per web client, or (say) ten of them if you want. We hasten to add that while we do not aim for the kind of *massive* concurrency that Erlang is known for, we certainly do our best not to introduce features in the Web Prolog language that would preclude this.

The design of Web Prolog takes a lot of inspiration from Erlang. However, from the outside its influence is only really visible in the ACTOR profile of Web Prolog, where primitives for process creation and messaging makes Erlang-style concurrent programming possible. As we shall see, an Erlang-style actor-based architecture for Web Prolog and the Prolog Web also creates something of a semantic foundation for all profiles. On top of actors, we shall build pengines, and using pengines we shall implement web APIs as well as a simple yet powerful predicate for making non-deterministic remote procedure calls.

Why did we choose Erlang as a source of inspiration, rather than any alternative? After

[1]By companies such as WhatsApp and Klarna for example.

all, there are other actor programming languages – Akka, Rust, E and Ambivalent, to name just a few, and Elixir to name the latest and hottest. One reason is that historically, Erlang evolved from Prolog and the first versions of Erlang were actually implemented in Prolog (Armstrong et al., 1995). This still shows in that Erlang and Prolog are in many ways similar. Most data types map cleanly between the two languages. Atoms are the same in Erlang and Prolog, and the same goes for integers and floating point numbers. Lists look and behave the same in Prolog and Erlang, and an Erlang tuple {`foo,a,1`} corresponds most naturally to a Prolog term `foo(a,1)`.[2] Furthermore, Prolog variables are assign-once variables, just like Erlang variables, pattern matching is the dominant way to pass values, just like in Erlang, and just like in Erlang, loops are normally implemented by means of recursion.

The most important difference between the two languages is of course the fact that Erlang is a functional language, whereas Prolog is relational. Since a function is just a special case of a relation this difference must not be exaggerated, but it does show in the way function calls may be nested, something that cannot be done in Prolog since arguments are used to represent outputs as well as inputs.[3].

The following listings of Prolog code (to the left) and Erlang code (to the right) show some of the similarities as well as some of the differences between the two languages:

```
% length                                  % length

length([], 0).                            length([]) -> 0;
length([_|T], N) :-                       length([_|T]) ->
    length(T, N1),                            1 + length(T).
    N is N1 + 1.

% naive reverse                           % naive reverse

reverse([], []).                          reverse([]) -> [];
reverse([H|T], R) :-                      reverse([H|T]) ->
    reverse(T, RevT),                         reverse(T) ++ [H].
    append(RevT, [H], R).
```

It is sometimes lamented that the inventors of Erlang should have chosen a more "traditional" syntax – it is quite unlike what most programmers are used to and is a hurdle to cross. Prolog programmers do of course beg to differ – we are fine with the syntax. What we do not like has more to do with semantics and the sacrifice of a lot of good things that Prolog has to offer. It can hardly be denied that some very useful features were lost somewhere on the road from Prolog to Erlang.[4] Here is an attempt to list those features that are available in Prolog (and in Web Prolog) but not in Erlang:

[2]However, note that {`foo,a,1`} is a term in Prolog too, of the form `{}((foo,(a,1)))`

[3]One can of course build a preprocessor than translates functional notation into the relational syntax of Web Prolog.

[4]There is even a talk by Joe Armstrong on YouTube where he seems to be regretting removing too much Prolog from Erlang. See `https://www.youtube.com/watch?v=h8nmzPh5Npg#t=40m17s`

- Built-in backtracking search
- Unification
- Logic-based knowledge representation
- Reasoning
- Meta-programming
- User defined operators
- A term expansion mechanism
- Definite Clause Grammar (DCG)

Those are some powerful features that were removed from Prolog in order to arrive at Erlang. In no way should this be seen to imply that we think that the inventors of Erlang made a *mistake* when they scrapped Prolog in favour of a simple functional language (a.k.a. Sequential Erlang), *despite* the fact that the first versions of Erlang, as we mentioned above, were actually implemented in Prolog. Given the circumstances at the time, given the nature of the problems with programming telephone switches that they set out to solve, and perhaps in an attempt to avoid being dragged down by the post fifth generation dismissal of logic programming, they probably made exactly the right choice. (Another factor that probably played a role is that Prolog is more difficult language to learn and to use correctly.) After all, Erlang is a very successful programming language, far more successful than Prolog, and this is the proof of the pudding.

For Web Prolog, the similarity with Erlang goes deeper than syntax or data types. Just like Erlang but in contrast to ISO Prolog, Web Prolog is an *actor-based language*. In Web Prolog, spawning an actor computing the length of a list may be done in the following way:

```
?- self(_Self),
   spawn((length([a,b,c],N), _Self ! N)),
   receive({_M -> io:write(_M)}).
3
true.
?-
```

In Erlang, it would look like so:

```
1> Self = self(),
   spawn(fun() -> Self ! length([a,b,c]) end),
   receive M -> io:write(M) end.
3ok
2>
```

A little bit of unnesting makes the similarities stand out more:

```
2> Self = self(),
   spawn(fun() -> N = length([a,b,c]), Self ! N end),
   receive M -> io:write(M) end.
3ok
3>
```

Apart from minor syntactic differences, only two differences strike us as a significant. First, (and again) Erlang is a functional language which allows nesting and Prolog is a relational language where nesting is in general not possible. Secondly, Erlang is a higher-order language. This can be seen in the way the spawn function takes an anonymous function as its argument. Prolog is not a higher order language in that sense. In Web Prolog `spawn/1` expects a callable goal to be passed. In the Prolog world, such predicates are often referred to as meta predicates.

Another thing that Web Prolog has in common with Erlang is that spawning and sending work also in a distributed setting. In Web Prolog, just like in Erlang, we can use the spawn operation to invoke a process on a remote node and subsequently communicate with it using send and receive. As we shall see, the primitives for spawning and messaging in Web Prolog are in some ways more expressive than the corresponding Erlang primitives.

The Erlang distribution mechanism is implemented using raw TCP/IP sockets. One could imagine implementing a Prolog distribution mechanism using TCP/IP sockets too. It has been done – in Qu-Prolog for example – but this is not the way we do it for Web Prolog. Instead we use WebSockets. This makes our job a lot easier, since, in contrast to TCP/IP, WebSocket is a message oriented protocol, more abstract than TCP/IP, which is used to transfer sequences of bytes rather than messages. And not only does the use of WebSockets save us from having to implement our own message based protocol on top of TCP/IP, it also allows communication to pass through firewalls, and implements various security-related features such as methods for authentication, Secure Websockets (i.e. WebSockets over HTTPS) and CORS (Cross-Origin Request Sharing).[5]

Although one could possibly argue that from a theoretical perspective, Erlang is not the best model for programming with concurrency, it is certainly more high-level and easier to use than the POSIX-based model underlying the ISO Prolog Threads draft standard. Besides, from a pragmatic point of view there is a lot that speaks for it: many programmers, many courses, many text books, and other languages (such as Elixir[6]) that have been based on exactly the same model.

Erlang's contribution here concerns not only reactiveness but also the orchestration/-choreography of concurrent and possibly distributed processes using message passing. As suggested by Armstrong (2003b), it allows us to organise our system as a set of communicating processes. By enumerating all the processes in our system, and defining the message passing channels between them, we can partition the system into a number of well-defined components which can be independently implemented and tested.

When describing Web Prolog, we adopt Erlang terminology as much as we can. It means that we rather write about messages than about events, about mailboxes instead of event queues, about nodes rather than servers, and about processes or actors rather than threads. We also name some of our predicates – the ones that are added to support spawning and messaging – after Erlang primitives with similar functionalities.

To summarise the findings of this section, there are mainly two reasons for why Erlang is interesting for our cause. First, that it seems to adapt very well to the future of hardware, and secondly, that it has its roots in Prolog. The first reason makes it attractive for everyone who believes that the multi-core hardware revolution means something for the future of computing, while the second reason concerns only those who believes that Prolog, and logic programming more generally, has a role to play in this future.

[5] `https://en.wikipedia.org/wiki/Cross-origin_resource_sharing`
[6] `https://elixir-lang.org`

1.5 Web Prolog is inspired by JavaScript

> Scripting languages are a lot like obscenity. I can't define it, but I'll know it when I see it. – *Larry Wall*

Over the years, the Web has become a key delivery platform for a variety of sophisticated interactive applications in just about any conceivable domain. As a consequence, JavaScript, more or less the only game in town for programming the front-end of a web application, has become the most commonly used programming language on Earth. Even back-end developers are more likely to use it than any other language.[7] Indeed, JavaScript must be regarded as *the* web programming language of our times.

Comparing Web Prolog with JavaScript may seem far-fetched since, as languages and in terms of programming paradigms, they are as different as two programming languages can be. They are both dynamically typed multi-paradigm languages and... yes, that is about it! JavaScript supports object-oriented, imperative and functional programming styles and does not support concurrency, whereas (full) Web Prolog is a concurrency-oriented relational/logic programming language. Yet, as web programming languages they share a common domain-specific purpose.

We will have to look to the runtime environments and the role they play on the Web in order to spot the similarities between the languages. JavaScript is a sandboxed scripting language, most commonly found embedded on the client-side of the web (e.g. in web browsers), but which in recent years has found a role on the server-side too (e.g. in node.js). We argue that Web Prolog too should be construed as an embedded scripting language – first and foremost on the server-side, embedded in programming systems that may or may not be Prolog-based, but eventually also in web browsers as an optional (at least partial) replacement for JavaScript. This explains why Web Prolog, just like JavaScript, must not include any file I/O, networking or storage, but must rely upon the host environment in which it is embedded for such functionality.

So, Web Prolog does indeed share some of the properties that made JavaScript succeed on the Web. A visit to a web application starts a JavaScript process on the client, running code that has been downloaded from the application's host to the user's client. Such a process must be allowed to run there (i.e. the owner of the client must allow it), and when it is, it must execute in a way that does not harm the client. A Web Prolog process is usually run on the node rather than on the client. It must be allowed by the owner of the node to do so, and it must run without harming the node. So one thing that they share, Web Prolog and JavaScript, is the ability to run untested and untrusted source code, authored by unverified and untrusted programmers, in a sandbox on someone else's machine.

1.6 Web Prolog everywhere?

Once solely a client-side language, JavaScript is now welcomed on the server-side too. Not having to learn more than one language in order to become a so called "full-stack developer", and not having to switch languages when changing from working on the server-side

[7] `http://stackoverflow.com/research/developer-survey-2016#most-popular-technologies-per-occupation`

to working on the client-side are just two of the advantages. Although the present proposal is only concerned with server-side Web Prolog, in the future we would like to be able to offer developers client-side Web Prolog as well, and potentially allow them to run the same code on both the client and the server.

On the one hand, we want to keep most of our Web Prolog source code on servers, in particular if the code represents data supposed to be shared among many clients. On the other hand, with our data on servers, there is no way we can avoid making network round-trips. Some applications, especially on mobile devices, cannot afford the latency involved.

The ideal would be if a programmer had a choice here, but for this to happen, Web Prolog needs to be supported on the client-side too.

At least one leading engineer at Google thinks we need more web programming languages.[8] With languages such as Elm and ClojureScript, functional programming is moving in this direction. When it comes to web logic programming languages, not much is going on. We hope and believe that Web Prolog could eventually become such a language. However, to really count as a full-blown web programming language, we suspect that Web Prolog must be made available on the client side too.

[TODO: This section needs more work.]

1.7 The Prolog Web in a nutshell

The Web started out as a distributed hypertext system, but already in its current form, thanks to inventions such as web services and APIs, the Web has also become a simple and flexible environment for distributed programming. At one point in the history of its evolution, the Hypertext Web morphed into a *Programmable Web*. With the help of state-of-the-art web technologies, the Prolog Web tries to take this a step further. The programmable Prolog Web supports two different distributed programming models, one based on actors and asynchronous communication over WebSockets, and one based on the synchronous communication provided by HTTP. With the help of these technologies, we argue that the Prolog Web can be made *programmable* in an unusually strong sense of this word.

The traditional Web is distributed, decentralised and open. These are traits that we want the Prolog Web to also possess. Whereas distribution is nicely conceptualised in the actor model, and nicely handled by an actor programming language such as Erlang, decentralisation and openness require features that we must rely on the Web as such to contribute. Erlang and most other platforms for distributed programming are not open in the sense that we require. They usually rely on TCP for transport and are, for reasons of security, assumed to be operating in a closed, trusted environment where we directly control the machines that we are operating.[9] In other words, when Erlang runs on a *cluster*, it is a cluster that is *closed*. We can think of the Prolog Web as a cluster as well, but one that is *open*.

Naturally, creating a Prolog Web entails adapting Prolog to the current Web rather than the other way around. Not only does this involve designing a special-purpose programming

[8]`https://www.pcworld.com/article/2362500/google-engineer-we-need-more-web-programming-languages.html`

[9]This is also the case for the Qu-Prolog system developed by Clark et al. (2001) – one of the few distributed logic programming systems that we are aware of.

language such as Web Prolog, but it also means that when designing the architecture of the Prolog Web, we must do our best to align with the principles guiding the development of the current Web, including the conceptual model of the relationship between URIs, resources and representations, the use of transport protocols and APIs, and architectural styles such as REpresentational State Transfer (REST).

When it comes to the Web's conceptual model of the relationship between URIs, resources and their representations we are dealing with at least three kinds of resources: Web Prolog programs, actors (pengines in particular) and answers to Web Prolog queries. Such resources should be regarded as fairly abstract entities. The corresponding concrete representation for a program is a valid Web Prolog program text. The concrete representation for an actor is a pid identifying the mailbox of the actor. Answers to queries have two kinds of representations: the Prolog representation and the JSON representation. Addressability, i.e. providing URIs to as many resources as possible within a system, is an hallmark of an open Web.

Web APIs play a central role on the Prolog Web. Not only do they allow a non-Prolog client – typically implemented in JavaScript and built to be run in a web browser – to create and communicate with a pengine running on a remote node, but they also allow this pengine to talk to *other* pengines on the Prolog Web, thus forming a multi-pengine system capable of utilising local as well as remote resources for satisfying the information and/or control needs of the user running the client. The only reliable way to interact with a node is to use the web APIs that it provides.

We focus on two transport protocols in particular: WebSocket and HTTP. We did not have much choice here, since if we do not use HTTP or WebSocket, we simply are not on the Web. Having been developed to serve communication on the Web, these protocols are characterised by their ability to traverse firewalls and to play nicely with proxies. Using these protocols, rather than developing our own is, besides the addressability of resources provided by URIs, another way to ensure the openness of the Prolog Web.

In this report we shall distinguish two kinds of APIs supporting the communication between a client and a Prolog Web node: an asynchronous and stateful WebSocket API, and a synchronous and stateless HTTP API. Some nodes will offer both, while other nodes may only offer the HTTP API.

WebSocket is a simple, bi-directional and stateful protocol for asynchronous communication between a client and a server. A connection must be initiated with an HTTP request, but after this has been performed, messages carry very little overhead. Bi-directionality here means that two-way communication is possible. A server can notify a client of anything at any time, so that when something of interest to a client occurs, the server simply sends a message to the client. This is often referred to as *server push*.

Statefulness allows the build-up of a server-side context which makes earlier interactions matter to how later ones are handled. The server needs to maintain one such context per client session and therefore must be able to distinguish one client from another. As we shall see, the pid acts as an identifier that allows a node on the Prolog Web to do just that.

Bi-directionality and statefulness are properties that make WebSocket by far the most flexible transport protocol for the Prolog Web. Over a WebSocket connection, a client can be in almost total control of an actor such as a pengine. (We write "almost" here since the owner of the node on which the actor is running always has the ultimate say when it comes to how much resources will be allocated to the running of a actor.)

During the last couple of years the WebSocket protocol has matured and is now almost universally supported. All major web browsers support the `WebSocket` object which offers methods and event handlers for setting up a WebSocket connection, and many programming languages, SWI-Prolog included, offer libraries for building both clients and servers using WebSockets.

HTTP is a simple, synchronous and stateless request-response protocol, where the request must be made by the client and the response be given by the server. Most programming languages provide developers with an API for making use of this protocol, either as built-in functions or methods, or as libraries. Here again, SWI-Prolog offers everything a developer could ever wish for. In JavaScript, developers have access to the `XMLHttpRequest` object,[10] allowing both synchronous and asynchronous HTTP requests to be made.[11]

Since HTTP is in essence a protocol for one-way communication only, the HTTP API is more restricted than the WebSocket API as the pengine is not allowed to send any kind of message messages back to the client at any time. For the same reason, communication with actors in general cannot be supported either.

When using the stateless API there is no need for a client to make an explicit request for the creation of a pengine on a node. Omitting any mention of the name of a particular pengine in a GET request to the URI `/ask`, the node can still be queried. Here though, an integer offset must be sent along, pointing to a particular solution to the query. Despite not being mentioned by name, readers may suspect that a pengine is involved here too, and this may well be the case. In fact, when running a query to completion, *more* than one pengine may be involved. We refer to them as a *anonymous* pengines. The important thing here is that the request can be made *as if* no pengines were involved, and *as if* the request was made directly to the node. The stateless API is as close to a so called RESTful API that we will get in this report, and the RESTfulness and the scalability that RESTfulness offers is what makes it useful, despite certain disadvantages to be revealed and discussed further down in the text. All Prolog Web nodes provide this API, and for some nodes it is the only API on offer.

1.8 A brief history of Logic Programming on the Web

We do not want to give the impression that the suitability of Prolog for the Web has not been investigated before. On the contrary, in the mid 90-ties the advent of the Web sparked a great deal of interest in the logic programming community, reflected in two workshops devoted to the subject and in the development of libraries that mostly relied on creating so called CGI programs running Prolog. Here is a snippet of text lifted from the call for papers for the second workshop:[12]

> This workshop is the second in a series intended to explore the elective affinities between Logic Programming and Internet technologies with emphasis on en-

[10] `https://en.wikipedia.org/wiki/XMLHttpRequest`

[11] HTTP is a synchronous protocol, but through the use of JavaScript callbacks the `XMLHttpRequest` object can make it look asynchronous from the point of view of a JavaScript programmer.

[12] `http://www.cliplab.org/lpnet/proceedings97/preface.html`

> hancing the World Wide Web with knowledge, deductive abilities and superior forms of interactive behavior. With the paradigm shift to highly inter-connected computers and programming tools, logic programming languages have a unique opportunity to contribute to practical Internet application development. Simplicity, remote executability, robustness, automatic memory management, are among the features some LP languages share with emerging tools like Java. Superior meta-programming and high-level distributed programming facilities, built-in grammars and dynamic databases, declarative semantics are among their competitive advantages.

Since then the Web has developed with an amazingly fast pace, and web applications have become increasingly speedier and more interactive. The Web of today is *very* different from what it was twenty years ago – faster, more reliable, more mobile, and with a fantastic selection of browser APIs that lets a web developer add features such as spoken interaction, video or 3D graphics to an application. There is a reason that we nowadays talk about, not only the Semantic Web, but also the Conversational Web, the Mobile Web, and the Web of Things.[13]

Furthermore, the WebSocket standard did not exist twenty years ago, which meant that every approach to web logic programming had to work solely with HTTP. Important aspects of some of the inventions described in this book relies on the WebSocket protocol, a so called server-push technology which opens a whole slew of opportunities for developers of web applications based on Prolog.

With the Web (technically) so much better and so different from what it used to be, there is every reason to make a new attempt at "enhancing the World Wide Web with knowledge, deductive abilities and superior forms of interactive behavior".

We do not suggest than nothing has happended between now and then. [TODO: Here we ought to mention at least: Loke and Davison (2001), Alferes et al. (2003), Loke (2006) ..., Piancastelli and Omicini (2008) who (probably) coined the term "Web logic programming" and presented an approach in line with REST and ROA, and our own work (Lager and Wielemaker, 2014).]

Among Prolog systems, SWI-Prolog (Wielemaker et al., 2012) in particular has for a long time offered stable and mature libraries for working with protocols such as HTTP and WebSocket, and with web formats such as HTML, XML, JSON and RDF. Although SWI-Prolog is not *entirely* devoted to the Web, it certainly comes closer to being just that than any other Prolog system that we are aware of.

If we broaden our outlook somewhat and include work on distributed logic programming which is not necessarily on the Web we ought to mention April and Qu-Prolog.

The notion of explicit Prolog engines has been explored extensively by Tarau and Majumdar (2009) and Tarau (2011) in the context of the Jinni agent programming language and the BinProlog system. Tarau's engines are in-process and primarily designed to provide a clean alternative implementation for Prolog language constructs such as the all-solutions predicates (e.g., findall/3), exception handling as well as language constructs that are less common in the Prolog world, such as multiple coroutining blocks.

[TODO: This section needs more work.]

[13] See (Lager and Myrendal, 2012) for a description of the many facets of the current Web.

1.9 Running the proof-of-concept implementation

Vision without execution is just hallucination – *Thomas Alva Edison*

This report is accompanied by an proof-of-concept online demonstration of Web Prolog, available at the following location:

`http://...` [Sorry, but the online demonstration is not yet available.]

The demonstrator features a tutorial that takes visitors on a tour of the language and allows them to run most of the programs appearing in this report. It is also equipped with a decent editor which together with a traditional Prolog shell allow them to engage in the usual interactive edit-run cycle and compose their own programs.[14]

Not much is novel about Web Prolog and its development environment, except perhaps for the way in which Prolog-related and Erlang-related concepts are *combined* in order to support Prolog-style logic programming as well as Erlang-style concurrent and distributive programming. We predict that for people familiar with both Prolog and Erlang, not much about the language will be difficult to grasp. We also hope that for those people, the tutorial will serve to substantiate our claim that Web Prolog is a natural, readable and potentially very useful web programming language.

For those who wish to run their own installation of Web Prolog the implementation of the demonstrator is available here:

`https://github.com/Web-Prolog/swi-web-prolog`

The tutorial is included in the download.

[14]In the future, we plan to integrate Web Prolog with SWISH (see `https://swish.swi-prolog.org`), a much more mature online IDE for Prolog than the one offered by the proof-of-concept implementation. The traditional Prolog shell might be one of several tools on offer, selectable from a tool bar along with the present one. The tutorial will likely be rewritten as a Prolog notebook.

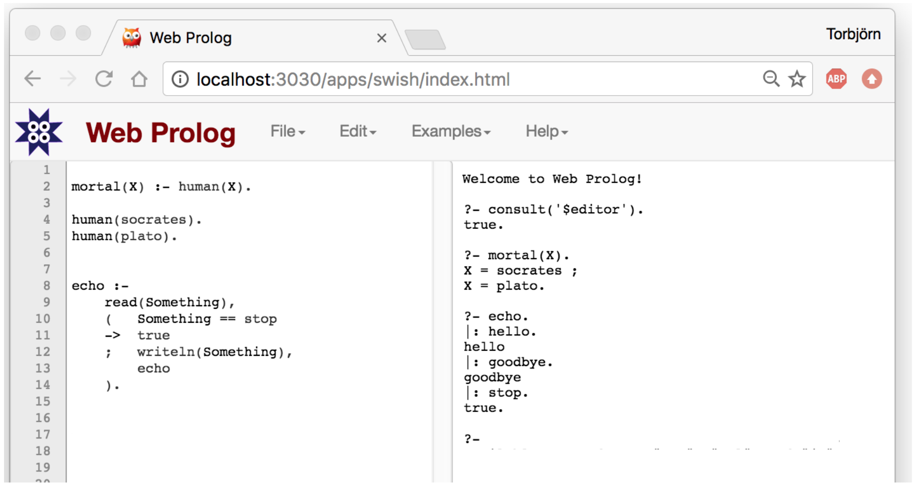
Chapter 2

SWISH, nodes and pengines

In this chapter we introduce SWISH, a web-based integrated development environment for Web Prolog. SWISH is implemented in SWI-Prolog (on the server-side) and HTML, CSS and JavaScript (on the client-side) and runs in a web browser.

2.1 A first encounter with SWISH

SWISH offers many useful tools but in this section we shall only deal with two of them, the *editor* and the *shell*. Figure 2.1 shows them as they appear in a browser window, with the editor to the left and the shell to the right.



```
mortal(X) :- human(X).

human(socrates).
human(plato).


echo :-
    read(Something),
    (   Something == stop
    ->  true
    ;   writeln(Something),
        echo
    ).
```

```
Welcome to Web Prolog!

?- consult('$editor').
true.

?- mortal(X).
X = socrates ;
X = plato.

?- echo.
|: hello.
hello
|: goodbye.
goodbye
|: stop.
true.

?-
```

Figure 2.1: The SWISH integrated development environment. [TODO: Replace image]

A proper editor is the most important component of a usable programming environment and we are confident that the SWISH editor is not going to be disappointing in this regard. The shell is quite an ordinary Prolog shell.

Here we give an example of how to use SWISH. Suppose that we (inspired by events that are said to take place around the times when logic was born) have written the following program in the editor:

```
mortal(X) :- human(X).

human(socrates).
human(plato).
```

Turning then to the shell, we consult the content of the editor, enter a query and inspect the results produced one-at-a-time in the usual lazy fashion:

```
?- consult('#editor').
true.
?- mortal(X).
X = socrates ;
X = plato.
?-
```

Here is another little program, demonstrating reading and writing:

```
echo :-
    read(Something),
    (   Something == stop
    ->  true
    ;   writeln(Something),
        echo
    ).
```

Here is how we test this program in the shell:

```
?- echo.
|: hello.
hello
|: goodbye.
goodbye
|: stop.
true.
?-
```

So despite being a web-based tool, the appearance and functionality of the SWISH shell is really very similar to traditional Prolog shells. This is intended as it will make our task of introducing the syntax and semantics of Web Prolog easier. Throughout this report, the great majority of Web Prolog example programs and queries will be presented in this old fashion editor-and-shell user interface. The idea, of course, is simply to make Prolog programmers feel right at home.

However, whereas a traditional Prolog shell is normally (always?) written in Prolog, and runs in the same process as the queries that are passed to it, the SWISH shell and the Prolog top-level to which it is attached are written in different programming languages, runs in different processes, and often on different machines. Indeed, we like to think of the "conversation" between a programmer and a Prolog top-level as a form of *mediated communication*, and of the shell as a *mediator* between the two of them.

2.2 A closer look at shell-pengine interaction

In this section, we will go through the examples in the previous section again, but now also briefly indicate what is taking place under the hood. In addition to the programmer and the SWISH application running in the programmer's web browser, this involves a Web Prolog *node*, identified (in this case) by the URI `http://node-1.org`. Figure 2.2 depicts this scenario.

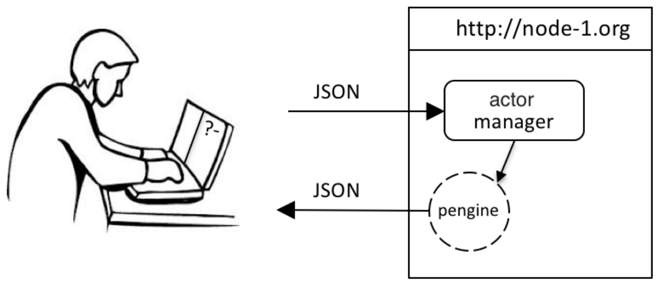


Figure 2.2: The shell offers mediated communication between a programmer and a pengine.

The purpose of a Prolog Web node is to host *pengines* and other actors. As we have already explained, a pengine is an actor and a programming abstraction modelled on the interactive top-level of Prolog. In other words, a pengine is like a first-class interactive Prolog top-level, accessible from Prolog as well as from other programming languages. Alternatively, it can be seen as an encapsulated Prolog session, or as a compute server for Prolog programming tasks.

Despite what Figure 2.2 may suggest, the programmer need not be alone in interacting with this particular node. Other programmers may be talking to other pengines running on the same node. The pengines are completely shielded from each other, unless of course the programs they are running have been written to allow them to communicate. In any case, pengines do not share memory, so in order to share information, they must exchange messages.

The task of the *actor manager* is to handle the reception of messages sent by clients and arriving over HTTP and WebSocket connections. They may be messages requesting the spawning of a pengine or termination of a pengine, or messages to be forwarded to the pengine addressed by a pid. The actor manager is also responsible for the registration and deregistration of pengines. The registration happens right after the creation of a pengine, deregistration after it has terminated.

Crucially, a node may also host a Web Prolog *program* – a database, an expert system, a game engine, a digital assistent, a home control system, or another kind of program. If it does, any pengine running on the node has access to the predicates defined by this program. The program is typically maintained by the owner of the node. Unless we are authorised to do so, we are not able to make any changes to it, only the owner is allowed to do so. However, by means of code injection in the workspace of the pengine, we are allowed to *complement* the hosted program with source code that we have written ourselves. Another way to put it is to say that we are *programming* the pengine, which in turn leads us to our claim that the Prolog Web is *programmable* in a sense stronger than the usual.

The interaction between the shell and the node with its pengines uses the *Pengine Communication Protocol* (PCP), implemented as a WebSocket communication sub-protocol. Other clients may use HTTP instead of websockets, but this comes at the price of only being able to use a more constrained version of the PCP protocol.

2.3 Under the hood

When the programmer enters the URI of the node in the browser's address field, a web socket connection is first established between SWISH and the node, and then used to ask the node to spawn a pengine.

```
connection.send(JSON.stringify({
  command: 'pengine_spawn',
  options: '[format(json)]'
}));
```

Messages sent to a node are strings, couched in the syntax of JSON, whereas messages arriving back from a node are expressed in either Prolog or in the JSON format. In this case, the pengine was instructed to use the JSON format. Therefore, the following message is received in response:

```
{ "type":"spawned",
  "pid":"9a343810-dbe8-11e6-bd74-bfe12f8b98b6"
}
```

SWISH creates the shell, presents some information about the node and the pengine, and prompts the programmer for input:

```
Welcome to SWI Web Prolog
?-
```

At this point in the interaction the workspace of the remote pengine is empty. In order to inject the content of the editor into it, the programmer calls the goal `?-consult('#editor')`, which under the hood results in a `pengine_ask` message being sent to the node:[1]

```
connection.send(JSON.stringify({
  command: 'pengine_ask',
  pid: pid,
  query: 'consult(" + <String> + ")'
}));
```

The node injects the editor content into the workspace of the pengine identified by the pid, and the following response indicating success is returned back to SWISH:

```
{ "type":"success",
  "pid":"9a343810-dbe8-11e6-bd74-bfe12f8b98b6",
  "data":[],
```

[1]The idea here is that the shell is responsible for replacing the reserved word '`#editor`' with the content of the

```
  "more":false
}
```

The shell acknowledges that the code is free from errors and has been loaded into the workspace of the pengine by showing:

```
?- consult('#editor').
true.
?-
```

The pengine can now be queried. When the programmer enters the query `?-mortal(X)`, the shell process sends a message of the following form to the node, which again forwards it to the pengine:

```
connection.send(JSON.stringify({
  command: 'pengine_ask',
  pid: pid,
  query: 'mortal(X))'
}));
```

SWISH will receive the following response in return:

```
{ "type":"success",
  "pid":"9a343810-dbe8-11e6-bd74-bfe12f8b98b6",
  "data":[{"X":"socrates"}],
  "more":true
}
```

As a consequence, the shell is updated to show the following:

```
?- mortal(X).
X = socrates |
```

Since the value `true` of the `more` property of the JSON message indicates that other solutions to the query may exist, a blinking cursor prompts the programmer for input, asking as it were: "You want more, or not?". When the programmer responds by typing a semicolon, the shell sends a `next` message to the pengine by means of the following lines of JavaScript code:

```
connection.send(JSON.stringify({
  command: 'pengine_next',
  pid: pid
}));
```

The pengine responds with a new JSON structure representing the second solution to the query `?-mortal(X)`:

```
{ "type":"success",
  "pid":"9a343810-dbe8-11e6-bd74-bfe12f8b98b6",
  "data":[{"X":"plato"}],
  "more":false
}
```

Since the value `false` of the `more` property now indicates that this is the last solution, the shell is updated to reflect this:

```
?- mortal(X).
X = socrates ;
X = plato.
?-
```

Messages recognised by the shell fall under different *types*. In addition to the message type that indicates success, there is a type that indicates failure of a query, and a type that indicates errors. Other message types are related to I/O. When the `echo/0` predicate defined in Section 2.1 calls `read/1` for example, the message that is sent to the shell is of type `prompt`:

```
{ "type":"prompt",
  "pid":"9a343810-dbe8-11e6-bd74-bfe12f8b98b6",
  "data":"|:"
}
```

When the programmer enters a term in response to the prompt, an `input` message is sent to the node and forwarded to the pengine:

```
connection.send(JSON.stringify({
  command: 'pengine_respond',
  pid: pid,
  term: 'hello'
}));
```

And when the pengine calls the `writeln/1` predicate, the following message is sent to the shell, which knows exactly what to do with it:

```
{ "type":"output",
  "pid":"9a343810-dbe8-11e6-bd74-bfe12f8b98b6",
  "data":"hello"
}
```

A fullblown web-based IDE for traditional Prolog is a fairly demanding type of web application. Although the conversation between the programmer and the pengine must always be initiated by the programmer using the shell, the interaction may at any point turn into a mixed-initiative conversation driven by requests for input made by the running query. The echo program in Section 2.1 is a trivial example of this. A game engine that very frequently needs to communicate the state of the game to a client would serve as a less trivial example. What makes unconstrained mixed-initiative interaction feasible is the support for efficient bi-directional messaging that an ACTOR node supports thanks to the use of WebSockets. Other kinds of web applications put less demands on a node and the language primitives that it must support. They may be happy treating a node as a mere "logic server", and thus have no need for mixed-initiative interaction. Then, as we shall see, HTTP may suffice.

2.4 Some additional language capabilities

The language capabilities of a Prolog Web node may (depending on its profile) include not only full traditional Prolog (excluding capabilities that do not fit the intended domain), but

editor.

also a number of primitives for spawning actors and sending messages between them. Such primitives have worked well for Erlang, and we believe that they will work well for Web Prolog too. As we shall see, we are able to build a Prolog Web on top of them in a principled way.

Using such primitives, we can for example easily write a program capable of responding with a *pong* to any client that passes along its pid in a *ping* message. In the Erlang community, such a program is usually referred to as a *server*. To write a ping server in Web Prolog we simply paste the following source code into the editor and tell our pengine to consult it:

```
server :-
    receive({
        ping(Pid) ->
            Pid ! pong,
            server
    }).
```

This code contains two primitives foreign to traditional Prolog: a receive predicate that is used to select and extract messages appearing in the mailbox of the process running the program, and a send operator for sending messages to the mailboxes of other processes. The basic operator for sending a message is `!/2` and it is used in the form `Pid ! Message`. The predicates `receive/1-2` and `!/2` are inspired by Erlang, and so is `spawn/2-3`, which is used for the creation of processes. This is how we spawn the server process and take it for a trial run:

```
?- spawn(server, Pid, [
       src_predicates([server/0])
   ]).
Pid = '8dca3c82-e24b-11e6-8f24-67ab0e08ee11'.
?- self(Self).
Self = '7c82cd5e-4b47-11e6-b903-6740407cea21'.
?- $Pid ! ping($Self).
true.
?- flush.
Shell got pong.
true.
?-
```

The `src_predicates` option ensures that the code that was injected into the workspace of the top-level pengine is now also injected into the workspace of the spawned server process. Calling `self/1` determines the identity of the top-level pengine, to which we want the server to return the pong message, and `!/2` is used to send a ping message to the server. Calling the utility predicate `flush/0` – also copied from Erlang – allows us to inspect the content of the top-level mailbox, where we do indeed find our pong message.

Note also the use of another shell utility feature borrowed from SWI-Prolog. Bindings resulting from the successful execution of a top-level goal may be reused in future top-level goals as `$Var`. (If the same variable name is used in a subsequent goal the system associates the variable with the latest binding.) Together with `flush/0`, this facility comes in very handy when doing interactive/conversational Web Prolog programming in the SWISH shell.

In addition to the `src_predicates` option, `spawn/2-3` supports a number of other options, some of which provide alternative ways to inject source code into the workspace of a pengine. Furthermore, a `node` option allows the programmer to spawn the actor process on a remote node instead of locally, and other options allow the caller to monitor the spawned process or to terminate it under certain conditions. A lot more details will be given further ahead.

Being able to spawn processes and send and receive messages is great for a particular approach to concurrent programming known as *Erlang-style concurrency*. We can for example easily create two actors, running in parallel and possibly also on different nodes, that play ping-pong with each other – the closest to a Hello World in concurrency oriented programming language that we can aim for. In Section 4.4 we will actually show how to do this.

A node that offers the ACTOR profile also comes with the ability to create pengines, i.e. more instances of the kind of actors that we are talking to with the help of the shell. Below we show how to create and interact with a pengine process that runs as a child of the current top-level process. Indeed, what we have here is a pengine running another pengine, a Prolog top-level running another Prolog top-level:

```
?- pengine_spawn(Pid, [
        node('http://node-2.org')
   ]),
   pengine_ask(Pid, member(X,[a,b,c]), [
        template(X)
   ]).
Pid = '7528c178-e0b4-11e6-9726-9bc695b6ff43'.
?- flush.
Shell got success('7528c178-e0b4-11e6-9726-9bc695b6ff43',[a], true)
true.
?- pengine_next($Pid, [
        limit(2)
   ]).
?- flush.
Shell got success('7528c178-e0b4-11e6-9726-9bc695b6ff43',[b,c], false)
true.
?-
```

There is quite a lot going on here. The `node` option of `pengine_spawn/1-2` (which it inherits from `spawn/2-3`) allows us to create the pengine on a remote node. Given the pid returned when calling `pengine_spawn/1-2`, we then call `pengine_ask/2-3` with the query of our choice, and by setting the `template` option we decide the form of answers. Answers are returned to the mailbox of the calling process (i.e. in this case the mailbox belonging to the pengine that is running our top-level). We can inspect them by calling `flush/0`. By calling `pengine_next/2` with the `limit` option set to 2 we then ask for the last two solutions, and finally call `flush/0` again in order to view them.

Last but not least, Web Prolog offers a predicate `rpc/2-3` capable of non-deterministic remote procedure calls:

```
?- rpc('http://node-2.org', human(Who)).
Who = socrates ;
Who = plato.
?-
```

While rpc/2-3 relies on synchronous communication with a node, the pair of predicates promise/3-4 and yield/2-3 allows a programmer to ask a query, do something else while it is running, and only later collect the answer:

```
?- promise('http://remote.org', human(Who), Reference, [
       offset(1)
   ]).
Reference = 'f7780f96-eda8-4117-9b88-92cd1e90d4f0'.
?- yield($Reference, Answer).
Answer = success(anonymous,[human(plato)],false).
?-
```

It may not be evident at this point, but one way (but hardly the only way) to implement rpc/2-3 and promise/3-4 is to do it on top of the pengine abstraction. We thus have three levels of abstractions: the *actor*, the *pengine* (built on top of an actor) and the *non-deterministic remote procedure call* (built on top of a pengine). Moreover, we may want to treat the *node* as a fourth abstraction (also built on top of the pengine abstraction). In chapters that follow we intend to show how these abstractions are related.

2.5 Under the hood (again)

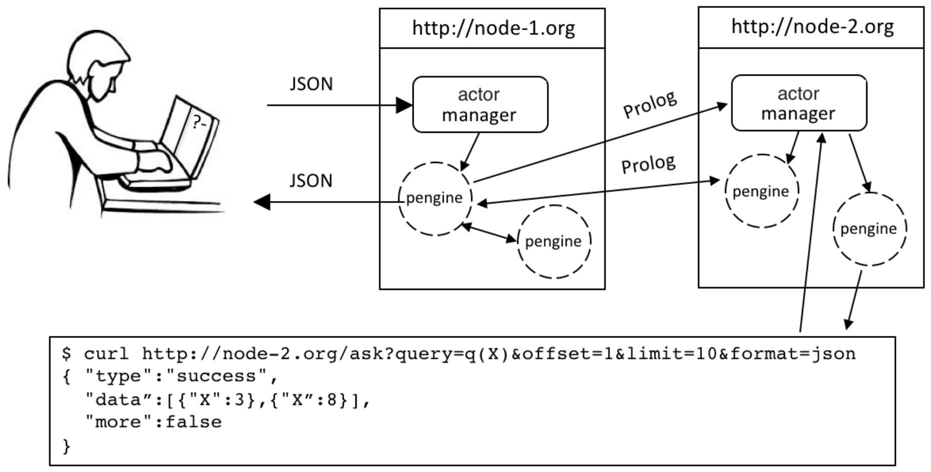
In our first scenario, the programmer was in control of just one single pengine. In other scenarios, this pengine may in turn be in control of and communicate with other pengines. They may be running sequentially, or concurrently. They may be running on the same node, or be distributed over other nodes. That is, in some scenarios, a programmer is in control of the orchestration of what we will refer to as a *multi-pengine system*. (Here we exploit the analogy with so called "multi-agent systems".)

Figure 2.3 depicts, very sketchily, a situation where a programmer uses SWISH in order to create and control a small system consisting of just three pengines. Another client is querying a forth pengine by means of an HTTP request to the node at http://node-2.org.

As we have already mentioned, and what will be explained in detail in Chapter 5, pengines are a kind of actors. What distinguishes pengines from other kinds of actors is the *protocol* they follow when they communicate, i.e. the kind of messages they listen for, the kind of messages they send, and the behaviour that this gives rise to. The pengine protocol allows a client to ask queries and a pengine running on a node to respond with answers. All pengines follow this protocol. The shell adheres to it as well, and even a human user of a shell talking to a pengine must adapt to it in order to have a successful interaction with the pengine.

The uniformity behind the interactions taking place here should be evident. The way the user talks to the shell, so does the shell talk to the pengine running on http://node-1.org, and so does the pengine running on http://node-1.org talk to the pengine running on http://node-2.org, and so on. The conversations share a common structure and adhere to the same protocol. The "sentences" they speak may look different on the surface – as "sentences" in the Prolog language of queries and answers, in the language of HTTP requests and responses, and in the language of JSON tend to do. But the meaning is the same – they can for example be translated into each other without loss or ambiguity. We could possibly even claim that they belong to the *same* language.

An ACTOR node is required to support both the WebSocket and HTTP web APIs. When making the choice between them, a programmer must be aware, though, that unrestricted messaging is not supported over HTTP. (In Section 6 we will look closer at how the HTTP API works, how this is related to the way pengines work, and why unrestricted sending of messages cannot be supported.)



```
$ curl http://node-2.org/ask?query=q(X)&offset=1&limit=10&format=json
{ "type":"success",
  "data”:[{"X":3},{"X”:8}],
  "more":false
}
```

Figure 2.3: Mediated by the shell, a programmer controls a small *multi-pengine system* consisting of just three pengines. Another client is accessing a fourth pengine by means of an HTTP request to the node at `http://node-2.org`.

2.6 Other Web Prolog profiles

A node announces itself as a supporter of a particular Web Prolog *profile*. A profile is a set of *capabilities*. Some are language capabilities, others are interactional capabilities. That is, a profile determines a Web Prolog language fragment as well as a communication protocol and one or more web APIs that supports it. In this report, four profiles will be presented and discussed in some details: ACTOR, ISOTOPE, ISOBASE and RELATION. There is hierarchy here. Nodes implementing the ACTOR profile are more capable than nodes implementing the ISOTOPE profile, and nodes implementing the ISOTOPE profile are more capable than nodes implementing the ISOBASE profile. The RELATION profile is the least capable of these four. Table 2.6 presents a overview of the profiles supported by Web Prolog.

In short, running against a node that offers the ACTOR or ISOTOPE profile, we can both query and program a pengine, but the ACTOR profile alone allows us to use spawn, send, receive, assert and retract in queries or programs. Running against an ISOBASE or RELATION node, we can only query it, and only the ISOBASE profile allows us to pose queries using built-in predicates. By far the most space will be devoted to the presentation of the ACTOR profile. The other profiles are dealt with in Chapter 10.

	ACTOR	ISOTOPE	ISOBASE	RELATION
Runs basic Prolog queries	✓	✓	✓	✓
Runs complex Prolog queries	✓	✓	✓	
Runs injected source code	✓	✓		
Concurrency Messaging	✓			

Table 2.1: A comparison of Web Prolog profiles

SWI-Prolog serves as the host language for the Web Prolog system presented in this report. In the future, we hope that developers of other Prolog systems, as well as developers of programming systems supporting other languages, will also implement Web Prolog. This will allow such systems to talk to each other. A node supporting the ACTOR profile is rather difficult to implement however, so by cutting up of the Web Prolog language into different profiles, some of which are comparatively easy to implement correctly, the chances of this actually happening is probably increased. Our decisions on exactly where the borders between different profiles should lie have been informed mainly by ease of implementation, the choice of architectural style (i.e. RESTful vs non-RESTful) as well as the capabilities of the transport protocols that are used on the Web (i.e. HTTP and WebSocket).

Embedding in other host environments is feasible and gets easier the less capable node one is aiming for. Implementing a node with actor capabilities from scratch in a low-level language would be extremely challenging, whereas the implementation of a node which is only able to offer the RELATION profile is fairly easy.

For a node to be able to offer a profile, it must be *capable* of computing with the language specified by the profile. A node owner may choose to offer less than what a node is actually capable of.

Chapter 3

The Web Prolog ACTOR profile

In this chapter we present the ACTOR profile of Web Prolog, a profile that supports features that make Web Prolog into an actor-based programming language fairly similar to Erlang. As we will show, this seems to be a good fit for a language that aspires to become a web programming language.

Actors, as specified in (Hewitt et al., 1973), involve entities that have their own mailbox. Communication with and among actors is asynchronous by nature, and actors have location transparency that spans runtimes and machines – if we have a reference of an actor, we can message it.

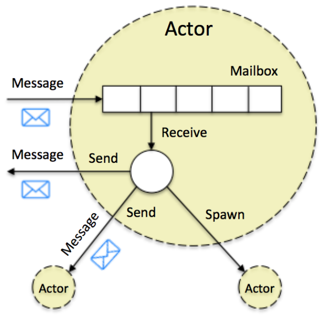


Figure 3.1: An actor

Several languages and libraries are branded as actor programming languages, yet the decision to seek inspiration from Erlang (rather than from other actor programming languages such as Akka, Rust, or Elixir) was easy to make. The syntax of Erlang is very close to the syntax of Prolog, and it made sense to take advantage of this when designing Web Prolog. After all, making Erlang programmers feel at home with Web Prolog could turn out to become just as important as making Prolog programmers feel at home with the language.

This chapter present a number of programming examples, many of them based on the assumption that a remote node contains the following node-resident program, an instance of the well-known *family tree* example:

```
ancestor_descendant(X, Y) :- parent_child(X, Y).
ancestor_descendant(X, Z) :- parent_child(X, Y), ancestor_descendant(Y, Z).

parent_child(X, Y) :- mother_child(X, Y).
parent_child(X, Y) :- father_child(X, Y).

mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).
```

The majority of the examples are there to show that Web Prolog programs can be written in the style of Erlang, whereas the ones towards the end of the chapter show how to deal with the non-determinism that is not present at all in Erlang.

A number of Web Prolog programming patterns inspired by similar patterns in Erlang will also be demonstrated: The use of concurrency and recursion for maintaining state, the use of protocols, and various abstractions for asynchronous and synchronous communication between processes, just to mention a few examples.

3.1 Programming with spawn, send and receive

The Web Prolog ACTOR profile introduces five core predicates inspired by the Erlang programming language: `self/1`, `spawn/2-3`, `!/2`, `receive/1-2` and `exit/1-2`. Consider the following run of a query involving four of them:

```
?- self(Self),
   spawn((mother_child(trude, Child), Self ! Child)),
   receive({Who -> true}).
Self = '09fe1809'@'http://localhost:3060',
Who = sally,
?-
```

Calling `self(Self)` binds `Self` to the "name" of the current process,[1] normally referred to as a *process identifier*, or a *pid*. Pids are used for sending messages, and can themselves be part of a message, allowing other actor processes to communicate back.

After having found the pid of the current process, `spawn/2` is called with a query in the first argument that looks for a child of Trude and, using `!/2`, sends the result back to the mailbox of `Self`, where it is picked up by the call to `receive/1`. Note that the spawned process, identified as `Pid`, runs a *copy* of the query and that is why `Child` is not bound in the process.

Simple as this example may seem, it serves to demonstrate an important principle: If you want an answer, you must provide a return address! That is, for a spawned process to return anything to the process that spawned it, the pid of the latter must be included in the goal passed to `spawn/2`.

[1] Recall that this may not be the process that the shell is running in.

Although that was not the case in the above example, there may in general be more than one message in the mailbox, where they will form a queue. Calling `receive/1-2` once can only extract at most one message before terminating. If no message is present, the call will just hang, and resorting to the `Ctrl-C` may be the only way out. In such cases, the Web Prolog utility predicate `flush/0` comes in handy as it extracts all the messages currently in the mailbox of the top-level process and prints them to the shell. The execution of `flush/0` never hangs.

```
?- self(Self),
   spawn((mother_child(trude, Child), Self ! Child)).
Self = '09fe1809'@'http://localhost:3060'.
?- flush.
Shell got sally
true.
?-
```

`flush/0` is implemented in terms of `receive/2`:

```
flush :-
    receive({
        Message ->
            io:format("Shell got ~q~n",[Message]),
            flush
    },[timeout(0)]).
```

This implementation serves as a good example of the use of a programming pattern frequently found in Erlang programs and destined to become very useful in Web Prolog too: a loop is defined where a call to `receive/1-2` is used to match a message in the mailbox, do something with it, and then continue looping by making a recursive call. In the case of `flush/0`, the value `0` of the `timeout` option ensures that the loop terminates immediately if no messages remain in the mailbox. This is how the hanging of the call is avoided.

A process uses the receive operation to extract messages from its mailbox. This operation specifies an ordered sequence of *receive clauses* wrapped in curly brackets and delimited by semicolons. A receive clause always has a *pattern* (a term) and a *body* of Prolog goals. Optionally, it may also have a *guard*, which is a goal prefixed with the `when` operator. Its role is to make pattern matching more expressive. Schematically:

```
{  Pattern1 [when Guard1] ->
         Body1;
   ...
   PatternN [when GuardN] ->
         BodyN
}
```

When `receive/1-2` is called, it scans the mailbox looking for the first message (i.e. the oldest) that matches any of the patterns and satisfies the corresponding guard (if any), blocking if no such message is found. If a matching clause is found, the message is removed from the mailbox and the body of the clause is called. Values of any variables bound by the matching of the pattern with the message and the evaluation of the guard will be available in the body.

Any goal may be used as a guard, but it is wise in general to keep guards as simple and efficient as possible and to avoid side effects and non-determinism. Erlang enforces purity and efficiency by only allowing a restricted set of operators in guards, and completely disallows calling user defined functions. For the use cases the people behind Erlang had, it probably made sense to impose such restrictions. We do not think we should impose them on Web Prolog though. The ability to carry the values of variables bound by a guard forward to the body of the receive clause seems like a very nice thing to have. To counter possible non-determinism, goals used as guards will run as if they were wrapped in `once/1`.

If no pattern matches the first message in the mailbox, the message is *deferred*, possibly to be handled later in the control flow of the process. The receive is still running, waiting for more messages to arrive, and for one that will match. This behaviour is particularly useful if we expect two messages, `hello` and `goodbye`, but are not sure which one will arrive first. If we insist on processing `hello` before `goodbye`, we can easily do that with receive:

```
wait_hello :-                          wait_goodbye :-
    receive({                              receive({
        hello ->                               goodbye ->
            io:write('Got hello!'),                io:write('Got goodbye!')
            wait_goodbye                       }).
    }).
```

Even if `goodbye` is queued before `hello`, calling `wait_hello/1` would result in `hello` being selected before `goodbye` and "Got hello!" would therefore be printed before "Got goodbye!". This is why the receive operator is often referred to as *selective* receive.

Clearly, Erlang's receive construct offers flexibility with respect to the way a particular message is selected from the mailbox, thus creating many degrees of freedom in the order in which messages may appear – a relief since in a concurrent and distributed environment not many guarantees on ordering can be given since a high degree of indeterminacy is introduced by the fact that different segments of a parallel computation can be distributed throughout space and be executed at indeterminate points in time relative to one another. Selective receive and its underlying deferring mechanism are therefore extremely important; without them, neither Erlang nor Web Prolog would be possible in their current forms.

As a way to demonstrate the use of the `when` operator, as well as the use of another `receive/2` option that causes a goal to run on timeout, here is a priority queue example borrowed from (Hebert, 2013).[2] The purpose of the following code is to build a list of all messages with those with a priority above 10 coming first:

```
important(Messages) :-
    receive({
        Priority-Message when Priority > 10 ->
            important(MoreMessages),
            Messages = [Message|MoreMessages]
    }, [ timeout(0),
         on_timeout(normal(Messages))
    ]).

normal(Messages) :-
    receive({
```

[2]Compare Erlang version at http://learnyousomeerlang.com/more-on-multiprocessing#selective-receives

```
        _-Message ->
            normal(MoreMessages),
            Messages = [Message|MoreMessages]
    }, [ timeout(0),
         on_timeout(Messages=[])
    ]).
```

Below, we test this procedure by first sending four messages to the top-level process, and then calling `important/1`:

```
?- self(S), S ! 15-high, S ! 7-low, S ! 1-low, S ! 17-high.
S = b0f80b2d@'http://localhost:3060'.
?- important(Messages).
Messages = [high,high,low,low].
?-
```

3.2 A comparison with Erlang

Here is how Hebert implements his priority queue example in Erlang:

```
important() ->
    receive
        {Priority, Message} when Priority > 10 ->
            [Message | important()]
    after 0 ->
        normal()
    end.

normal() ->
    receive
        {_, Message} ->
            [Message | normal()]
    after 0 ->
        []
    end.
```

We have made our best to make the Web Prolog receive primitive look like the Erlang construct, but there are definitely some major syntactic differences that need to be discussed. First, the relational syntax of Web Prolog demands that we perform some unnesting of the functional notation used in the Erlang example. This is a purely syntactical operation that can be done mechanically.

Secondly, while Prolog's `op/3` predicate allowed us to define the infix `when` operator, there is not much that can be done about the `receive...after...end` construct. This is why we chose to treat the whole construct as a binary predicate expecting an ordered sequence of receive clauses wrapped in curly brackets in its first argument and the `after...` part as a list of options implementing its intended semantics in the second argument.

While we could have taken advantage of the fact that an Erlang tuple such as {`Priority, Message`} is valid syntax in Prolog too, where it is somewhat more complex compound term of the form {}`((Priority, Message))`, we chose not to. In Prolog, it is always better to use less complex terms, taking up less memory. In this case the operator `-` was the most natural choice, but we could have used any valid Prolog term.

Getting the semantics of `receive/1-2` right is of course more important than getting the syntax right. We believe that we have succeeded in so far as it allows any use of receive in Erlang to be mechanically translated into a use of receive in Web Prolog. However, as we shall see in Section 3.6, the receive construct in Web Prolog has a surprise up its sleeve: it is a semi-deterministic predicates, i.e. it either fails or succeeds exactly once. As we will show in Section 4.6, this is an important property of `receive/1-2` which allows us to handle backtracking in Web Prolog in an elegant way.

At this point, it might be a good idea to ask ourselves to what extent we can count on being able to (more or less directly) translate Erlang programs into Web Prolog programs. Most examples in the report are too small to show much, but in Appendix C we have translated an Erlang program solving the *Dining Philosophers problem*[3] into Web Prolog. The experience with this slightly bigger program – which uses seven concurrently running actor processes – showed us that such translation is indeed fairly easy to perform. The resulting program can be taken for a test drive in the tutorial described in Section 1.9.

3.3 Forcing exits

Having spawned a process running a goal, it sometimes happens that we need to terminate it by force. This is where the `exit/2` predicate comes in. Consider the following session:

```
?- self(Self),
   spawn((repeat, fail), Pid, [
        monitor(true)
   ]).
Pid = '4e2a0ff3',
Self = b0f80b2d@'http://localhost:3060'.
?- exit('4e2a0ff3',my_reason).
true.
?- flush.
Shell got down('4e2a0ff3'@'http://localhost:3060',exited(my_reason))
true.
?-
```

By calling `exit/2` with the pid in the first argument and a term detailing the reason for exiting in the second, we were able to terminate the runaway process. Setting the option `monitor` to `true` allowed us to get informed about the outcome. We say more about this option in the next section.

3.4 Monitoring and linking

Below, we investigate the change of behaviour introduced by passing the `monitor` option to `spawn/3`:

[3] https://en.wikipedia.org/wiki/Dining_philosophers_problem

```
?- self(Self),
   spawn((mother_child(trude, Who), Self ! Who), Pid, [
       monitor(true)
   ]).
Pid = '88a67bd4',
Self = '9d03409e'@'http://localhost:3060'.
?-
```

In this way, we made sure that the spawning process is informed about what eventually will be the fate of the spawned process:

```
?- flush.
Shell got sally
Shell got down('88a67bd4'@'http://localhost:3060',exit)
true.
?-
```

The second of these messages serves to inform the spawning process that the spawned process has terminated and `true` in the second argument of the message term means that the goal succeeded (whereas `false` would have meant that it failed). Replacing the subgoal `mother_child(trude, Who)` with (say) `mother(trude,Who)` would (since `mother/2` is not defined) result in just one `down` message with a detailed error message in the second argument.

An actor is spawned on the initiative of a client that is itself an actor or another process (e.g. a shell process). If the value of the `link` option is `true`, it means that when the client terminates the child actor also terminates. For an authenticated user with the right authorisation it will be possible to set the `link` option to `false` in order to create what we think of as an *independent* and possibly (or even typically) *long running* actor. Such actors are free, not dependent on their lives on their creators.

When a process A spawns a process B, B becomes the *parent* of A, and A the *child* of B. Since B in turn may spawn other processes, the processes involved form an hierarchy. Often, A can be seen as the *supervisor* of B, and B the supervisor of any children it might have. Thus, we have what in Erlang lingo is referred to as a *supervisor hierarchy*.

Consider the case of an anonymous JavaScript client A, spawning a process on a node B, which in turn spawns two processes C and D. These four processes form a rather simple and rigid supervisor hierarchy: if A terminates, then B terminates too, and if B terminates, then C and D also terminates. At least for anonymous clients, this is the only rule. Authenticated and authorised users may be permitted to create processes on a node that do not terminate when the parent terminates.

Links in Web Prolog are somewhat simpler than links in Erlang. In contrast to Erlang's bi-directional links, they are uni-directional. As argued in (Svensson et al., 2010), uni-directional links simplify things and do not harm expressivity. The only kind of link currently supported in Web Prolog comes in the form of an option `link` to `spawn/3`. Its value is `true` by default, which causes a child process to terminate if its parent does. Only authorised clients can set it to `false` and thus an unauthorised client cannot spawn a process which is *not* linked to the process that spawned it. This is to avoid leaving orphaned processes around on a node. For the same reason, `unlink/1` is not available to unauthorised clients.

Termination can be automated. As a practical example let us consider how to implement a supervisor behavior using a uni-directional link. Assuming that we want to supervise a child process, specified as the goal `foo(X)`, so that if the supervisor terminates, the child terminates too, then the following supervisor code fragment suffices:

```
...
spawn(foo(X), Pid, [link(true)]),
...
```

Note that this very rigid supervisor hierarchy is an extreme case of an approach to error handling that by Erlang programmers is often referred to as "let it crash". If a user starts a process that spawns other processes (and in turn yet other processes) and something goes wrong, he can usually just make a new attempt. This is an approach that should be good enough for the Web.

Erlang offers a great deal of flexibility with respect to the way the graceful termination or crash of a process influences other processes to which it is related. In Erlang, such mechanisms are programmable, not hardwired. This is true for Web Prolog as well. In Web Prolog, it is easy to get the impression that they are hard wired, but the need to impose a strict master-slave regime comes from the need to avoid leaving orphaned processes around, and is not a limitation of the language. It would be possible to allow authenticated and authorised users to spawn pengines that do not terminate when the process that spawned them terminates.

3.5 Bringing code to the data, or data to the code

In the following example, the option `node` is passed to `spawn/3` with a URI as its value :

```
?- self(Self),
   spawn((mother_child(trude, Who), Self ! Who), Pid, [
        node('http://remote.org'),
        monitor(true)
   ]).
Pid = '88a67bd4',
Self = '9d03409e'@'http://localhost:3060'.
?- flush.
Shell got sally
Shell got down('88a67bd4'@'http://localhost:3060',exit)
true.
?-
```

The `node` option has the effect that the child process is created on the *remote* node identified as `http://remote.org` instead of locally. Errors may of course occur – the node may be down, or a predicate being called may throw a type error or may not be defined by the node. Again, by including the `monitor` option we make sure that the spawning process is informed about such errors.

Up till now, we have assumed that the predicate being called by the goal in the first argument of `spawn/2-3` is defined by the remote node. This is not an assumption we always need to make. Some of the options that can be passed to `spawn/3` can be used to inject Web Prolog source code into the process to be created. (See Section 4.8 for a full list of such options.) In the following example, a clause for `mother/2` is first defined locally and then the option `src_predicates` is used to inject the clause into the process, just before the goal is run:

```
?- assert((mother(Parent, Child) :- mother_child(Parent, Child))).
true.
?- self(Self),
   spawn((mother(Who, sally), Self ! Who), Pid, [
        node('http://remote.org'),
        src_predicates([mother/2])
   ]).
Pid = '34a677d3',
Self = '9d03409e'@'http://localhost:3060'.
?- flush.
Shell got trude
true.
?-
```

Bear in mind that when the clause for `mother/2` is injected into the spawned process, it becomes available there only, and will disappear completely when the process terminates. Other processes running on the same node will not see it.

One may think about this in the following way: The `src_* options` make a kind of sharing of knowledge between a parent actor and its child actor possible. The child is "born" with this knowledge – the knowledge is "innate". It is up to the parent actor to determine exactly what should be part of the child's knowledge. Through the use of the `node` option it is also up to the parent actor to decide in which environment the child should live, if it should stay with the parent or be "adopted away" to live in a Prolog context different from the parent's. Informed by data coming from the parent as well as from the environment in which it is born, it will form a certain behaviour.

The ability to inject source code into a remote process allows a developer to *bring code to the data*, rather than the other way around. To instead *bring the data to the code*, we can do something like this:

```
?- self(Self),
   spawn((mother_child(Who, sally), Self ! Who), Pid, [
        src_uri('http://remote.org/src')
   ]).
Pid = 'dea617e3',
Self = '9d03409e'@'http://localhost:3060'.
?- flush.
Shell got trude
true.
?-
```

Here the goal is run locally, but the code is fetched from a remote server.

We conclude that there is an useful symmetry here, allowing Web Prolog code to flow in either direction, from the client to the node as well as from the node to the client. However, bringing the data to the code will only work if the data is complete with respect to the goal. In general, this cannot be guaranteed. If the remote source code calls predicates that are not defined by the Web Prolog language, but are instead imported from a library available only on the remote node, an exception might be thrown at runtime.

3.6 The notion of *server* in Web Prolog

So far in this chapter our examples of how to program in Web Prolog have been really trivial. Spawned processes have terminated almost immediately and `!/2` has been used only to send variable bindings to the top-level process. A much more typical use of `spawn/2-3` is to call a locally or remotely defined Web Prolog procedure that specifies the (usually longtime) behaviour of the actor process thus created, the kind of messages it will listen for, and what kind of messages will be sent to other actors. As we have hinted at already in Section 2.4, some such actors are referred to as *servers* in the Erlang community. Servers can be either *stateless*, or *stateful*.

Simple stateless servers returning pong messages in return for ping messages can be written like so:

```
server :-                               server(Pid) :-
    receive({                               receive({
        ping(Pid) ->                            ping ->
            Pid ! pong,                             Pid ! pong,
            server                                  server(Pid)
    }).                                     }).
```

The difference between the server to the left and the server to the right is that the server to the left is capable of responding to any client that passes along its pid in the message, whereas the server to the right is able to serve only one process, the identity of which is determined when the server is spawned. We saw the left server implementation in action already in Section 2.4, and here is how the server to the right runs if we first spawn it and then act as its client:

```
?- self(Self),
   spawn(server(Self), Pid).
Self = 'f431a324-e549-11e6-ab9e-5ba20c9afb71',
Pid = '8915b2d4-e24b-11e6-9ef0-6f680d1ff1cc'.
?- $Pid ! ping.
true.
?- flush.
Shell got pong
true.
?-
```

To implement a looping behaviour, Prolog programmers occasionally use a *repeat-fail loop* that relies on backtracking rather than recursion. Using this technique, the servers presented above can be rewritten like so:

```
server :-                               server(Pid) :-
    repeat,                                 repeat,
    receive({                               receive({
        ping(Pid) ->                            ping ->
            Pid ! pong,                             Pid ! pong,
            fail                                    fail
    }).                                     }).
```

For an Erlang programmer this particular use of `receive/1` may come as a surprise. After all, the Prolog notions of *failure*, *backtracking* and the use of failure to force backtracking are foreign to Erlang. Prolog programmers may recognise a behaviour due to the fact that `receive/1-2` is a *semi-deterministic* predicate, i.e. a predicate that either fails or succeeds exactly once. We may want to compare `receive/1` with `read/1` in traditional Prolog, or with `thread_get_message/1` in the ISO Prolog Threads draft standard, both of which are also semi-deterministic, and also serve the purpose of receiving data from the environment. But while `read/1` and `thread_get_message/1` fail if the pattern (consisting of any term) in the argument does not unify with the term that is read or received, the only way that `receive/1-2` will fail is if the goal in the *body* of one of its receive clauses fails. We would suggest that this is the only reasonable semantics for `receive/1-2`.[4] To see how it pans out in a corner case, consider the following two code snippets:

```
    ...                                   ...
    receive({foo(X) -> true}),            receive({foo(X) -> fail}),
    ...                                   ...
```

In the situation to the left, the receive call will succeed (and leave no choice points) if a message matching the pattern `m(X)` appears in the mailbox. In the situation to the right, the receive call will fail (and possibly cause backtracking) once a message matching the pattern `foo(X)` appears. Only in the left situation will the variable `X` be bound; in both cases, the matched message will be removed from the mailbox.

Let us now turn to *stateful* servers and show how state may be programmed by using the two different programming patterns, one based on recursion and one based on backtracking. The two servers below implement *counters*. For the purpose of looping, the server on the left uses recursion, whereas the one on the right uses backtracking. They represent the state (i.e. the current count) in different ways, and are started slightly differently, but once they run they will behave in exactly the same way:

```
server(Pid, Count0) :-                server(Pid) :-
    receive({                             between(1, inf, Count),
        next ->                           receive({
            Pid ! Count0,                     next ->
            Count is Count0 + 1,                  Pid ! Count,
            server(Pid, Count)                    fail
    }).                                   }).
```

The server to the right demonstrates a pattern that we shall return to when we develop a *generic* query solver in Section 4.7.

As a tastier example of how a process can be made to hold an updatable state during a conversation, and for the purpose of demonstrating a receive call that uses different receive clauses in order to listen for more than one kind of message, we have adapted a fridge simulation example from Fred Hebert's excellent book on Erlang (Hebert, 2013):[5]

```
fridge(FoodList0) :-
    receive({
        store(From, Food) ->
            self(Self),
```

[4] What else could it possibly mean if the goal in the body of a receive clause fails? We would argue that letting the whole call to `receive/1-2` fail gives us the only reasonable semantics.

[5] `http://learnyousomeerlang.com/more-on-multiprocessing#state-your-state`

```
                From ! ok(Self),
                fridge([Food|FoodList0]);
            take(From, Food) ->
                self(Self),
                (   select(Food, FoodList0, FoodList)
                ->  From ! ok(Self, Food),
                    fridge(FoodList)
                ;   From ! not_found(Self),
                    fridge(FoodList0)
                );
            terminate ->
                true
        }).
```

The program creates a process allowing three operations: storing food in the fridge, taking food from the fridge, and terminate the fridge. It is only possible to take food that has been stored beforehand. Again, with the help of recursion the state of a process can be held entirely in the argument of the predicate. In this case we choose to store all the food as a list, and then look in that list when someone needs something.

Assuming that the above program is already available on a remote node, the following session creates the server process there. We can then use the shell, its top-level pengine, and `self/1`, `!/2`, and `flush/0` to act as a client:

```
?- spawn(fridge([]), Pid, [
       node('http://remote.org'),
       monitor(true)
   ]).
Pid = 'a6035b1a-3906-11e6-a001-cf4de65fa1fc'.
?- self(Me), $Pid ! store(Me, meat), $Pid ! store(Me, cheese).
Me = '4806702e-384d-11e6-96f4-8fcbb6c814f5'.
?- flush.
Shell got ok('a6035b1a-3906-11e6-a001-cf4de65fa1fc')
Shell got ok('a6035b1a-3906-11e6-a001-cf4de65fa1fc')
true.
?- self(Me), $Pid ! take(Me, cheese).
Me = '4806702e-384d-11e6-96f4-8fcbb6c814f5'.
?- flush.
Shell got ok('a6035b1a-3906-11e6-a001-cf4de65fa1fc',cheese)
true.
?- $Pid ! terminate.
true.
?- flush.
Shell got down('a6035b1a-3906-11e6-a001-cf4de65fa1fc', true)
true.
?-
```

So far, we expect the programmer to know the details of the required protocol and to interact with our fridge by making raw call using the send operator in combination with `flush/0` utility predicate. As suggested also by Fred Hebert in his book, that is a useless burden, and good way to solve this is to abstract messages away with the help of predicates (or in Hebert's case functions) dealing with receiving and sending them:

```
store(Pid, Food, Response) :-
    self(Self),
    Pid ! store(Self, Food),
    receive({
        Pid-Response -> true
    }).

take(Pid, Food, Response) :-
    self(Self),
    Pid ! take(Self, Food),
    receive({
        Pid-Response -> true
    }).
```

Using `store/3` and `take/3` interacting with becomes somewhat easier:

```
?- spawn(fridge([]), Pid, [
       node('http://remote.org'),
       monitor(true)
   ]).
Pid = 'e8035b1a-1906-11e6-a011-cf4de65fa1fc'.
?- store($Pid, cheese, Response).
Response = ok('e8035b1a-1906-11e6-a011-cf4de65fa1fc').
?- take($Pid, cheese, Response).
Response = ok('e8035b1a-1906-11e6-a011-cf4de65fa1fc', cheese).
?-
```

In the case of pengines, to be dealt with in Chapter 5, such abstraction will turn out to be of considerable value due to the complexity of the protocol. [TODO: This example doesn't work, and must be fixed.]

3.7 A universal stateful server with hot code swapping

Inspired by one of Joe Armstrong's lectures on Erlang[6] this is how we may program a *generic* stateful server in Web Prolog which can also handle *hot code swapping*:

```
server(Pred, State0) :-
    receive({
        rpc(From, Ref, Request) ->
            call(Pred, Request, State0, Response, State),
            From ! Ref-Response,
            server(Pred, State);
        upgrade(Pred1) ->
            server(Pred1, State0)
    }).
```

The first receive clause matches incoming rpc messages specifying a query, performs the

[6] `http://youtu.be/0jsdXFUvQKE`

required computation, and returns the answer to the client that submitted the query. The second clause is for upgrading the server.

As can be seen from the first receive clause, the generic server expects the definition of a predicate with four arguments to be present and callable from the server. In the case of our refrigerator simulation the expected predicate may be defined as follows in order to obtain the required specialisation of the generic server:

```
fridge(store(Food), FoodList, ok, [Food|FoodList]).
fridge(take(Food), FoodList, ok(Food), FoodListRest) :-
    select(Food, FoodList, FoodListRest), !.
fridge(take(_Food), FoodList, not_found, FoodList).
```

If we choose, the client can be built in a style, also commonly practised by Erlang programmers, that guarantees that the communication between client and server stays synchronous:

```
rpc_synch(To, Request, Response) :-
    self(Self),
    make_ref(Ref),
    To ! rpc(Self, Ref, Request),
    receive({
        Ref-Response -> true
    }).
```

Note the generation of a unique reference marker to be used to ensure that answers pair up with the questions to which they are answers.

In case we cannot rely on the server always being up and running, we may want to implement a timeout, like so:

```
   ...
   receive({
       Ref-Response -> true
   },
   [ timeout(10),
     on_timeout(Response=timeout_exceeded)
   ]).
```

This code too is generic and follows a common idiom in Erlang that implements a synchronous operation on top of the asynchronous send and the blocking receive. The predicate `rpc_synch/3` waits for the response to come back before terminating. It inherits this blocking behaviour from `receive/1`, and it is this behaviour that makes the operation synchronous. (We will see more of this pattern later.)

Here is how we start a server process on a remote node and then use `rpc_synch/3` to talk to it:

```
?- spawn(server(fridge, []), Pid, [
       node('http://remote.org')
   ]).
Pid = '4806702e-384d-11e6-96f4-8fcbb6c814f5'.
?- rpc_synch($Pid, store(meat), Response).
Response = ok.
?- rpc_synch($Pid, take(meat), Response).
```

```
Response = ok(meat).
?-
```

Now, suppose that we want to upgrade our server with a faster predicate for grabbing food from the fridge, perhaps one that uses an algorithm more efficient than the sequential search performed by `fridge/4`. Assuming a predicate `faster_fridge/4` is already loaded and present at the node we can make the upgrade without first taking down the server. This means that we can retain access to the state (and thus not risk losing any food in the process):

```
?- $Pid ! upgrade(faster_fridge).
true.
?-
```

Remote procedure calls being synchronous means that the caller is suspended until the computation terminates and we have to do an idle wait for the answer, although we may have something better to do. An alternative approach may be to use a so called *promise*, an asynchronous variant of a remote procedure call. To implement this, we can split the client code above into two parts, and thus create two predicates, `promise/3` and `yield/2`:

```
promise(To, Request, Ref) :-
   self(Self),
   make_ref(Ref),
   To ! rpc(Self, Ref, Request).

yield(Ref, Response) :-
   receive({
       Ref-Response -> true
   }).
```

We can now separate the sending of request from the receiving of the response, thus trading a somewhat messier code for a little bit of concurrency where the caller can perform the RPC, do something else and try the claim the computed value at a later time, which may or may not be ready. Like so:

```
   ...
   promise(To, store(meat), Ref),
   ... do something else here...
   yield(Ref, Response),
   ...
```

With `promise/3` and `yield/2` defined, we can of course define `rpc_synch/3` as follows instead of as above:

```
rpc_synch(To, Request, Response) :-
    promise(To, Request, Ref),
    yield(Ref, Response).
```

Abstractions such as these can be compared to Erlang *behaviours*. They are not always easy to build, but once they are built they can be fairly easily instantiated and tailored to specific tasks. As we shall see further down in the text, Web Prolog comes with powerful promise and yield primitives built-in.

3.8 The Web Prolog dynamic database

In Web Prolog each actor is equipped with its own workspace, a dynamic database private to it. As long as the actor process is alive, a client is allowed to update the workspace using the predicates `assert/1`, `retract/1` and `retractall/1`. Such updates must be performed via messaging – a client basically must ask the actor to update itself – and they will only affect that actor's workspace, not actors running elsewhere, and not even actors running on the same node. This means that problems caused by two or more processes trying to update the same database simultaneously cannot arise. Keep in mind that since database updates performed by one process is not seen by other processes, assert and retract cannot be used for intra-process communication. Web Prolog adheres to the idea that processes should "communicate to share memory, rather than share memory to communicate".[7]

Therefore, although the following goal is permitted, it is quite meaningless since the clause `foo(a)` disappears as the goal has run and the spawned process has terminated.

```
?- spawn(assert(foo(a)), Pid).
Pid = '75a7715c-64b8-11e7-aed5-8bb40804b542'.
?-
```

The dynamic database gives us an alternative way to maintain the state of an actor. Here is how the fridge simulation may be written using assert and retract:

```
start(Pid) :-
    spawn(fridge, Pid, [
        src_text("

            :- dynamic food/1.

            fridge :-
                receive({
                    store(From, Food) ->
                        assert(food(Food)),
                        self(Self),
                        From ! ok(Self),
                        fridge;
                    take(From, Food) ->
                        self(Self),
                        (   retract(food(Food))
                        ->  From ! ok(Self, Food),
                            fridge
                        ;   From ! not_found(Self),
                            fridge
                        );
                    terminate ->
                        true
                }).
        ")
    ]).
```

[7]A mantra attributed to CSP.

The dynamic database is a non-logical extension to Prolog and the use of it for managing state is usually frowned upon by Prolog programmers since changes to the database are not reverted on backtracking and since clauses of dynamic predicates are like destructible global variables and therefore subject to the usual problems with such entities. However, for storing very large quantities of food in our fridge using the dynamic database may still be the best solution.

A Prolog-style dynamic database is not something that Erlang has, and we need to determine if it could lead to problems in Web Prolog. The answer is “no” since Erlang does in fact have something which plays the same role as the dynamic database. In Erlang, each process has a local store called the *process dictionary* and a handful of functions for manipulating it are built into the language. Process dictionaries are private and cannot be modified by any other process. Erlang programmers are advised again using them since they destroy referential transparency and make debugging difficult.

One cannot help noticing a similarity here. This leads us to believe that the dynamic database in Web Prolog is not more dangerous than the process dictionary in Erlang. It is arguably also more powerful and useful than the process dictionary, which probably means that programmers are tempted to use it more often than is recommended.

Whereas the dynamic database of a Web Prolog and the process dictionary of an Erlang process are private to an actor, Erlang also supports a built-in term storage known as *ETS* and a distributed database named *Mnesia*. They are not only mutable but also *shared* between processes. The question whether or not Web Prolog should allow for some kind of memory over and above what can be passed around in the arguments of predicates or asserted in the private dynamic database is interesting. In line with the focus of Web Prolog on the web domain, a solution based on a transactional RDF database might be appropriate and is easily implemented in (at least) SWI-Prolog, which has a library for it.

Chapter 4

Advanced actor programming

4.1 A note on I/O

Compared to most Prolog dialects, Web Prolog has far fewer built-in predicates for I/O. Since sending and receiving will take over some of the work that writing and reading performs, and since the concepts of files, sockets and streams are non-existent in Web Prolog, there is less need for such predicates. For file I/O, a node's host platform will have to take responsibility.

However, Web Prolog does support a couple of output predicates, notably `write/1`, `writeln/1` and `format/2`. (We saw a use of `writeln/1` above.) The result of using them depends on the client. When they write to a text-based terminal they just output a plain string, while output to a web client is given in the form of JSON, part of which is presented as a string by the client. They are there for the purpose of testing and light debugging from the top-level. It makes sense to implement the writing of a term as the sending of the term to a process designated as a user client.

The difference between *writing* to the shell and *sending a message* to the process that the shell is attached to can be demonstrated in the following way:

```
?- write(p(a)).
p(a)
true.
?- spawn(write(p(a)), _).
p(a)
true.
?- pengine_output(p(a)).
p(a)
true.
?- spawn(pengine_output(p(a)), _).
true.
?- flush.
Shell got output('5416742f-324e-22e1-1644-8f5ff6c544f2', p(a))
true.
?-
```

Note that a shell can only receive and render output that is written from actors that 1) live on the same node as the top-level pengine to which the shell is attached, and 2) are descendants of this top-level pengine. (This restriction may be lifted in the future.) In order to send a message from one node to another, `pengine_output/1` must be used.

4.2 The process registry

Calling `register(Name,Pid)` associates an atom `Name` with a `Pid`. The name can be used instead of the pid when using `(!/2)`. Here is a trivial example associating the atom `me` with the result of calling `self/1`:

```
?- self(S), register(me, S).
S = '1f810f77-18d0-48d7-92ee-a1b4b8f6577e'@'http://localhost:3060'.
?- me ! hi.
true.
?- flush.
Shell got hi
true.
?-
```

The next section demonstrates a more interesting use of `register/2`.

4.3 Node-resident actor processes

As we have already explained, the owner of a node may install node-resident programs, i.e. source code for predicates that can be accessed by any client to this node, as if those predicates were built-in. In addition, the owner may install *node-resident actor processes* – long-lived processes that can be accessed by any client.

Here is all that is needed for a simple publish-subscribe service:

```
pubsub_service(Subscribers0) :-
    receive({
        publish(Message) ->
            forall(member(Pid, Subscribers0), Pid ! msg(Message)),
            pubsub_service(Subscribers0);
        subscribe(Pid) ->
            pubsub_service([Pid|Subscribers0]);
        unsubscribe(Pid) ->
            (   select(Pid, Subscribers0, Subscribers)
            ->  pubsub_service(Subscribers)
            ;   pubsub_service(Subscribers0)
            ).
    }).
```

We will assume that the owner of the node has started the service by running the following goal:

```
?- spawn(pubsub_service([]), Pid),
   register(pubsub_service, Pid).
```

In the following example we subscribe to the service, and invoke a repeat-fail loop waiting for messages to arrive from it:

```
?- self(Self),
   pubsub_service ! subscribe(Self),
   repeat,
   io:write("Waiting for a message ..."),
   receive({
       msg(Message) ->
           io:format("Received: ~p", [Message]),
           fail
   }).
Waiting for a message ...
Received: hello
Waiting for a message ...
```

The message "hello" was received when someone with a connection to the same node published it to the service using the following kind of call:

```
?- pubsub_service ! publish(hello).
true.
?-
```

Node-resident actors are usually of a kind that we do not want unauthorised external clients to be able to create (or destroy). Doing this should be the privilege of the owner of the node or other authenticated users with the right authorities.

4.4 Concurrent programming

Erlang has for a long time been used as a powerful tool for concurrent programming. Introducing spawn, send and receive in Web Prolog allows us to do the same. Asynchronous communication is key to the kind of concurrent programming that Web Prolog's actor abstraction supports.

In the following example,[1] two processes are first created and then start sending messages to each other a specified number of times.

```
ping(0, Pong_Pid) :-
    Pong_Pid ! finished,
    io:format('Ping finished').
ping(N, Pong_Pid) :-
    self(Self),
    Pong_Pid ! ping(Self),
    receive({
        pong ->
            io:format('Ping received pong')
    }),
    N1 is N - 1,
    ping(N1, Pong_Pid).
```

[1] http://erlang.org/doc/getting_started/conc_prog.html#id68696

```
pong :-
    receive({
        ping(Ping_Pid) ->
            io:format('Pong received ping'),
            Ping_Pid ! pong,
            pong;
        finished ->
            io:format('Pong finished')
    }).

start :-
    spawn(pong, Pong_Pid),
    spawn(ping(3, Pong_Pid), _).
```

Here is how we test it:

```
?- start.
true.
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Pong received ping
Ping received pong
Ping finished
Pong finished
?-
```

We can replace `start/0` with the definition below, thus spawning one of the processes on a different node, `http://remote.org`. Assuming that we do not want to install the procedure for `pong/0` on this node (or if we are not permitted to do so), we choose to use the `src_predicates` option in order to send it along to the process instead:

```
start :-
    spawn(pong, Pong_Pid, [
        node('http://remote.org'),
        src_predicates([pong/0])
    ]),
    spawn(ping(3, Pong_Pid), _).
```

Here is result of running the modified program:

```
?- start.
true.
Ping received pong
Ping received pong
Ping received pong
Ping finished
?-
```

So, why do we get the output of the "pinger" only, and not the output written by the "ponger"? The reason is that a shell can only receive and render output that is written from actors that

1) live on the same node as the top-level pengine to which the shell is attached, and 2) are descendants of this top-level pengine. (This restriction may be lifted in the future.)

Note that this program cannot be written in Erlang since Erlang is assumed to be operating in a trusted environment where we directly control the machines that we are operating.

In the following example we implement a so called *process ring*. A master process spawns each process and links to it. Then each process starts a loop. When the ring is completed the first process injects the message and terminates. `loop/1` blocks each process making it wait for a message to pass along.

```
start(NumberProcesses, Message) :-
    self(Self),
    create(NumberProcesses, Self, Message).

create(1, NextProcess, Message) :- !,
    NextProcess ! Message.
create(NumberProcesses, NextProcess, Message) :-
    spawn(loop(NextProcess), Prev, [
        link(true)
    ]),
    NumberProcesses1 is NumberProcesses - 1,
    create(NumberProcesses1, Prev, Message).

loop(NextProcess) :-
    receive({
        Msg ->
            NextProcess ! Msg
    }).
```

Inspired by (**?**), and making use of `rpc_synch/3` from Section 3.7), here is a pattern for *parallel* RPC:

```
parallel_rpc(Pids, Request, Responses) :-
    self(Self),
    maplist(par_rpc_aux(Self, Request), Pids, Refs),
    yield(Refs, Responses).

par_rpc_aux(Self, Request, Pid, Ref) :-
    uuid(Ref),
    spawn((rpc_synch(Pid, Request, Response), Self ! Ref-Response), _).

yield([], []).
yield([Ref|Refs], [Response|Responses]) :-
    receive({Ref-Response -> true}),
    yield(Refs, Responses).
```

[2]`http://logtalk.org/plstd/threads.pdf`

4.5 Comparing Web Prolog with ISO Prolog Threads

A number of Prolog systems (at least SWI-Prolog, YAP and XSB) implement the ISO Prolog working draft for threads.[2] As the following example shows, it implements aspects of what `self/1`, `spawn/2-3`, `!/2` and `receive/1-2` has to offer:

```
?- thread_self(Self),
   thread_create((parent(ida,Child),thread_send_message(Self,Child)),Pid,[]),
   thread_get_message(Who).
Self = main,
Pid = 2,
Who = valdemar.
?-
```

Concurrency and distribution is central to Web Prolog, and not something that could have been added later on.

Compared to the corresponding Web Prolog predicates, the `thread_*` predicates lack options and mechanisms in support of distribution. It would for example be easy to translate our locally running ping-pong example into code that uses the thread predicates rather than spawn, send and receive, but it is far from easy to translate the part that came next: running the two processes playing ping-pong on two different machines.

We see several advantages with Web Prolog compared to ISO Prolog and its POSIX-based concurrency model. First of all, concurrent programming with POSIX threads is ridiculously difficult. In particular, concurrency based on shared memory – encouraged by this model – is asking for trouble. We believe that most users will find that concurrent programming in Web Prolog is a lot easier to master. (A warning is in place though. Even with the help that languages such as Web Prolog and Erlang provides, concurrent programming is far from easy, and it is always difficult to know when the last bug has been removed.)

Secondly, and related to the first point, network transparency with respect to where a process is running, locally or remotely, is compatible with message-passing concurrency, but not with shared-state concurrency. Adding distributed message-passing to concurrent Erlang did not add any new concepts, and no new concept are added when we add distributed message-passing to concurrent Web Prolog. The passing of a URI in the `node` option is all that is required in order to move a local computation to a remote node.

Compared to the fairly low-level ISO Prolog Threads standard, the corresponding Web Prolog predicates are clearly more high-level. The ISO standard gives a developer more choice, but also more ways to shoot himself in the foot. For example, the ISO standard comes with predicates for creating *message queues* (a.k.a. channels), reading from and sending to them. In Web Prolog, the only message queue a process needs access to is the mailbox. As another example, the ISO standard provides *mutexes* as a way for developers to realise related updates to the Prolog dynamic database that can never lead to an inconsistent state. A Web Prolog actor (e.g. a pengine) has *its own* database that will never be shared with other actors, and therefore the problem will not arise in the first place. This is not to say that the standard is not useful. After all, we have been able to implement important aspects of Web Prolog on top of it!

4.6 Getting answers a tuple at a time

It is probably wise to entertain a suspicion of unexpected interactions between language features and possible impedance mismatches between the two paradigms – between Prolog's relational, non-deterministic programming model and Erlang's functional and message passing model. How well do the Erlang-ish constructs mix with Prolog, with backtracking for example?

In theory, we should be on the safe side. Sequential Erlang is basically Erlang with its data types, one-way pattern matching, functions and control structures, but without spawn, send, receive, and other constructs designed for concurrent programming. The idea behind Web Prolog can thus be described as an attempt to "plug out" the sequential part from Erlang and "plug in" sequential Prolog instead. There seems to be no principled reasons why we would not be able to replace the sequential functional language with a sequential relational language, e.g. a logic programming language such as Prolog. Indeed, Armstrong (2003a) writes:

> Erlang is a concurrent programming language with a functional core. By this we mean that the most important property of the language is that it is concurrent and that secondly, the sequential part of the language is a functional programming language.
>
> The sequential subset of the language expresses what happens from the point it time where a process receives a message to the point in time when it emits a message. From the point of view of an external observer two systems are indistinguishable if they obey the principle of observational equivalence. *From this point of view, it does not matter what family of programming language is used to perform sequential computation.* (Our emphasis.)

Alternatively – and this will be our choice – our approach will be described as an attempt to *extend* Prolog with constructs such as spawn, send and receive. We will keep everything that core Prolog has to offer, and extend it with a number of those primitives that make Erlang such a great language for programming message-based concurrency. The choice between extending Prolog with Erlang-ish constructs and extending Erlang with Prolog-ish constructs is easy to make, and a lot has to do with syntax. Provided we can accept using a syntax which is relational rather than functional, precluding the nesting of function calls, it should be clear by now that the surface syntax of Prolog can easily be adapted to express the needed Erlang-ish primitives ("adapted" is perhaps too much to say, since all we need is two infix operators that can be declared using `op/3`). It is arguably a lot harder to express Prolog rules and other constructs using the syntax of Erlang.

Let us now turn to an example which shows that the mix is both sound and easy to understand. Suppose the goal given in the argument to `spawn/2` has more than one answer, a goal such as `ancestor_descendant(mike, Who)` for example. We can call the goal, send the result back to the calling process, and then use `receive/1` in order to listen for a message of the form `next` or `stop` before terminating:

```
?- self(Self),
   spawn(( ancestor_descendant(mike, Who),
           Self ! Who,
            receive({
                next -> fail;
                stop ->
                    Self ! stopped
            })
         ), Pid).
Pid = 'a4b940a8-3b9a-11e6-9044-8fef75cbd8ac',
Self = '4806702e-384d-11e6-96f4-8fcbb6c814f5'.
?- flush.
Shell got tom
true.
?- $Pid ! next.
true.
?- flush.
Shell got sally
true.
?- $Pid ! stop.
true.
?- flush.
true.
?-
```

As the above session shows, the spawned goal generated the solution `tom`, sent it to the mailbox of the Prolog top-level and then suspended and waited for messages coming from the top-level process. When the message `next` arrived, the forced failure triggered backtracking which generated and sent `sally` to the mailbox of the top-level shell process. The next message was `stop`, so the spawned process terminated.

Again, it is easy to see that it is due to its semi-determinism that `receive/1` is doing an important job here. And again, reminding ourselves about the similarity with `read/1` and `thread_get_message/2-3`, we may conclude that a receive construct is not adding anything completely new and foreign to Prolog which could give us cause for concern about any fundamental incompatibility between the two paradigms.

This suggests that Prolog's backtracking mechanism is perfectly compatible with, and in fact complements, the proposed Erlang-like mechanism for spawning actors and handling the communication between them. We need to observe, however, that the goal to be solved in the above example is hard-coded into the program, and that the program handles neither failure of the spawned goal, nor exceptions thrown by it. There is clearly a need for something more generic. In Chapter 5 we will describe the API to a full-blown generic pengine abstraction, but first, in the next section, we provide an implementation which is considerably more generic than the one we have looked at so far.

4.7 A generic encapsulated search procedure

As we have seen, it is possible for a programmer to access and process different results of a non-deterministic computation from within a program. This is often referred to as *encapsulated search*. But for encapsulated search to become a key feature of the ACTOR profile of Web Prolog, we need to provide something more generic, and we also need to demonstrate that results may be computed and collected remotely as well.

Below, we show how we can build a generic predicate by using a meta-predicate (such as `call_cleanup/2`) and by specifying a small set of custom messages carrying answers and/or the state of the process that needs to be returned to the calling process. The predicate `call_cleanup/2` is here used not only to call a goal, but also to check if any choice points remain after the goal has been called or backtracked into.[3]

```
search(Query, Self, Parent) :-
    call_cleanup(Query, Det=true),
    (   var(Det)
    ->  Parent ! success(Self, Query, true),
        receive({
            next -> fail;
            stop ->
                Parent ! stop(Self)
        })
    ;   Parent ! success(Self, Query, false)
    ).
```

Our demonstration of how to access the generic procedure remotely will be a little bit of a show off, because we will not (this time) assume that the implementation is available at the remote node. Instead, by passing the `src_predicate` option, we send the locally defined predicate `search/3` along to the remote node for injection into the workspace of the process:

```
?- self(Self),
   spawn(search(ancestor_descendant(mike, Who), Pid, Self), Pid, [
       node('http://remote.org'),
       monitor(true),
       src_predicates([search/3])
   ]).
Self = '4806702e',
Pid = 'b0ffdb86'.
?- flush.
Shell got success('b0ffdb86',ancestor_descendant(mike, tom),true)
true.
?- $Pid ! next.
true.
?- flush.
Shell got success('b0ffdb86',ancestor_descendant(mike, sally),true)
true.
?- $Pid ! stop.
true.
?- flush.
Shell got stop('b0ffdb86')
Shell got down('b0ffdb86', true)
true.
?-
```

Note that the code in this example does not say anything about what should happen if an

[3]`http://www.swi-prolog.org/pldoc/man?predicate=call_cleanup/2`

exception is thrown, which would for example be the case if the predicate called by the goal is not defined. We do not catch such exceptions, and the omission of a catch is intentional. As an Erlang programmer would say, we do not have to program defensively, let it crash instead! Since the spawned process is monitored, the error message will eventually reach the mailbox of the spawning process anyway, in the form of a `down` message. The same is true of failure.

Clearly, a pengine is a kind of server (in the sense of Erlang – see Section 3.6). Also, note that the server, despite not explicitly threading any state, nor using the dynamic database, must still be considered stateful. Rather than using an explicit data structure for holding the state, it is the underlying Prolog process *as such* that holds it, most clearly shown in the way the pengine "remembers" its history and how its behaviour is influenced by this, enabling it to react appropriately when a client asks for the *next* solution to a query.

4.8 The actor API

Before ending this chapter, we document the actor predicate API.

Predicate: `self/1`

```
self(-Pid) is det.
```

Binds `Pid` to the process identifier of the calling process.

Predicate: `spawn/2-3`

```
spawn(+Goal, -Pid) is det.
spawn(+Goal, -Pid, +Options) is det.
```

Creates a new Web Prolog process running `Goal`. Valid options are:

- `node(+URI)`
  `URI` points to the Prolog Web node on which to create the process. Default is the current node `localnode`.
- `monitor(+Boolean)`
  Default is to not monitor.
- `link(+Boolean)`
  Default is to not link.
- `timeout(+IntegerOrFloat)`
  Terminates the spawned process (or the process of spawning a process) after `IntegerOrFloat` seconds.
- `src_list(+ListOfClauses)`
  Injects a list of Web Prolog clauses into the process.
- `src_text(+AtomOrString)`
  Injects the clauses specified by a source text into the process.

- `src_uri(+URI)`
  Injects the clauses specified in the source code located at `URI` into the process.
- `src_predicates(+List)`
  Injects the local predicates denoted by `List` into the process. `List` is a list of predicate indicators.

Predicate: `!/2`

```
+PidOrName ! +Message is det.
send(+PidOrName, +Message) is det.
```

Sends `Message` to the mailbox of the process identified as `PidOrName`. A message can be any ground Web Prolog term. The sending is asynchronous (fire-and-forget), i.e. `!/2` does not block waiting for a response but continues immediately. Also, `!/2` does not throw exceptions, so if a process named Pid does not exist, nothing happens.

Predicate: `receive/1-2`

```
receive(+Clauses) is semidet.
receive(+Clauses, :Options) is semidet.
```

`Clauses` is a sequence of receive clauses delimited by a semicolon:

```
{  Pattern1 [when Guard1] ->
         Body1 ;
   ...
   PatternN [when GuardN] ->
         BodyN
}
```

Each pattern in turn is matched against the first message (the one that has been waiting longest) in the mailbox. If a pattern matches and the corresponding guard succeeds, the matching message is removed from the mailbox and the corresponding body is evaluated. If the first message is not accepted, the second one will be tried, then the third, and so on. If none of the messages in the mailbox is accepted, the process will wait for new messages, checking them one at a time in the order they arrive. Messages in the mailbox that are not accepted are left in the mailbox without any change in their contents or order.
Valid options are:

- `timeout(+IntegerOrFloat)`
  If nothing appears in the current mailbox within `IntegerOrFloat` seconds, the predicate succeeds anyway. Default is no timeout.
- `on_timeout(+Goal)`
  If the timeout occurs, `Goal` is called.

Predicate: `exit/1-2`

```
exit(+Reason) is det.
exit(+PidOrName, +Reason) is det.
```

Executing `exit/1` terminates the current process. The predicate `exit/2` can be used to terminate any process, but only if you know its pid or registered name, and only if you own it.

Chapter 5

The pengine behaviour

In traditional Prolog the top-level is *lazy* in the sense that new solutions to a query are computed only upon the programmer's request. However, this top-level is inaccessible to programs, i.e. a program cannot *internally* create a top-level, pose queries and request solutions lazily. In Web Prolog, a pengine is a programming abstraction modelled on the interactive top-level of Prolog. A pengine is like a first-class interactive Prolog top-level, accessible from Web Prolog as well as from other programming languages.

Whereas encapsulated search (as implemented in Section 4.7) allows only one query per process, a pengine allows a full *session* per process created, possibly involving many queries. Indeed, we may want to refer to a pengine as an *encapsulated Prolog session*, an abstraction designed to make Prolog programmers feel right at home.

A pengine is an actor, a special *kind* of actor. Its behaviour differs from the behaviours of other kinds of actors in that it adheres to a communication protocol specific to pengines.

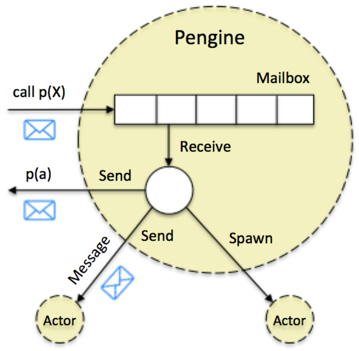


Figure 5.1: A pengine

The design behind pengines is inspired by the informal communication protocol that we as programmers adhere to when we invoke a Prolog shell from our OS prompt, load a program, submit a query, are presented with a solution (or a failure or an error), type a semi-colon in order to ask for more solutions, or hit return to stop. These are "conversational moves" that Prolog understands. But there are more such moves, since after having run one query to completion, the programmer can choose to submit another one, and so on. The session does not end until the programmer decides to terminate it.

Beginners interacting with a traditional Prolog shell quickly realises that not all actions and input are valid at all times. With experience however, as they respond to the affordances and adapt to the constraints that a Prolog shell embodies, the protocol becomes second nature to most people. To a large extent, they are guided by the prompts and messages returned by the system. A prompt such as `?-` informs them that they are expected to submit a query, a blinking cursor to the right of a printed solution (i.e. a set of variable bindings) is in fact also a prompt that tells them that they can either type a semicolon in order to ask for the next solution, or type return in order to complete the running of the query. (The lack of a response within a certain time limit is also a response, indicating a non-terminating query.) A prompt such as `|:` signals that Prolog is ready to read input from the shell, input expected to be a valid term terminated by a full stop.

There are only a few moves you can successfully make in each state, and the possibilities can easily be described, by a state machine for example, as we shall do in the next section.

5.1 The Pengine Communication Protocol (PCP)

The diagram in Figure 5.2 depicts the PCP, a protocol for the communication between a *client* and a *server* (in the Erlang sense of these terms). The server is a pengine. The client can be any kind of process (including another pengine) capable of sending the messages and signals in bold to the server. The server is responsible for returning the messages with a leading `/` back to the client.

We choose to model the protocol by means of a *statechart*. Since statecharts may be hierarchical, i.e. a state may contain another statechart down to an arbitrary depth, we are saved from having to add transitions for the event of abort or exit from every state where it is relevant. Suppose, for example, that the current state of the protocol is **s2**, and that the **abort** signal is received. The transitions leaving **s2** are tried from the inside and out, and since no label on a transition leaving **s2** matches, but one placed one step higher in the hierarchy does, a transition to state **s1** takes place.

As hinted at already, Web Prolog comes with built-in predicates which allows a client to spawn a local or remote pengine (`pengine_spawn/1-2`), and to send the messages in bold to it (`pengine_ask/2-3`, `pengine_next/1-2`, `pengine_stop/1`, `pengine_respond/2`, `pengine_abort/1` and `pengine_exit/1`). It also comes with web APIs allowing a non-Prolog process to achieve the same thing.

A process is able to *spawn* a pengine. After having done so, the process becomes the client of the pengine and can start communicating with it according to the protocol. Initially, the protocol is in state **s1**, where the pengine rests idle, waiting for an **ask** message containing a query. When such a message arrives, the protocol transitions to state **s2**, where the pengine is actually doing work. The protocol will remain in state **s2** until some work is done and the pengine sends a message indicating either `/success`, `/failure`, `/error`, `/prompt`, `/output`, `/abort` or (if the process is monitored) `/down` to the client. On the event of the pengine sending `/failure` or `/error`, the protocol will transition back to state **s1**. This will

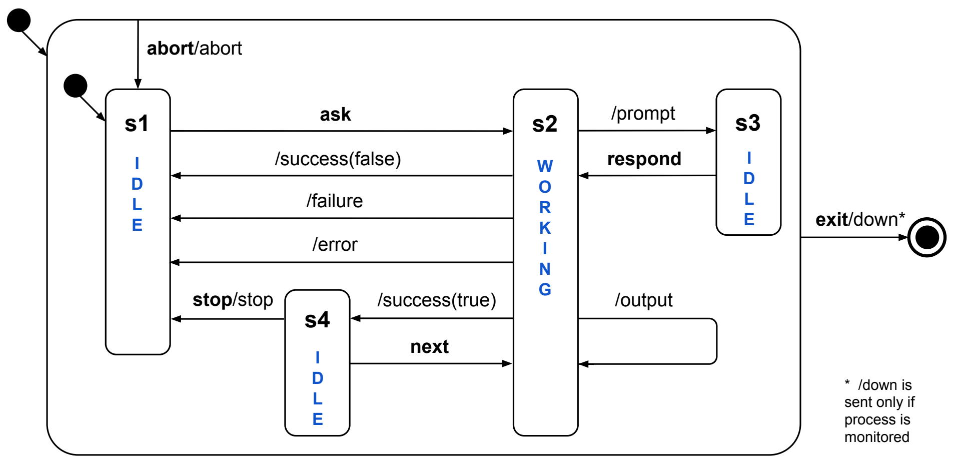


Figure 5.2: Statechart depicting the Pengine Communication Protocol (PCP) for a complete Web Prolog session. The transitions are labeled with *message types*. Types in bold are sent from the client to the pengine, whereas message types with a leading / goes in the opposite direction, from the pengine to the client.

be the case also for a `/success` message that indicates that no more solutions to the query can be found (marked with false in the chart). However, if a `/success` message indicates that more solutions may exist (marked with true in the chart), the protocol transitions to state **s4**, where it will wait for a message **next**, **stop**, **ask**, **abort** or **exit** to arrive from the client. If the message is **stop** or **abort** the protocol will transition back to state **s1**, if it is **next** it will transition to state **s2** and trigger the search for more solutions, and finally, if it is **exit**, it will (if the process is monitored) force the pengine to send a message `/down` back to the client and then to terminate.

When the protocol is in state **s2** the pengine may send `/prompt` messages to the client. The client can respond to them in three ways: by sending an **response** message, by sending **abort**, or by sending **exit**. Sending **response** allows the pengine to carry on the current computation informed by the input provided by the client. As before, the other messages will either abort the current computation and return to state **s1**, or terminate the pengine.

5.2 Some properties of the PCP protocol

As is often pointed out by Joe Armstrong, message-based systems give us a whole new level of abstraction. In his words, “We have the black boxes that do things, and we have the messages in between. What goes on inside the black boxes does not really matter; what takes place there is abstracted over.” This is just as true for pengines and multi-pengine systems as it is for other actors and actor systems. As long as a process delivers the correct answers in response to Prolog queries over a Prolog program, it can, for all intents and purposes, be counted as a pengine.

The protocol is *abstract* – an abstract, declarative specification of a desired communicative behavior. It abstracts away from *location* (i.e. does away with the local/remote distinction). It is equally valid for an in-memory context that allows for a fine-grained, low-latency interaction between an actor and a pengine running on the same node, as it is valid for the interaction between a client (an actor or another kind of process) and a pengine running on a remote node. The protocol is abstract also in the sense that it has nothing to do with Horn clause logic, or SLD resolution, or unification, or top-down, depth-first search. The statechart does not tell us to force backtracking in order to generate the next solution, nor does it suggest that the way to do this is to fail.

Another observation is that the protocol is *complete* in that it covers the whole life cycle of an interactive Prolog session. It even goes a bit further than that, in that it supports asking for more than one solution at a time by means of the `limit` option. This can be seen as an aspect of the pragmatics of the communication.

Thirdly, the protocol will probably remind many readers about Byrd's *box model* of Prolog execution (Byrd, 1980). They are surely related, but while the box model concentrates on the ins and outs of the procedural calls behind SLD resolution, the PCP protocol focuses on communication. The similarity probably just shows that reasoning and communication is a very thightly knit pair in Web Prolog, and that pengines are indeed capable of both tasks. Logic and protocol form two separate aspects of Prolog, and a pengine is both a reasoning engine *and* an interaction engine.

5.3 The pengine predicate API

As we saw with the fridge simulation example in Section 3.6 we should aim to equip programmers with a proper predicate API, as abstract as possible, rather than to force them to operate directly on the protocol using send and receive. This is even more important when dealing with a protocol as complex as the PCP protocol.

In the present report we deal with two kinds of predicate APIs related to communication with and among pengines: an asynchronous API where predicates are deterministic, and a synchronous API which comprises just one non-deterministic predicate. Below, we present the documentation of the asynchronous API. We will get back to the synchronous API in Section 7.12.

Predicate: `pengine_spawn/1-2`

```
pengine_spawn(-Pid) is det
pengine_spawn(-Pid, +Options) is det
```

The predicate `pengine_spawn/1-2` inherits all options from `spawn/3` and adds a new one as well. Options inherited from `spawn/3` means that code may be injected into pengine processes, just like it may be for spawned processes. The option that is added is:

- `exit(+Boolean)`
  Determines if the pengine session must exit after having run a goal to completion. Defaults to true.

Predicate: `pengine_ask/2-3`

```
pengine_ask(+Pid, +Goal) is det.
pengine_ask(+Pid, +Goal, +Options) is det
```

Calls pengine `Pid` with the goal `Goal`. Valid options are:

- `template(+Template)`
  `Template` is a variable (or a term containing variables) shared with the query. By default, the template is identical to the goal.

- `limit(+Integer)`
  Retrieve solutions in lists of length `Integer` rather than one by one. A value of 1 means a unary list (default). Other integers indicate the maximum number of solutions to retrieve in one batch.

`pengine_ask/2-3` is deterministic, even for queries that have more than one solution. Variables in `Goal` will not be bound. Instead, results and other kinds of output will be returned in the form of messages delivered to the mailbox of the process that called `pengine_spawn/2-3`.

- `success(Pid, Terms, More)`
  `Pid` refers to the pengine that succeeded in solving the query. `Terms` is a list holding instantiations of `Template`. `More` is either `true` or `false`, indicating whether or not we can expect the pengine to be able to return more solutions, would we call `pengine_next/1-2`.[1]

- `failure(Pid)`
  `Pid` is the pid of the pengine that failed for lack of (more) solutions.

- `error(Pid, Term)`
  `Pid` is the pid of the pengine throwing the exception. `Term` is the exception's error term.

- `output(Pid, Term)`
  `Pid` is the pid of a pengine running the goal that called `pengine_output/1`. Term is the term that was passed in the argument of `pengine_output/1` when it was called.

- `prompt(Pid, Term)`
  `Pid` is the pid of the pengine that called `pengine_input/2` and `Term` is the prompt.

- `down(Pid, Term)`
  `Pid` is the pid of the pengine that terminated and `Term` is the reason.

Predicate: `pengine_next/1-2`

```
pengine_next(+Pid) is det.
pengine_next(+Pid, +Options) is det
```

Asks pengine `Pid` for the next solution to `Goal`. The only valid option is:

[1] We are considering adding a fourth argument `Info`, where `Info` is a structure containing extra information such

- `limit(+Integer)`
  Retrieve solutions in a list of length `Integer` rather than one by one. 1 means no limiting (default). Other positive integers indicate the maximum number of solutions to retrieve at once.

The messages delivered to the mailbox of the process that called `pengine_next/1-2` are the same as for `pengine_ask/2-3`.

Predicate: `pengine_stop/1`

```
pengine_stop(+Pid) is det.
```

Asks pengine `Pid` to stop. If successful, delivers a message `stop(Pid)` to the mailbox of the process that called `pengine_spawn/2-3`.

Predicate: `pengine_abort/1`

```
pengine_abort(+Pid) is det.
```

Tells pengine `Pid` to abort any goal that it currently runs. If successful, delivers a message `abort(Pid)` to the mailbox of the process that called `pengine_spawn/2-3`.

Predicate: `pengine_output/1`

```
pengine_output(+Term) is det.
```

Sends Term to the parent process.

Predicate: `pengine_input/2`

```
pengine_input(+Prompt, -Term) is det.
```

Sends `Prompt` to the parent process and waits for input. `Prompt` may be any term, compound as well as atomic.

Predicate: `pengine_respond/2`

```
pengine_respond(+Pid, +Input) is det.
```

Sends a response in the form of the term `Input` to a process that has prompted its parent process for input.

Predicate: `pengine_exit/1-2`

```
pengine_exit(+Pid) is det.
pengine_exit(+Pid, +Reason) is det.
```

Same as `exit/1` and `exit/2`.

as timing information.

5.4 Creating and querying a pengine

We create a remote pengine and call it with a goal as follows:

```
?- pengine_spawn(Pid, [
        node('http://remote.org')
   ]),
   pengine_ask(Pid, between(0,15,N), [
        template(N),
        limit(5)
   ]).
Pid = '773e4d16-f1c6-11e6-9a48-0b08209e091b'.
?- flush.
Shell got success('773e4d16-f1c6-11e6-9a48-0b08209e091b',[0,1,2,3,4],true)
true.
?-
```

Similar to pengine_ask/2-3, pengine_next/1-2 also accepts a limit option. Below we use it to get ten more solutions, instead of the five solutions we would have received if we did not pass the limit option:

```
?- pengine_next($Pid,[limit(10)]).
true.
?- receive({
        success(_, Solutions, _) -> writeln(Solutions)
   }).
[5,6,7,8,9,10,11,12,13,14]
true.
?-
```

Note that we here, for a change, used receive/1-2 instead of flush/0 in order to check the mailbox for messages.

5.5 I/O and pengines

Referring back to Section 4.1 where we compared writeln/1 and pengine_output/1, this is how I/O works in a pengine:

```
?- pengine_spawn(Pid, [
        node('http://remote.org')
   ]).
Pid = '18f0b35c-ef82-11e6-8f73-1b4ebfe88deb'.
?- pengine_ask($Pid, writeln(hello)).
hello
true.
?- pengine_ask($Pid, pengine_output(hello)).
true.
?- flush.
Shell got output('18f0b35c-ef82-11e6-8f73-1b4ebfe88deb', hello)
```

```
true.
?-
```

5.6 Pengines and the message deferring mechanism

One consequence of the message deferring mechanism (which relies on the selected receive – see Section 3.1) is that messages, to some extent, are allowed to arrive to a mailbox in the "wrong" order. In the following example, a pengine is spawned, and then `pengine_next/1` is called. The protocol is obviously not in a state where it can react on the `next` message. The message is therefore deferred and it is not until `pengine_ask/3` is called and the protocol changes states that the message `next` has an effect.

```
?- pengine_spawn(Pid).
Pid = 'e3gf4b66-1934-3ee6-81c8-d33f5336c91f'.
?- pengine_next($Pid).
?- flush.
true.
?- pengine_ask($Pid, member(X,[a,b,c]), [template(X)]).
true.
?- flush.
Shell got success('e3gf4b66-1934-3ee6-81c8-d33f5336c91f', [a], true)
Shell got success('e3gf4b66-1934-3ee6-81c8-d33f5336c91f', [b], true)
true.
?-
```

One way to think of this is in terms of the so called *robustness principle*: "Be conservative in what you send, be liberal in what you accept".[2] Due to the deferring behaviour a pengine is liberal in this way but, as implied by the principle, clients are advised not to rely on this behaviour.

5.7 Emulating once/1 and findall/3

In this section we show two examples inspired by (Tarau and Majumdar, 2009). Equipped with `pengine_spawn/2-3` we can implement a variant of `once/1` as follows:

```
once(Goal) :-
    pengine_spawn(Pid),
    pengine_ask(Pid, Goal),
    collect_answers(Pid, Goal).

collect_answers(Pid, Goal) :-
    receive({
        success(Pid, [Goal], _) ->
            pengine_stop(Pid),
            collect_answers(Pid, Goal);
        stop(Pid) ->
```

[2] https://en.wikipedia.org/wiki/Robustness_principle

```
            true;
        failure(Pid) ->
            fail;
        error(Pid, Error) ->
            throw(Error)
    }).
```

In the second example we implement a variant of the `findall/3` predicate like so:

```
findall(Template, Goal, Solutions) :-
    pengine_spawn(Pid),
    pengine_ask(Pid, Goal, [
        template(Template)
    ]),
    collect_answers2(Pid, Solutions).

collect_answers2(Pid, Solutions) :-
    receive({
        success(Pid, [Solution], true) ->
            Solutions = [Solution|Rest],
            pengine_next(Pid),
            collect_answers2(Pid, Rest);
        success(Pid, [Solution], false) ->
            Solutions = [Solution];
        failure(Pid) ->
            Solutions = [];
        error(Pid, Error) ->
            throw(Error)
    }).
```

Admittedly, these programs may be somewhat less elegant, as well as less efficient, than the programs suggested by (Tarau and Majumdar, 2009). However, from a practical point of view, we would argue that our programs are more reasonable, simply due to their closeness to Erlang.

5.8 A tiny expert system example

There was a time when expert systems was a big thing. There is even a Prolog text book that focuses on this (Merritt, 1989). But expert systems never really went anywhere, so expert systems in Prolog, using the quite elegant meta-interpreter approach, never really went anywhere either. Nevertheless, the approach is worth presenting as an example. Here is a meta-interpreter implementing a tiny expert-system featuring a query-the-user facility which in interaction with the system allows a user to determine if Tweety is a good pet or not:

```
prove(true) :- !.
prove((B, Bs)) :- !,
    prove(B),
    prove(Bs).
prove(H) :-
```

```
    clause(H, B),
    prove(B).
prove(H) :-
    askable(H),
    format(atom(Q), 'Is it true that: ~q?', [H]),
    pengine_input(Q, Answer),
    Answer == yes.
```

And here is a simple rule system:

```
good_pet(X) :- bird(X), small(X).
good_pet(X) :- cuddly(X), yellow(X).

bird(X) :- has_feathers(X), tweets(X).

yellow(tweety).

askable(tweets(_)).
askable(small(_)).
askable(cuddly(_)).
askable(has_feathers(_)).
```

The system is run like so:

```
?- prove(good_pet(tweety)).
```

A user in interaction with the system will then have to answer "yes" or "no" to the questions being asked and the system's verdict will be either `true` (meaning that yes, Tweety is a good pet) or `false` (meaning no, Tweety is not a good pet).

This is a nice example of how cooperation (between user and system) is made possible by communication. It is also an example of a quite useful kind of program that would be much more difficult to write if only synchronous and stateless interaction over HTTP could be supported.

5.9 Echo example using the predicate API

```
?- pengine_spawn(Pid, [
      node('http://remote.org'),
      src_list([(echo :- pengine_input(?,M), pengine_output(M), echo)])
   ]),
   pengine_ask(Pid, echo),
   loop(Pid).

loop(Pid) :-
    receive({
        prompt(Pid, Prompt) ->
            write(Prompt),
            read(Response),
            pengine_respond(Pid, Response);
        output(Pid, Term) ->
```

```
            writeln(Term)
    }),
    loop(Pid).
```

5.10 Concurrent programming with pengines

[TODO: To be convincing, we need to show that concurrent non-deterministic programming is feasible. Our approach should perhaps be to reimplement a couple of the concurrency predicates written in SWI-Prolog?]

Chapter 6

Two kinds of web APIs

As stated already in Chapter 1, for maximal flexibility some nodes on the Prolog Web offer two kinds of APIs: 1) an asynchronous and stateful WebSocket API, and 2) a synchronous and stateless HTTP API. In this chapter we will take a closer look at them, describe how they differ, and demonstrate how they can be used.

The web APIs are not an afterthought. They are extremely important. Indeed, if we did not use HTTP or WebSocket, we would not be on the Web. We use both, and the whole distribution layer of Web Prolog and the Prolog Web depends on them, for openness, as well as for security. The web APIs must be capable of everything that the predicate APIs are capable of, and vice versa. Otherwise they would not be able to support the distributed programming model on which the Prolog Web is based.

6.1 The WebSocket API

In order to enable a client to control all aspects of a set of pengines and other actors, an ACTOR node offers a WebSocket sub-protocol. WebSocket is a real-time, low latency, bi-directional protocol for asynchronous communication between a client and a server. WebSockets differs from TCP in that it enables a stream of messages instead of a stream of bytes. With WebSockets, data is always sent as a whole message. The receipt of a message is event driven and the data payload in an event is always the entire message that the other side sent. Under the hood, WebSockets is built on top of normal TCP sockets.

For Web Prolog, all of this works to our advantage since we specifically want to use the message passing type of paradigm that WebSockets offers. Thus, being already message-based, WebSockets save us from most of the work involved in implementing (for example) the PCP protocol.

By their nature of being initiated via an HTTP connection which is then switched over to the WebSocket protocol, WebSockets can operate on the same port as an HTTP server. There is a vast infrastructure for HTTP and HTTPS that already exists (proxies, firewalls, caches, and other intermediaries), ensuring that security requirements of the modern web are fulfilled and the browser and the node can validate each other.

In addition, we want web browsers to be able to connect to our nodes, and browsers do not support plain TCP, only HTTP requests and WebSocket connections. Also, other types of clients can easily use a WebSocket connection, arguably more easily than they would be able to use some custom protocol built on top of TCP.

WebSocket client libraries are available for the most common general programming languages, SWI-Prolog included. In all the major web browsers, the WebSocket object provides a JavaScript API for creating and managing a WebSocket connection to a server, as well as for sending and receiving data on the connection.[1] The role of JavaScript here is to construct a new websocket, to define the necessary handlers for `onopen`, `onmessage`, `onerror` and `onclose` messages, and to call methods for sending messages or closing the connection. A synopsis that leaves out a lot of details can be given as follows:[2]

Constructor

```
var ws = new WebSocket(<URI>[,<protocol>);
```

Event listeners

```
ws.onerror = function(message) {...}

ws.onopen = function(message) {...}

ws.onmessage = function(message) {...}

ws.onclose = function(message) {...}
```

Methods

```
ws.send(message)

ws.close()
```

The messages that are used to spawn pengines or other actors as well as messages that can be used to control them are stringified JSON. Given a connection, and somewhat schematically, spawning a pengine is done like so, where the options part is optional:

```
connection.send(JSON.stringify({
  command: 'pengine_spawn',
  options: <options>
}));
```

By design, the messages understood by the PCP sub-protocol matches the `pengine_*` predicates quite well. Subject to options, answers and other kinds of messages arriving from the node are returned in the form of JSON or Prolog text. Client code written in JavaScript would normally request that they be encoded in JSON. The pid of the created pengine is returned to the client, not in the form of the binding of a variable (as in the predicate API), but in the form of a message {`"type":"spawned","pid":<pid>`}.

[1]See `https://www.w3.org/TR/websockets/`

[2]For all the gory details as well as a good general introduction to WebSocket we recommend (Lombardi, 2015).

Given the pid, we can ask a query by sending a message which exactly reflects the `pengine_ask/2-3` predicate in the predicate API:

```
connection.send(JSON.stringify({
  command: 'pengine_ask',
  pid: <pid>
  query: <query>,
  options: <options>
}));
```

The full set of options that is valid in the predicate API is valid in the web API as well. Since the options for `pengine_spawn/2`, `pengine_ask/3` and `pengine_next/2` are documented in Section 5.3, we do not repeat the details here. A brief reminder should be sufficient. In the message corresponding to `pengine_spawn/2` we can use options such as `exit`, `monitor`, `link`, `timeout`, `src_list`, `src_text`, `src_uri` and `src_predicates`. (Recall that the majority of these options are inherited from `spawn/3`.) For `pengine_ask/3` we can use `template`, `offset` and `limit`. For `pengine_next/2` there is only one valid option, namely `limit`.

6.2 Two comprehensive WebSocket examples

In this section we give two examples of how the WebSocket API can be used. They are both written in a combination of HTML and JavaScript. For both examples, the HTML can be given as follows, where the JavaScript to be run should be linked-in as on line 6:

```
 1  <!DOCTYPE html>
 2  <html lang="en">
 3    <head>
 4      <meta charset="utf-8"/>
 5      <title>WS example</title>
 6      <script src="/example.js"></script>
 7    </head>
 8    <body>
 9      <div id="output"></div>
10    </body>
11  </html>
```

Upon page load, the first application will create a WebSocket connection to the node, spawn a pengine there, and (when having received a pid) ask the query `ancestor_descendant(mike, Who)`. Here is the complete JavaScript source code for this application:

```
var ws = new WebSocket('ws://localhost:3060/ws','pcp-0.2');
ws.onopen = function (message) {
  ws.send(JSON.stringify({
    command: 'pengine_spawn',
    options: '[format(json)]'
  }));
};
ws.onmessage = function (message) {
  var event = JSON.parse(message.data);
10      if (event.type == 'spawned') {
11        ws.send(JSON.stringify({
12          command: 'pengine_ask',
13          pid: event.pid,
14          query: 'ancestor_descendant(mike, Who)'
15        }));
16      } else if (event.type == 'success') {
17        document.getElementById("output").innerHTML +=
18            JSON.stringify(event.data) + "<br/>";
19        if (event.more) {
20          ws.send(JSON.stringify({
21            command: 'pengine_next',
22            pid: event.pid
23          }));
24        }
25      }
26    };
```

On line 1 we create a new WebSocket connection, passing the URI of the server as well as a string indicating the name of the sub-protocol. The `onopen` handler, defined on line 2-7, is triggered when the connection is opened. From within the handler, a request for the creation of a pengine is made, where the remote node is told to respond with messages encoded as JSON.

Messages arriving from the node trigger the onmessage handler defined on line 8-26. Messages are JavaScript objects with a number of properties associating names with values. Here, we are mostly interested in the `data` property of the messages, the value of which is always a string.[3] In our case, since we have requested responses in the JSON format, the string will look something like this:

```
{ "type":"spawned",
  "pid":"4ebe2da3-5d1c-43fe-a90d-c30d2db50235"
}
```

On line 9 we parse the string into a JavaScript object before doing anything else. The object is stored in the variable `event`.

On line 11-15, we start talking to our newly created pengine. We send it a request to solve the goal `ancestor_descendant(mike,Who)`. The result arrives in the form of a event of type `success`.

```
{ "type":"success",
  "pid":"4ebe2da3-5d1c-43fe-a90d-c30d2db50235",
  "data":[{"Who":"tom"}],
  "more":true
}
```

On line 17-18, we log the event data, and if the value of `event.more` is `true`, indicating that there may be more solutions to the query, we submit a request for the next solution.

When this code runs in a browser, the answer bindings will be shown on the web page,

[3]According to the standard, binary is also possible, but this is not something that we try to take advantage of at this point.

like so:

```
[{"Who":"tom"}]
[{"Who":"sally"}]
[{"Who":"erica"}]
```

This example was very simple. We created a connection, we spawned just *one* remote pengine process, we used it to request all solutions, *one* at a time, to just *one* query, and then the process terminated. But bear in mind that as soon as we have created a WebSocket connection, it can be used to spawn and communicate with *more* than one remote process over the same connection. (This can be seen as a form of multiplexing.) Also, we often want to use a spawned pengine for solving more than one query – first one, then another one, and so on – especially if the workspace of the pengine is updated in between queries, but also for performance reason since reusing a pengine is always cheaper than spawning a new one. Finally, instead of requesting solutions one by one, we can add a property `options` with the value of (say) `[limit(10)]` to the `pengine_ask` message on line 11-15. There are only three solutions so this would result in the following answer bindings:

```
[{"Who":"tom"},{"Who":"sally"},{"Who":"erica"}]
```

Our next example demonstrates I/O. It uses the `src_text` option of `pengine_spawn/1` to inject a simple echo program into a pengine when it is spawned. When run in a browser, the JavaScript program, in interaction with the pengine, will open a widget asking for input, send it to the pengine which will in turn echo it to the client and then ask for more input:

```
var ws = new WebSocket('ws://localhost:3060/ws','pcp-0.2');
var program =
   ‘echo :-
        pengine_input('Input a term!', Something),
        (   Something == null
        ->  true
        ;   pengine_output(Something),
            echo
        ).‘;
ws.onopen = function (message) {
  ws.send(JSON.stringify({
    command: 'pengine_spawn',
    options: '[src_text("' + program + '")]'
  }));
};
ws.onmessage = function (message) {
  var event = JSON.parse(message.data);
  if (event.type == 'spawned') {
    ws.send(JSON.stringify({
      command: 'pengine_ask',
      pid: event.pid,
      query: 'echo'
    }));
  } else if (event.type == 'prompt') {
    var response = prompt(event.data);
    ws.send(JSON.stringify({
```

```
27          command: 'pengine_respond',
28          pid: event.pid,
29          term: response
30        }));
31      } else if (event.type == 'output') {
32        document.getElementById("output").innerHTML +=
33            JSON.stringify(event.data) + "<br/>";
34      }
35    };
```

A potential drawback of the proposed API is that it forces the author of a JavaScript client to engage in a lot of ugly and error prone string hacking, building syntactically correct Prolog queries by concatenating pieces of strings together. The obvious answer to this problem is for the programmer to write, or for the node to offer, a client-side library dedicated to this task. (Such wrapper libraries can of course also be written in and for languages other than JavaScript.) Better perhaps, is that recent versions of JavaScript supports both string interpolation features and multi-line strings, which should make such code both easier to write and clearer to read.[4] Note that lines 3-9 exemplifies using a multi-line string, delimited by backticks.

6.3 The RESTful HTTP API

HTTP is the oldest transport protocol for the Web and was, during many years, the only one that counted. It is designed as a simple request-response protocol between a web client and a web server. The client is always initiating the exchange by making a request, whereas the server is responsible for the response. HTTP 1.X has a reputation for wasting space and being slow, but HTTP 2.0 seems to bring improvement that promise to make HTTP both terser and faster.

In contrast to WebSockets, HTTP is a stateless protocol, meaning that each request message must be understood in isolation. This means that every request needs to bring with it as much detail as the server needs to serve that request, without the server having to store a lot of information from previous requests. The `query` parameter is the only mandatory parameter:

```
GET <URI>/ask?query=<Query>
```

The other parameters accepted are `template` (default is the variable in the query), `offset` (default is `0`), `limit` (default is `1`), `timeout` (default is `inf`) and `format` (default is `json`).

Here is how we ask for the first solution to a query:

```
GET http://remote.org/ask?query=ancestor_descendant(mike,Who)&offset=0
```

As before, solutions are (by default) given in the form of Prolog variable bindings, encoded as JSON. In this case, there is only one binding, of `Who` to the atom `tom`:

```
{ "type":"success",
  "pid":"anonymous",
  "data":[{"Who":"tom"}],
```

[4]`https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Template_literals`

```
  "more":true
}
```

The value `true` here indicates that there may be more solutions to the query. To ask for the next solution, we can make a new GET request, setting `offset` to `1` this time:

```
GET http://remote.org/ask?query=ancestor_descendant(mike,Who)&offset=1
{ "type":"success",
  "pid":"anonymous",
  "data":[{"Who":"sally"}],
  "more":false
}
```

Sure enough, the node responded with the next solution. The value `false` shows that there are no other solutions to be found. If we insist anyway, by setting `offset` to `2`, we would receive a response of the form {`"type":"failure", "pid":"anonymous"`}.

The web API allows us to add a parameter `limit` to the request. As before, this saves us some network roundtrips and allows us to do "pagination" of solutions. Here we show two requests and their responses. The first request asks for the first two ways to split a list, and the second request asks for the remaining ways. We leave out the `offset` parameter for the first request, since its default value is `0`, but for the second request we set it to `2`:

```
GET http://remote.org/ask?query=append(Xs,Ys,[a,b,c])&limit=2
{ "type":"success",
  "pid":"anonymous",
  "data":[{"Xs":[], "Ys":["a","b","c"]},{"Xs":["a"], "Ys":["b","c"]}],
  "more":true
}

GET http://remote.org/ask?query=append(Xs,Ys,[a,b,c])&offset=2&limit=2
{ "type":"success",
  "pid":"anonymous",
  "data":[{"Xs":["a","b"], "Ys":["c"]},{"Xs":["a","b","c"], "Ys":[]}],
  "more":false
}
```

In addition to the other query parameters, a node may also accept `src_text`. To allow for large values, a node should also offer POST in addition to GET. A POST request such as

```
POST http://isotope.org/ask
query=p(X)
offset=1
limit=2
src_text=p(a). p(b). p(c).
```

would produce the following JSON formatted response:

```
{
  "type":"success",
  "pid":"anonymous",
  "data":[{"X":"b"},{"X":"c"}],
  "more":false
}
```

6.4 A comprehensive RESTful HTTP API example

In this section we show how to program the same example as in Section 6.2 using the RESTful API instead. To keep the example reasonably short, we are using jQuery, a JavaScript library which allows developers to invoke XMLHttpRequest functionality without coding directly to the XMLHttpRequest API.

```
$.ajax({
  url:"http://remote.org/ask?query=ancestor_descendant(mike,Who)",
  success: function(event) {console.log(JSON.stringify(event.data))};
});
$.ajax({
  url:"http://remote.org/ask?query=ancestor_descendant(mike,Who)&offset=1",
  success: function(event) {console.log(JSON.stringify(event.data))};
});
$.ajax({
  url:"http://remote.org/ask?query=ancestor_descendant(mike,Who)&offset=2",
  success: function(event) {console.log(JSON.stringify(event.data))};
});
```

Note that the only difference between the three requests is the value of the `offset` parameter. The web page would show us the same result as before:

```
[{"Who":"tom"}]
[{"Who":"sally"}]
[{"Who":"erica"}]
```

Again, it is important to understand that when our client relies on the work of remote anonymous pengines there are certain things we must not expect them to be able to do. We cannot make them work with code that we supply to it by using `consult/1` or `assert/1`. And I/O is not supported so the program that we are calling must not produce any output apart from answers of the form success, failure or error. If it does, we simply will not receive it.

6.5 An efficient implementation of the RESTful HTTP API

A naive implementation of the RESTful HTTP ask API is easy to build. For a request of the form `<BaseURI>/ask?query=<Q>&offset=<N>&limit=<L>` the node needs to compute the slice of solutions `0..N+L`, drop the solutions `0..N` and respond with the rest of the slice. Even a CGIs script could do this, but this would be slow, not only because CGI by its nature is slow, but also since such a naive approach in many cases involves a lot of recomputation. Computing the first slice (i.e. the one starting at offset 0) is as fast as it can be, but computing the second slice involves the recomputation of the first slice and, more generally, computing the nth slice involves the recomputation of all preceding slices, the results of which are then just thrown away. This, of course, is a waste of resources and puts an unnecessary burden on the node.

To achieve a more efficient and less wasteful querying, recomputation must be avoided. This should remind us about concepts such as *caching* and *memoisation*. We shall describe an approach which involves pengines and where the actor manager (subject to a setting) may cache the state of the pengine that produced the nth slice of solutions to a query, so that the work spent on producing it will not have to be repeated. This can be done without requiring that the actor manager remembers *which* client made the request for the previous slices of solutions. Indexing the relevant pengine on the combination of the query `Q` and an integer `N` indicating the next state is sufficient, and this is the key to the RESTfulness of our approach.[5]

The method can also be seen as a kind of *pooling* of pengines, but while pooling usually involves a pool of merely initialised processes, the method described here involves a pool where each member has already done some real work. In other words, the idea here is not to cache *already computed* solutions but rather to cache the *potential* for new solutions in the form of pengine processes that have "more to give". The method would probably be compatible with ways of caching already computed solutions, but a further discussion of such possibilities would take us outside the scope of the report.

A consequence of our approach is that it allows the computation of the full set of solutions to a query to be distributed over more than one pengine. We need to avoid spawning a new pengine for each incoming request, but instead, when available, select a member from a pool of suspended pengines which, since it has already performed some of the work, needs to do as little as possible in order to compute the requested solutions. Using this approach, it is likely (but not guaranteed) that the work that generated the nth slice of solutions does not have to be repeated if a request for the next slice comes in.

The way to realise this is to make the actor manager responsible for the maintenance of a cache consisting of entries pointing to members of the pool of suspended pengines. The signature of a cache entry can be given as follows:

```
cache(+QID, +N, -Pid) is det.
```

Here, `QID` is an identifier representing a query `Q`, `N` is an integer > 0, and `Pid` is the pid of an already spawned pengine which, after having computed `N` solutions to the query `Q`, is now suspended. A cache is simply a dynamic predicate comprising an ordered sequence of `cache/3` clauses.[6] The cache will be searched from the top and updates will be added to the bottom.

For the sake of an illustrative concrete example, suppose that the cache for a node `http://remote.org` contains the entries represented by the following clauses:[7]

```
cache(between(1,12,_), 5,  '1f821ea6').
cache(between(1,12,_), 5,  '6a912be5').
cache(p(_, _),         50, '7f2111a1').
```

Each cache entry points to a pengine which is currently resting in its suspended state. The pengines with the pids `'1f821ea6'` and `'6a912be5'` have both been used to produce the first five solutions to the query `?-between(1,12,N)`, and the pengine named '7f2111a1' has been used to compute fifty solutions to the query `?-p(X,Y)`. Again, note that the actor manager does *not* attempt to keep track of *which* clients made the requests for the solutions to these queries.

[5]The initial idea is due to Jan Wielemaker – bugs and inconsistencies should be blamed on me.

[6]The implementation of the cache as a Prolog predicate is not mandated. A node would be free to implement it in a way that suits the host platform best.

[7]We simplify things here. In an actual implementation `QID` will have to consists of more than just a query. The template as well as (a hash representing) source code to be injected (if any) must also be part of the query identifier.

Note also that the cache does not represent the *full* history of the traffic between the node and its clients. In the actual history there may have been many requests that were successfully responded to in full, but which did not leave any entries in the cache since the queries handled by the pengines involved were all run to completion.

Let us now look at what happens when a request such as the following is made to the node:

```
http://remote.org/ask?query=between(1,12,N)&offset=5&limit=5
```

The actor manager inspects the request and determines that since the very first entry in the cache points to a pengine which will be able to generate the requested slice of solutions, there is no need to spawn a new pengine. By using `'1f821ea6'`, the next batch of solutions may be computed.

```
{
  "data": [ {"N":6},  {"N":7},  {"N":8},  {"N":9},  {"N":10} ],
  "more":true,
  "pid":"anonymous",
  "type":"success"
}
```

Note the value `anonymous` of the `pid` property. Revealing the pid of the pengine that computed the solutions would perhaps not hurt, but would be potentially misleading since the pengine cannot be used outside the caching scheme.

Since those five solutions are (as indicated by the `more` property) not the only ones remaining, the pengine will continue to exist in a suspended state, and a new cache entry will be created which relates the combination of the query identifier and the pid to the offset of the next (not yet computed) slice of solutions. The old cache entry will be removed and the new entry will be added as the last clause. The updated cache looks like follows:

```
cache(between(1,12,_), 5,  '6a912be5').
cache(p(_, _),         50, '7f2111a1').
cache(between(1,12,_), 10, '1f821ea6').
```

Now, let us see what happens when a request for the remaining solutions is made:

```
http://remote.org/ask?query=between(1,12,N)&offset=10&limit=5
{
  "data": [ {"N":11},  {"N":12} ],
  "more":false,
  "pid":"anonymous",
  "type":"success"
}
```

The value of the `more` property now indicates that the query has run to completion and that the pengine with the pid '1f821ea6' has terminated. Therefore, the corresponding cache entry has been removed, and this leaves us with a cache consisting of just two entries:

```
cache(between(1,12,_), 5,  '6a912be5').
cache(p(_, _),         50, '7f2111a1').
```

Next, let us assume that the following request is made:

```
http://remote.org/ask?query=member(X,[a,b,c])&offset=1&limit=1
{
  "data": [ {"X:b} ],
  "more":true,
  "pid":"anonymous",
  "type":"success"
}
```

This is a request for which the cache could offer no help. The actor manager spawned a new pengine and assigned it the task of searching for the second solution to the query. To do this, it had to discard the first solution. Since there are even more solutions, the pengine will continue to exist in a suspended state, and a cache entry will be created which relates the combination of the query identifier and the pid to the offset 2. The cache is updated with this entry, asserted as the last clause of the predicate, which will be of help to a subsequent requests asking for the third solution to `?-member(X,[a,b,c])`:

```
cache(between(1,12,_),   5,  '6a912be5').
cache(p(_, _),           50, '7f2111a1').
cache(member(_,[a,b,c]), 2,  '2e2131b9').
```

The *size* of the cache is defined as the number of entries it contains, or equivalently, as the number of pengines that coexist in the pool. In order to properly maintain and protect the node's resources, a *maximum size* must be set. We expect that a realistic maximum size of a cache/pool would normally be in the tens or perhaps hundreds of thousands of entries, but let us for the sake of the example pretend that the maximum size is just 3, and that our cache has therefore already reached its limit. If a client makes a request that causes a new entry to be added to the cache, an entry must also be removed and the associated pengine be terminated. Suppose that a client makes the following request:

```
http://remote.org/ask?query=member(X,[a,b,c])&offset=0&limit=1
{
  "data": [ {"X:a} ],
  "more":true,
  "pid":"anonymous",
  "type":"success"
}
```

Again, the value `true` of the `more` property suggests that the pengine just created should be suspended and an entry be added to the cache. The entry at the top is retracted in order to keep the size within its limit:

```
cache(p(_, _),           50, '7f2111a1').
cache(member(_,[a,b,c]), 2,  '2e2131b9').
cache(member(_,[a,b,c]), 1,  '44d1d1d1').
```

The usefulness of the method described rests on the assumption that if a client has asked for the nth slice of solutions to a query Q, and if Q does in fact have even more solutions, the chance is quite high that the same client will come back and ask for the next slice too. If this does not happen, there may either be another client that asks for it, or else the cache entry will sooner or later expire and the corresponding pengine terminate. Should the first client *then* make a request for this particular slice, it may well be that a new pengine must be spawned and that solutions must be discarded.

At this point, we can actually calculate how much has been saved by using our method rather than a naive approach. The total number of solutions that the four requests had to compute turns out to be 10, and only one solution was discarded. A naive approach would have to compute 25 solutions, and throw away 16 of them. Although such statistics does not tell us the whole story about the prospects for a RESTful HTTP API which is scalable also in the real world, it does appear to be somewhat promising.

Note that unless questionable methods such as long-polling[8] are employed, and due to the strict request-response regime of HTTP, no API based on it can be compatible with unrestricted I/O. While the very nature of HTTP is what makes it incompatible with general I/O, the proposed caching method is what makes it incompatible with dynamic database updates by means of assert and retract. Developers in need of unrestricted I/O and/or update of dynamic database are advised to use the WebSocket protocol.

Note also that the method implies that the responsibility for managing the node's resources lies with the owner of the node rather than with the client. The maximum size of the cache for a particular node can be specified by its owner by means of a setting. What is a reasonable size depends on the host platform of the node, and in particular on the cost of keeping suspended processes around. Our SWI-Prolog implementation uses Tarau's engines which are quite cheap since they are not attached to an OS thread when they are suspended (cf Section **??**). (An Erlang implementation, if one can be built, would most likely be more scalable in this regard.)

Setting the maximum size of the cache is not the only way to control the resources spent by a node accepting RESTful requests. Normally, request must also be served under a tight time limit and queries that do not terminate within the limit must be forced to do so. This must be done by the node's actor manager, since when a request has been made, the client that made it can no longer intervene.

Let us make a brief assessment of the risks involved in exposing an API of this kind to unauthorised clients. The danger of *denial-of-service* (DoS) attacks, characterised by an *intentional* attempt by attackers to prevent legitimate use of a service by flooding it with requests, is of course always present. Deliberate DoS attacks are difficult to guard against and Web Prolog node are just as much exposed to them as any other service. What is more worrisome is that a node may also be harmed by *unintentional* client behaviour that may cause similar problems as those caused by deliberate attacks. A client making many request in a row where each by mistake leaves an unintended choice points around may for example stress the cache to its limit. The problem is that the node rather than the client takes the hit.

Before closing this section, let us take a step back and again remind the reader about the importance of devising methods that allow nodes to respond to RESTful HTTP requests in an as efficient and economical way as possible. It all starts with the realisation that URIs of the form

```
<BaseURI>/ask?query=<Q>&offset=<N>&limit=<L>
```

constitute a generic Prolog solver API which is in many ways ideal. Such URIs are simple, they are meaningful, they are declarative, they can be bookmarked, responses are cachable by intermediates, and so on. We would therefore suggest that *any* attempt to come up with

[8] https://en.wikipedia.org/wiki/Push_technology

a generic HTTP API to Prolog should take the form of such URIs as its point of departure. Performance might be a challenge, but it is a challenge worth taking on. Failure would come with the cost of not only being unable to offer clients this kind of web API, but also of not being able to offer them what can be built on top of it in the form of high-level predicates for remote procedure calls that can be used with nodes that do not offer WebSockets. All would not be lost, since basing the communication needs of the Prolog Web on only the WebSocket protocol would probably work too, but in relation to the full vision of a Prolog Web a failure to devise an efficient and scalable generic HTTP API would definitely be a clear setback.

[This section is too long and too hairy. I would appreciate hints on how to improve it. Other concepts? A more formal explanation? Perhaps the section should be moved to an appendix?]

6.6 Discussion

As we have already remarked, certain features of the ACTOR profile of Web Prolog may appear to be overkill for a language aiming to be used as a scripting language for the Web. In particular, allowing a client to define, create and be in almost full control of several concurrently running pengines and/or actors may not seem to correspond to any common use cases for such languages. However, overkill is only apparent, because when we look at the Prolog Web from a bird's-eye point of view we should realise that "swarms" of concurrently running actors form its backbone, both when seen as a viable and remarkably simple theoretical model of how might work, and as (possibly one of several) ways to implement it. In other words, along with the protocols that allow them to interact, actors are truly fundamental to the architecture underlying the Prolog Web, regardless of whether they are seen as such by individual programmers or end users of Web Prolog.

As a collective, pengines serve the RESTful HTTP API, and individually they serve the WebSocket API that allows a client to have a more or less permanent connection to an actor throughout a session that may last for a long time.

When designing the APIs, we have tried to emphasise rather than hide the close connection between the predicate APIs and the web APIs. When `pengine_spawn/2`, `rpc/2-3` or `promise/3-4` are used to query a remote node, it is important to understand the extent to which the web APIs are relied upon. In the case of `pengine_spawn/1-2`, when the following call is being made

```
?- pengine_spawn(Pid, [
       node('http://remote.org')
   ]).
```

a number of things would normally take place:

1. Unless a websocket connection between the current node and the node at the address `http://remote.org` is already established, we create one. If one exists, it is used instead.

2. A message {`"command":"pengine_spawn","options":"[format(prolog)]"`} is sent to the remote node over the connection.

3. The call waits for a message in the form of a Prolog term `spawned(Pid)`. When this arrives, the first argument of `pengine_spawn/1-2` is instantiated to the value of the variable `Pid` and then the call terminates.

Consider a follow-up query such as this one:

```
?- pengine_ask($Pid, member(X,[a,b,c])).
```

All messages received from the remote node are going to appear in the mailbox of the local process that called `pengine_spawn/1-2`. Since the underlying transport is WebSocket, and since WebSocket is essentially an asynchronous protocol, to the programmer, this will appear as if the communication between the local and remote process is asynchronous. Calls to predicates such as `pengine_next/1-2`, `pengine_stop/1`, `pengine_abort/1` and `pengine_exit/1` will have to work in the same non-blocking way, just adding messages to the mailbox for the caller to handle as it sees fit.

Chapter 7

Remote procedure calls (RPC)

Web Prolog supports two kinds of predicate APIs for making remote procedure calls, one which is synchronous and very easy to use, and one which is asynchronous and somewhat harder to use.

7.1 Non-deterministic RPC

For the purpose of a very straightforward approach to the distribution of programs over two or more nodes, Web Prolog offers `rpc/2-3`, a meta-predicate for making non-deterministic remote procedure calls. No real concurrency is involved, and this is probably what makes it so simple to use.

The `rpc/2-3` predicate allows a process running in a node A to call and try to solve a query in the Prolog context of another node B, taking advantage of the data and programs being offered by B, just as if they were local to A. (Recall that `pengine_spawn/1-2` can also do this, but only in a more roundabout way.) As a client, a Web Prolog process queries a node by calling `rpc/2` with the first argument a URI pointing to the node, and the second argument a goal that we want to run with the Web Prolog programs offered by the node. Here is a trivial example of its use:

```
?- rpc('http://remote.org', parent(Parent,valdemar)).
Parent = ida ;
Parent = johan.
?-
```

Calls to `rpc/2-3` can of course be made also from the body of a clause:

```
foo(X) :-
    rpc('http://remote.org', bar(X)),
    baz(X).
```

However, we suspect that this is in general the preferred way to structure code:

```
foo(X) :-
    bar(X),
    baz(X).
```

```
bar(X) :-
    rpc('http://remote.org', bar(X)).
```

This also creates the opportunity to mix in local data with data retrieved from a remote node, like so:

```
bar(a).
bar(X) :-
    rpc('http://remote.org', bar(X)).
bar(z).
```

Tabling might be interesting to use as a cache:

```
:- table bar/1.

bar(X) :-
    rpc('http://remote.org', bar(X)).
```

7.2 Working with rpc/2-3 options

Options accepted by `rpc/3` are inherited from `pengine_spawn/1-2` and `pengine_ask/2-3`. Passing a `timeout` option allows us to set a max time for answers to appear. If this does not happen, an exception is thrown. In this way, the hanging of a process waiting for a node that is down can be avoided.

For the purpose of reducing the number of network roundtrips, the option `limit` may be passed:

```
?- rpc('http://remote.org', parent(Parent,valdemar), [
       limit(2)
   ]).
Parent = ida ;
Parent = johan.
?-
```

As the above example tries to convey, the behaviour of the call, as seen from the point of view of the client, does not change. After having been presented with the first solution to the query the programmer still needs to type a semicolon in order to see the next solution. But under the hood, the next solution has already been computed and returned to the client as the second member of a list containing both solutions. Thus no new request to the node needs to be made. So while the retrieval of the two solutions to the query required two network roundtrips before we applied the option, it will now only require one roundtrip. Had there been four solutions to our query, two roundtrips would have been necessary. The meaning of the query is still intact. The `limit` option serves as a performance optimisation only, and has nothing to with logic and the declarative reading of the query. In Chapter **??** we will see exactly how and why this works.

The options that `rpc/2-3` inherits from `pengine_spawn/2-3` include the options that are used for the injection of source code into the workspace of the remote pengine. They will be explained in the two sections that follow.

7.3 Bringing code to the data

On the internet, the cost of shuffling data back and forth across remote boundaries is significant, yet cannot be avoided. But what do we mean by “data” and in which direction should we shuffle it? Is bringing the code to the data better than bringing the data to the code? The answer is most likely that it varies.

It is often more efficient to bring a few lines of code to the data rather than moving an often large amount of data to the code. In the case of data being frequently updated (think stock market data for example) the gain is probably significant. On the other hand, if the data is more or less static (think geo-political data for example) and we want to query it a lot, then bringing the data to the code may be more efficient.

The obvious way to bring code to the data in Web Prolog is to inject source code into the remote processes started by `rpc/2-3`. In the following two examples, we do just that. The first example demonstrates that `rpc/3` will accept more than one `src_*` option:[1]

```
?- rpc('http://remote.org', foo(X), [
       transport(websocket),
       src_text("foo(a)."),
       src_list([foo(b), foo(c)])
   ]).
X = a ;
X = b ;
X = c.
?-
```

In our second example we instead assert three clauses of `foo/1` and then call `rpc/3` passing the `src_predicates` option with the value `[foo/1]`:

```
?- assertz(foo(a)), assertz(foo(b)), assertz(foo(c)).
true.
?- rpc('http://remote.org', foo(X), [
       transport(websocket),
       src_predicates([foo/1])
   ]).
X = a ;
X = b ;
X = c.
?-
```

7.4 Bringing data to the code

The special purpose URI `localnode` serves as the default value of the `node` option for both `spawn/2-3` and `pengine_spawn/1-2`. Thus it follows, perhaps a bit counter-intuitively, that in combination with the `src_uri` option `rpc/3` can also be used to bring the data to the code:

```
?- rpc(localnode, mother(X, valdemar), [
```

[1]Therefore, an order must be defined over them. We propose that we use the order in which they are placed in the list.

```
        src_uri('http://remote.org/src')
    ]).
```

To understand what this means, we must consider the definition of `rpc/2-3` in Section 7.6. A pengine is spawned on the node pointed at in the first argument. This would be a local pengine running in its own workspace isolated from other pengines running on the same node, but it is still able to "see" the clause defining `mother/2`. The source held by the node at `http://remote.org` and accessible at `http://remote.org/src` is injected into the workspace before the goal `mother(X,valdemar)` is tried.

Bear in mind that since `rpc/2-3` is not supposed to run in parallel to the calling process, there is no need to spawn a separate process. When the first argument of `rpc/2-3` is `localnode`, the call will run just fine in the same process as the caller. We elaborate on this point in Section **??**, where we also provide an implementation that avoids creating the spurious process.

We are reminded once again of the symmetry allowing Web Prolog code to flow in either direction, from the client to the node or from the node to the client. The choice is determined by the programmer's selection of combinations of options that will configure the actor to be created, but the choice can in principle also be made programmatically at runtime.

When discussing the bringing of "code to the data" versus "data to the code", it is important to keep in mind that the distinction between code and data is not as sharp in Prolog as it is in other programming languages. In Prolog, as in other homoiconic programming languages, data is code and code is data. Suppose the node at `http://remote.org` contains `bar(X):-foo(X)`, a clause that most people would consider code rather than data. The following call seems to bring data to the code, rather than the other way around.

```
?- rpc('http://remote.org', bar(X), [
        src_text("foo(a). foo(b). foo(c).")
    ]).
```

What seems to matter most is the Prolog context that a node provides, a context that is not always portable, but may be determined by the host environment, parts of which may not be directly accessible for reasons of security.

7.5 Promises

The API described above gives us almost total control over a pengine that we spawn. We can decide to use it for a one-shot query, or if we think that we will need it for more than one query, run a whole session. We can choose to inject a program, or perhaps only a few clauses, into the pengine at the time of creation, code that will remain private to it and that other pengines running on this node will not see. We are furthermore able to abort a query should the need arise. Finally, the node allows us to terminate the pengine at will. For almost all intents and purposes, although we may not be the owner of the node, we do own the pengine running on it.

In this section we introduce another, much simpler way to query a node, where we ask the node directly, rather than aiming our query at a particular pengine running on the node. We do not have to spawn a pengine and get hold of a pid. Instead of a pid, we must pass the URI of the node in the first argument of `promise/3-4`:

```
?- promise('http://remote.org', parent(Parent,valdemar), Reference, [
       template(Parent)
   ]).
Reference = 'f7780f96-eda8-4117-9b88-92cd1e90d4f0'.
?- yield($Reference, Answer, [timeout(3)]).
Answer = success(anonymous,[ida],true).
?-
```

Note that anonymous pengines are somewhat crippled compared to "named" ones. In dealings with anonymous pengines, injecting code into the process to be created is still possible, but the use of assert and retract is not allowed. An anonymous pengine simply lacks the private workspace necessary for that to work. Furthermore, anonymous pengines do not run sessions and thus only one-shot queries are allowed. In fact, they do not even have to *be* pengines, but must behave *as if* they were.

The direct querying of a node comes with benefits for the client as well as for the node. For the client it means that the duration between asking for the first answer to a query and asking for the next answer may be arbitrarily long. Things will still work as they should. For the node it means that instead of having each client tie up part of the node's resources for as long as the conversation lasts, it can release them as soon as it wants.

Since we did not use `pengine_spawn/1-2` to explicitly create a pengine on the node, predicates such as `pengine_stop/1`, `pengine_abort/1` and `pengine_exit/1` cannot be used since they are expecting the passing of a pid. Therefore, the use of anonymous pengines implies that the node must take full responsibility for the management of resources.

7.6 An implementation of rpc/2-3

By default, `rpc/2-3` uses the RESTful HTTP API to communicate with the remote node. Calling `rpc(<URI>,<Query>)` involves making one or more requests for consecutive slices of solutions to `<Query>` using the stateless HTTP protocol. Each request takes the form `<URI>/ask?query=<Query>&offset=N&limit=L` where `N` is initially `0` and is incremented by `L` between requests. But then, of course, we have to handle the necessary options as well as the responses too. A somewhat sketchy implementation may look as follows:

```
rpc(URI, Query, Options) :-
    option(limit(Limit), Options, 1),
    rpc(URI, Query, 0, Limit).

rpc(URI, Query, Offset, Limit) :-
    parse_url(URI, Parts),
    format(atom(QueryAtom), "(~p)", [Query]),
    rpc(Query, Offset, Limit, QueryAtom, Parts).

rpc(Query, Offset, Limit, QueryAtom, Parts) :-
    parse_url(ExpandedURI, [ path('/ask'),
                             search([ query=QueryAtom,
                                      offset=Offset,
                                      limit=Limit,
                                      format=prolog
```

```
                                   ])
                          | Parts]),
    setup_call_cleanup(
        http_open(ExpandedURI, Stream, []),
        read(Stream, Answer),
        close(Stream)),
    wait_answer(Answer, Query, Offset, Limit, QueryAtom, Parts).

wait_answer(error(anonymous, Error), _, _, _, _, _):-
    throw(Error).
wait_answer(failure(anonymous), _, _, _, _, _):-
    fail.
wait_answer(success(anonymous, Solutions, false), Query, _, _, _, _) :-
    !,
    member(Query, Solutions).
wait_answer(success(anonymous, Solutions, true),
            Query, Offset0, Limit, QueryAtom, Parts) :-
    (   member(Query, Solutions)
    ;   Offset is Offset0 + Limit,
        rpc(Query, Offset, Limit, QueryAtom, Parts)
    ).
```

7.7 Another implementation of rpc/2-3

This section looks at a simple implementation of `rpc/2-3` built on top of a pengine that is spawned on a remote node when the predicate is called. Simplifying somewhat, here is our implementation:

```
rpc(URI, Query, Options) :-
    pengine_spawn(Pid, [
         node(URI),
         exit(true),
         monitor(false)
       | Options
    ]),
    pengine_ask(Pid, Query, Options),
    wait_answer(Query, Pid).

wait_answer(Query, Pid) :-
    receive({
        failure(Pid) -> fail;
        error(Pid, Exception) ->
            throw(Exception);
        success(Pid, Solutions, true) ->
            (   member(Query, Solutions)
            ;   pengine_next(Pid),
                wait_answer(Query, Pid)
            );
        success(Pid, Solutions, false) ->
            member(Query, Solutions)
    }).
```

When spawning the pengine we pass `exit(true)` in order to make sure that the process terminates when the first query being sent to it has run to completion. Note also that we do not monitor the spawned process. Thus, when the query has run to completion, no `down` message is going to come along and report this, and so the process running the `wait_answer/2` predicate does not have to bother looking for this message in its mailbox. It can safely terminate on failure, error or when having visited the last solution in the last success message received from the node.

The first clause of the implementation shows clearly how the inheritance of options from `pengine_spawn/2` and `pengine_ask/3` to `rpc/2-3` works. Note that the value of an option passed explicitly to `pengine_spawn/2` takes precedence over the value of the same option if it's given in the third argument of `rpc/3`. Thus, the only options that can have an effect in a call to `rpc/3` are the `timeout` and the `src_*` options.

The most interesting parts of the implementation are the use of the disjunction in the body of the third receive clause and the use of `member/2` in the third and fourth clauses. They are responsible for turning the deterministic calls made by `pengine_ask/3` and `pengine_next/1` into the non-deterministic behaviour that we want `rpc/2-3` to show.

This implementation should be clean enough to nail down the semantics of `rpc/2-3`. It should be noted though, that there are other ways to implement `rpc/2-3`, ways that do not depend on `pengine_spawn/2` and friends, and therefore not on WebSockets.

It may be helpful to think of what is going on here as a very simple kind of *buffering*, and in particlar as the use of what is usually referred to as a *bounded buffer*. The value of the `limit` option represents the size of the buffer. The producer (i.e. the remote pengine process) is allowed to get ahead of the consumer (i.e. the process running the `rpc/3` call), but only until the buffer is full. The consumer can take elements from the buffer immediately without waiting. When the buffer is empty, the producer is prompted to produce more elements, until the buffer is full. In our case however, the consumer and the producer are not running concurrently, i.e. they do not execute different tasks during overlapping periods of time. A possible extension would be to allow them to do so by letting the remote process compute additional solutions while waiting for a `next` message and return all available solutions if it arrives. If a `stop` message arrives instead, the additional solutions are discarded.

7.8 Implementation of promise/3-4

In the case of `promise/3-4`, when the following call is being made

```
?- promise('http://remote.org', member(X, [a,b,c]), Reference).
```

a couple of other things are normally happening:

1. ...
2. ...

Since the transport is HTTP – an intrinsically synchronous protocol – care must be taken. One idea is to make the GET request, wait for its response, and then send the response to the mailbox of the process that called `promise/3-4`, all in a lightweight separate actor process that will terminate afterwards.

The asynchronous way of working proposed here may perhaps be compared to how the `XMLHttpRequest` object available in a web browser works. There too, despite the fact that HTTP is a synchronous protocol, a request being made will not (normally[2]) stop the caller from continue doing what it did when it made the request. The main motive is that we do not want to cause the web browser to "freeze" while waiting for a response to arrive. Instead, the request is (normally) made in the asynchronous mode. The response will be made available in the form of a callback.

7.9 Remote modules

Some rather interesting schemes built on top of `rpc/2-3` are easy to imagine and we give a couple of examples in this section and in the sections that follow. We first sketch a mechanism for the definition of *remote modules*, and then, as a step towards full network transparency, a construct referred to as *global goals*. We also suggest a way to allow a meta interpreter to generate proofs for programs that are distributed by means of `rpc/2-3`.

Below, we sketch an implementation of *remote modules*:[3]

```
use_remote_module(URI, ImportList, Options) :-
    maplist(import(URI, Options), ImportList).

import(URI, Options, Functor1/Arity as Functor2) :- !,
    functor(Head1, Functor1, Arity),
    Head1 =.. [Functor1|Args],
    Head2 =.. [Functor2|Args],
    assertz((Head2 :- rpc(URI, Head1, Options))).
import(URI, Options, Functor/Arity) :-
    functor(Head, Functor, Arity),
    assertz((Head :- rpc(URI, Head, Options))).
```

A remote module import directive is, thanks to the `src_*` predicates, able to import *any* predicate into the current process, even those that are not defined by the remote node. Using the `src_*` predicates, the calling node simply needs to send along a definitions of the predicate in question, in terms of predicates that *are* defined by the remote node. A suggestion for how to do this is provided by the following example:

```
:- use_remote_module('http://remote.org', [foo/1], [
       src_list([(foo(X) :- bar(X), baz(X))])
   ]).
```

[2]We write "normally" here since is also possible to use the `XMLHttpRequest` object to make synchronous requests too, although this is seldom recommended.

[3]What we here refer to as *remote modules* are similar to what Ciao Prolog provides in the form of *active modules*.

7.10 Global goals

Our second example takes a step towards network transparency and is inspired by (Loke, 2006) and an approach to peer-to-peer computing that offers calling *global goals* of the form `?-*Goal`. The existence of a database of peers is assumed, something that can be represented as below:

```
peer(foo(_), 'http://ex1.org', [limit(10)]).
peer(bar(_), 'http://ex2.org', [timeout(1),limit(100)]).
peer(bar(_), 'http://ex3.org', [timeout(3),limit(100)]).
peer(baz(_), 'http://ex3.org', [timeout(1),limit(50)]).
```

Given such a database,[4] `*/1` can easily be implemented like so:

```
:- op(100, fx, *).

* Goal :-
    peer(Goal, Peer, Options),
    catch(rpc(Peer, Goal, Options), timelimit_exceeded, false).
```

The `catch/3` is there to ensure that the call fails if the call to `rpc/3` throws a timeout exception. Here is an example:

```
?- *bar(X).
X = a ;
X = b.
?-
```

[Check (Loke, 2006) for ways to avoid circularity.]

7.11 rpc/2-3 and proof trees

As an example of a more interesting use of `rpc/2-3` and code injection we suggest a way to generate a *proof tree* for a query that uses rules that may call `rpc/2`. The idea is to use a proof constructing meta-interpreter that injects *itself* in the pengine running the call to `rpc/3`. (Look carefully at the second clause of `prove/2` below.) With this program it is possible to generate a proof tree for any query (involving the subset of Web Prolog that the meta-interpreter can handle), even those that need access to another node to be answered (and which in turn may need yet another node, and so on).

```
prove(true, json{label:true}) :- !.
prove(rpc(URI, Bs), Proof) :- !,
    rpc(URI, prove(Bs, Proof0), [
        src_predicates([prove/2, prove_body/2])
    ]),
    atomic_list_concat([Proof0.label, URI], '@', Node),
    Proof = json{label:Node, children:Proof0.children}.
prove(H, Proof) :-
    clause(H, Bs),
```

[4]We make no assumptions on how the peer database is updated.

```
    prove_body(Bs, BSProof),
    term_to_atom(H, HA),
    Proof = json{label:HA, children:BSProof}.

prove_body((B, Bs), [BT|BTs]) :- !,
    prove(B, BT),
    prove_body(Bs, BTs).
prove_body(B, [BT]) :-
    prove(B, BT).
```

Note the use of the `rpc/3` option `src_predicates(List)`. It sends the local predicates denoted by `List` to the underlying remote pengine, where `List` is a list of predicate indicators. Note also the use of dicts for building the proof tree. This ensures that the proof tree will reach the client in the form of JSON, where it can be rendered as a tree.

```
?- prove(r(X,Y), Proof),
   draw_tree(Proof).
```

[Insert trees from example run with database in Figure 2].
The above meta-interpreter is likely to blow up for scenarios involving very large proof trees. This is not really a problem since the suggestion is not to make `prove/2` a builtin predicate. It is the author of the client code who decides what kind of proofs that should be built and how, and it is this author's responsibility to write a version of a proof collector predicate that works for a given scenario.

In practice we probably do not want to show the proof at the clause level. The approach may however also be useful for showing the distribution of the query solving work over multiple nodes. Such a distribution can also be rendered as a tree. In order to avoid including clause level proofs, we need to replace the third clause of `prove/2` with:

```
prove(H, Proof) :-
    clause(H, Bs),
    prove_body(Bs, Proof).
```

If we make this change we are no longer building proofs, so we should probably also change the name of the predicate, to better fit its purpose.

7.12 RPC predicate APIs

Predicate: `rpc/2-3`

```
rpc(+URI, +Query) is nondet.
rpc(+URI, +Query, +Options) is nondet.
```

Semantically equivalent to the sequence below, except that the query is executed in (and in the Prolog context of) the node referred to by `URI`, rather than locally.

```
copy_term(Query, Copy),
call(Copy),              % executed on node at URI
Query = Copy.
```

All the options for `pengine_spawn/3` are valid for `rpc/3` too, except for `node`, `exit` and `monitor`.

Predicate: `promise/3-4`

```
promise(+URI, +Query, -Reference) is det.
promise(+URI, +Query, -Reference, +Options) is det.
```

Make asynchronous RPC call to node `URI` with `Query`. This is a type of RPC that does not suspend the caller until the result is computed. Instead, a reference is returned, which can later be used by `yield/2-3` to collect the answer. The reference can be viewed as a promise to deliver the answer. Valid options are `template`, `offset`, `limit` and `timeout`.

Predicate: `yield/2-3`

```
yield(+Reference, ?Message) is det.
yield(+Reference, ?Message, +Options) is det.
```

Returns the promised answer from a previous call to `promise/3-4`. If the answer is available, it is returned immediately. Otherwise, the calling process is suspended until the answer arrives from the node that was called. The only valid option is `timeout`.

Note that this predicate must be called by the same process from which the previous call to `promise/3-4` was made, otherwise it will not return.

Chapter 8

Programming reactivity

The modern Web is not comprised of static collections of information only, but constitutes a dynamic, state-changing system. Reactivity, the ability to detect and react to events signalling state changes, is therefore indispensable for many applications on the Web. New information comes in, calling for insertions of new data; information is out-of-date, calling for deletions and replacements on data, and complex workflows of often time-dependent actions and messaging may have to be implemented. In this chapter we will show that the receive primitive in combination with the network-transparent messaging capabilities provided by Web Prolog can give us most of what we need.

8.1 Rules and state machines

In order to handle reactivity in an as elegant and high-level way as possible, condition-action rules, event-condition-action rules, and various forms of state machines are typically employed. In this chapter, we shall distinguish between the following constructs:

Event-condition-action (ECA) rules of the form

```
on Event if Condition do Action.
```

where `Action` is performed if `Event` happens and `Condition` is satified.

Event-driven state-machines with transitions of the form

```
in State on Event if Condition do Action go NewState.
```

where `Action` as well as a transition to `NewState` is performed if the machine is in State, `Event` happens and `Condition` is satified. As we shall see, such machines may also support condition-action transitions of the form

```
in State if Condition do Action go NewState.
```

Since they involve actions having side-effects, such constructs are clearly less declarative than constructs based on nothing but pure (side-effect free) logic. However, we agree with those who argue that this is inherent to the task and should not withhold efforts to make reactive constructs as declarative as possible (Bry and Eckert, 2006). We believe that Web Prolog has placed us in a good position to do just that.

8.2 Network transparent event messaging

Constructs such as these are not for node-internal use only, but must be able to play a role in the communication between nodes as well. On the Prolog Web, the communication between nodes is based on a peer-to-peer communication model where all parties (when they are `actor` nodes) have the same capabilities and every party can initiate a communication session. Event messages, i.e. messages representing events that have occurred on the Web, are directly communicated between nodes. Here, two strategies are possible: the *push* strategy, i.e. a node informs other (possibly) interested nodes about events, and the *pull* strategy, i.e. interested nodes poll data residing on other nodes in order to detect changes. Both strategies may be useful and both are supported by Web Prolog. The pull strategy is supported by queries to persistent data by means of `rpc/2-3`, and through the use of the `!/2` operator (and the underlying WebSocket protocol) Web Prolog offers also the push strategy.

8.3 Implementing ECA rules

Thanks to the receive construct, Web Prolog provides a simple and direct way to encode a system of ECA rules. Patterns in receive clauses can be used to match against event messages, guards to encode conditions, and the bodies of receive clause to perform actions. Consider the following two ECA rules

```
on e1(X) if c1(X,Y) do a1(Y).
on e2(X) if c2(X,Y) do a2(Y).
```

and an example showing how one may implement them by using the receive construct in a loop:

```
run :-
    receive({
        e1(X) when c1(X,Y) ->
            a1(Y),
            run;
        e2(X) when c2(X,Y) ->
            a2(Y),
            run
    }).
```

Calling `run/0` will enter a loop waiting for an event message of the form `e1(X)` or `e2(X)` to appear in the mailbox of the process running the loop. If `e1(X)` is matched and `c1(X,Y)` holds, `a1(Y)` will be called, and if `e2(X)` is matched and `c2(X,Y)` holds, then `a2(Y)` is called.

Note that since guards in Erlang are severely restricted (cf. Section 3.1), this kind of translation of rules into code works much better in Web Prolog than in Erlang. In Web

Prolog, conditions (represented as guards) can be formulated declaratively in the logic of pure Web Prolog and the evaluation of a condition may bind variables to values that are carried over to the body of the receive clause, something which is not possible in Erlang. By using `rpc/2-3` the evaluation of conditions may even be done against remote nodes.

Actions are not in general declarative since their purpose is usually to actually *do* something – insert a new clause in the dynamic database, send a message to a (local or remote) actor, spawn a new actor, etc. Actions can be rather complicated and composed of many smaller actions.

8.4 Implementing state machines

The step from a system of ECA rules to an event-driven state machine is short and Web Prolog's receive primitive can be used to implement simple machines in straightforward code. Figure 8.1 depicts a machine comprising two states and four transitions.

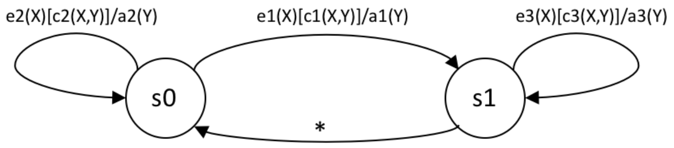


Figure 8.1: A simple state machine

Using predicate names to encode states, here is how it can be implemented in Web Prolog:

```
s0 :-
    receive({
        e1(X) when c1(X,Y) ->
            a1(Y),
            s1;
        e2(X) when c2(X,Y) ->
            a2(Y),
            s0
    }).
```

```
s1 :-
    receive({
       e3(X) when c3(X,Y) ->
            a3(Y),
            s1;
       _AnyEvent  ->
            s0
    }).
```

8.5 Implementing interpreters

One problem with what we have seen so far is that rules do not *look* like rules, and that state machine specifications do not look familiar either. As always, it is a good idea to separate the (almost) declarative rules/transitions from how they are executed by an interpreter implementing a suitable algorithm. Consider the following specification of the machine in Figure 8.1:

```
in s0 on e1(X) if c1(X,Y) do a1(Y) go s1.
in s0 on e2(X) if c2(X,Y) do a2(Y) go s0.
in s1 on e3(X) if c3(X,Y) do a3(Y) go s1.
in s1 on _AnyEvent if true do true go s0.
```

The following Web Prolog program implements an interpreter for such systems:

```
state_machine(State) :-
    receive({Event -> true}),
    state_machine(Event, State).

state_machine(Event, State) :-
    in State on Event if Condition do Action go NewState,
    once(Condition),
    !,
    once(Action),
    state_machine(NewState).
state_machine(_Event, State) :-
    state_machine(State).
```

Note that since the receive will match any message, events will never be deferred, but due to the catch-all clause in the definition of `state_machine/2`, they will have no effect if there is no matching transition. Note also that actions are not allowed to fail, because if they do, `state_machine/1` fails.

The following example suggests how an ECA transition in a machine implementing network-transparent reactivity might look like:

```
in
  state_5
on
  foo(X)
if
  rpc('http://remote.org', bar(X,Y))
do
  assert(baz(Y)),
  counter ! done(Y)
go
  state_9.
```

8.6 Handling CA rules

The code below adds a single clause in the front of the previous interpreter. This allows it to handle CA transitions as well as ECA transitions. The CA transitions always take precedence over the ECA transitions in the sense that if no CA transition can be taken from the current state, it will suspend waiting for an event message to show up that can eventually be handled by an ECA transition.

```
state_machine(State) :-
    in State if Condition do Action go NewState,
    once(Condition),
    !,
    once(Action),
    state_machine(NewState).
state_machine(State) :-
```

```
    receive({Event -> true}),
    state_machine(Event, State).

state_machine(Event, State) :-
    in State on Event if Condition do Action go NewState,
    once(Condition),
    !,
    once(Action),
    state_machine(NewState).
state_machine(_Event, State) :-
    state_machine(State).
```

8.7 Summing up

This completes our demonstration of the usefulness of the `receive/1-2` primitive for programming reactivity in Web Prolog. In combination with predicates such as `!/2` and `rpc/2-3`, implemented on top of transport protocols such as HTTP and WebSocket, it appears that Web Prolog allows us to have all the network-transparent messaging that we could possibly ask for. We hasten to add that the purpose of the demonstration has *not* been to suggest that the demonstrated interpreters and related formats for rules and state-machine specifications should be available in Web Prolog out of the box. More sophisticated machineries may be called for, perhaps with support for features such as these:

- Hierarchical state machines
- Actions performed when entering and exiting states
- Hibernation of state machines
- Various forms of time-outs
- Postponing (deferring) of events
- Handling of complex event sequences

As far as we aware, event-driven state machines have not been used much in the Prolog world. Perhaps the reason is that primitives for sending and receiving messages did not appear in Prolog until (a few) platforms implemented the ISO Prolog Threads standard. As can be seen by an OTP behaviour such as `gen_statem`,[1] which implements most of the features listed above, Erlang takes event-driven state machines very seriously. Perhaps it is time for Prolog to follow suit.

[TODO: Explore the connection to SCXML – se Section B.3. Perhaps the section on generic event handling from the same section should be part of this section too?]

[1]`http://erlang.org/doc/man/gen_statem.html`

Chapter 9

Web Prolog and the Prolog Web

The Network is the Computer – *John Gage, Sun Microsystems, 1984*

An interesting application made possible by Web Prolog is what we have already started to refer to as the *Prolog Web* – Prolog programs distributed over an overlay network of nodes acting as both clients and servers. The Prolog Web forms the (global) environment in which pengines and other actors are born, act, and die.

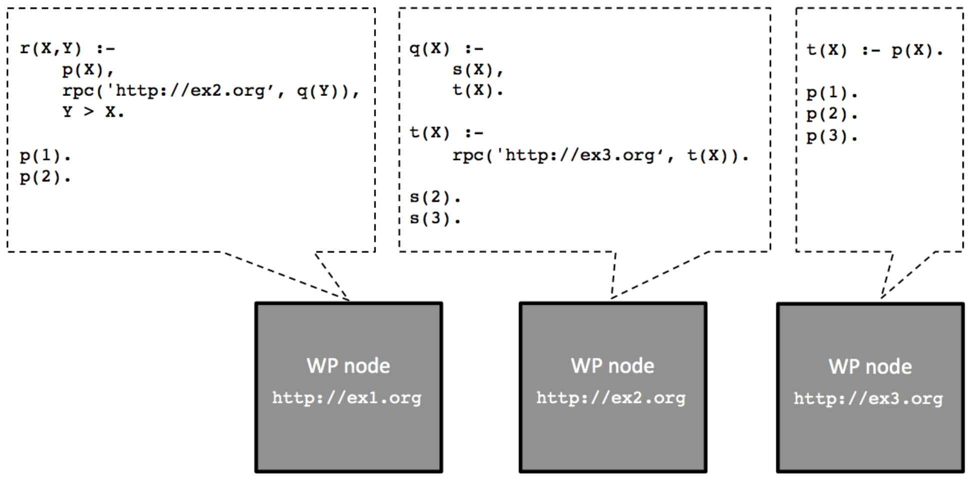
In its simplest form, the Prolog Web is characterised by the interlinking of Prolog programs by means of `rpc/2-3` and a URI resolving to what is essentially a remote Prolog module. The idea is very simple, bordering on the naive, but this may in fact be its biggest strength.

As an example of a trivial fragment of the Prolog Web, consider Figure 9.1, where a scenario involving three nodes is depicted. Each node is named by a URI, and each node makes available a Web Prolog program, displayed inside the boxes with dashed borders. Apart from the programs that they hold, the nodes should be regarded as "black boxes".
Since the Prolog Web is a network of nodes rather than a programming language, it provides a different perspective on the relation between Prolog and the Web, one in which the Web is emphasised. Looking at it from this angle, we can perhaps say that we focus on the whole rather than the parts.

The Prolog Web is simultaneously a Web of Code, a Web of Actors, and a Web of Answers.

The Web of Code can be crawled by following the links resulting from adding the path `/src` to the base URI. Thus, the URI `http://ex.org/src` will get us the source code (= representation) of the program (= resource) held by the node at `http://ex.org`. The MIME type of a resource consisting of Web Prolog source code is `text/web-prolog` (or `x-web-prolog` until `web-prolog` has been registered as a MIME subtype with IANA).

The Web of Actors is the aspects of the Prolog Web which links running pengines and other actors into a network. Actors such as pengines are typically created by sending a spawn message over a WebSocket connection. Actors are identified by their pids. In some profiles, the code executed by the actor may be a combination of the code held by the



```
r(X,Y) :-
    p(X),
    rpc('http://ex2.org', q(Y)),
    Y > X.

p(1).
p(2).
```

```
q(X) :-
    s(X),
    t(X).

t(X) :-
    rpc('http://ex3.org', t(X)).

s(2).
s(3).
```

```
t(X) :- p(X).

p(1).
p(2).
p(3).
```

Figure 9.1: A tiny Prolog Web [TODO: This example has to be made into a truly pure program. The use of the larger-than operator has to be eliminated.]

node and code contributed by the client. It is therefore not possible to equate the Web of Code and the Web of Actors. Besides, the Web of Actors is dynamic in a way that the Web of Code is not, with pengines and other actors frequently being created, used for a while, and then destroyed, in more or less quick successions.

The Web of Answers can be accessed by adding the path `/ask` plus a number of parameters to the base URI. Thus, as explained in much more detail in Section 6.3, the URI `http://ex.org/ask?query=p(X)` will get us the first answer to the query `?-p(X)` over the program held by the node at `http://ex.org`. More answers, if any, can be retrived by appending parameters such as `offset` and `limit`. The MIME type of an answer is `text/web-prolog` or `application/json`, depending on the value of the `format` option.

Normally, a base URI such as `http://ex.org` will take us to an instance of SWISH, or to a similar GUI, or result in a 404 response if no such GUI exists. The MIME types involved here are of course the usual ones, such as `text/html`.

9.1 Using modules to simulate a Prolog Web

The whole thing is quite simple. We can write a fairly accurate simulation of the tiny Prolog Web above, using ordinary Prolog modules:

```
:- module(ex1,[]).
:- use_module(ex2).

r(X, Y) :-
    p(X),
    ex2:q(Y),
    Y > X.

p(1).
p(2).
```

```
:- module(ex2,[]).
:- use_module(ex3).

q(X) :-
    s(X),
    t(X).

t(X) :-
    ex3:t(X).

s(2).
s(3).
```

```
:- module(ex3,[]).

t(X) :- p(X).

p(1).
p(2).
p(3).
```

Figure 9.2: A tiny Prolog Web simulated by standard Prolog modules

In a traditional Prolog system such as SWI-Prolog, we can test the simulation as follows:

```
?- use_module(ex1).
true.
?- ex1:r(X,Y).
X = 1,
Y = 2 ;
X = 1,
Y = 3 ;
X = 2,
Y = 3.
?-
```

The simulation is accurate up to the point of giving the same solutions, generated in the same order, as with the "real" Prolog Web example in Section 9.3. Furthermore, the data encapsulation provided by modules avoids the potential danger of mixing predicates with the same name and arity (`p/1` in the example) and thus simulates another important aspect of the Prolog Web. However, in the simulation, contrary to in the real Prolog Web, solutions are computed using just *one* process running on just *one* computer. In neither the Prolog Web example nor in the simulation is concurrency at work. Everything is sequential.

9.2 Pure Web Prolog and the Pure Prolog Web

Regardless of which notion of *pure Prolog* we accept (there is more than one![1]) it seems reasonable to describe `rpc/2-3` as a predicate that is pure, at least as long as the goal that gets called in the second argument is pure. Indeed, the following seems to hold:[2]

Pure Web Prolog = *Pure Prolog* + *`rpc/2-3`*

[1] https://www.w3.org/2005/rules/wg/wiki/Pure_Prolog

[2] We may want to compare this with the `call/N` family of predicates. They too retain logical purity of the predicates they call.

When a particular partition of the Prolog Web is written in pure Web Prolog it forms a piece of *Pure Prolog Web*:

A Pure Prolog Web = a partition of the Prolog Web written in Pure Web Prolog

With a large number of nodes running pure Web Prolog programs that interlink to other pure Web Prolog programs, we may *in theory* be able to create a Web of Logic on top of the conventional Web, a layer that can *in principle* grow as big as we want it to grow, and even into a network spanning the whole globe, just like the conventional web has done, while still adhering to the formal (model or proof theoretic) semantics of the pure subset of the Prolog language.

Even in scenarios where a portion of the Prolog Web is pure, it often makes sense to regard each node as a kind of agent, and to treat the Web Prolog code held by the node as the *beliefs* of this agent. Thinking about programs in this way, which is really only possible when programs are declarative, suggests that `rpc/2` can be treated as a higher-order *epistemic operator* where a call such as `rpc(URI,Query)` is actually asking whether the pengine at `URI` *believes* that `Query`. Thus, a clause such as

```
r(X,Y) :- p(X), rpc('http://ex2.org',q(Y)), Y > X.
```

can be read declaratively, as "`r(X,Y)` is true if `p(X)` is true, and it is true that the pengine at `http://ex2.org` believes that `q(Y)`, and it is true that `Y` is larger than `X`". That is, the clause expresses a logically sufficient condition for `r(X,Y)` to hold. Or in the syntax of predicate logic enhanced with a higher-order predicate *believes/2*:

$$\forall x \forall y[p(x) \wedge believes(ex_2, q(y)) \wedge x > y \rightarrow r(x,y)]$$

Thinking about it in this way, it makes perfect sense to say that one node believes that `p(3)` is true, while another node does not:

```
?- rpc('http://ex3.org', p(3)).
true.
?- rpc('http://ex1.org', p(3)).
false.
?-
```

The following query simply asks whether the nodes `http://ex1.org` and `http://ex3.org` *agree* that `p(X)`:

```
?- rpc('http://ex1.org', p(X)),
   rpc('http://ex3.org', p(X)).
X = 1 ;
X = 2.
?-
```

Two nodes may hold contradictory beliefs, and sometimes this can be detected:

```
?- rpc('http://ex1.org', square(Item)),
   rpc('http://ex3.org', circle(Item)).
Item = a132.
?-
```

We do not have to go as far as to define a special epistemic operator, however. Another way to understand what is going on here is to realise that any realistic knowledge representation language needs an explicit construct for specifying the *scope* or the *context* of the inference – the knowledge base with respect to which inference is to be made. As used in `rpc/2-3`, the humble URI can be seen as a way to indicate such a scope. As argued by Kifer et al. (2005), scoped inference is in fact mandatory for realising Negation As Failure (NAF) on the Web, because to apply any form of the closed world assumption, one needs to know the entire knowledge base first, something which is not really possible for the Web since its boundaries cannot be clearly delineated.

9.3 A web of agents known as pengines

The Prolog Web follows the tradition of the Internet and the conventional Web in that its important interfaces are defined in terms of protocols specifying the syntax, semantics, and sequencing constraints of the messages interchanged.

Up till now, Prolog Web nodes have been regarded as black boxes and the logical aspect of the Prolog Web could be explained without examining what is going on inside the boxes. In this section, we shall remind ourselves that nodes are populated by pengines (some of which may be anonymous), and that the Prolog Web is supported by a multi-pengine system.

Consider Figure 9.3, an elaboration of the diagram in Figure 9.1. We have added a client to the picture in the form of a Prolog shell attached to a top-level pengine running a session. On each node, a pengine is running. All three pengines are involved in solving the query posed by the programmer operating the shell. Underlying the process of finding the solutions to `r(X,Y)` is a "conversation" among the client and the pengines running on the three nodes, a conversation in accordance with the PCP communication protocol.

As can be seen in the diagram, the programmer using the shell has entered the query `r(X,Y)`, looked at the first solution, typed a semicolon in order to retrieve a second solution, and is now, we can assume, thinking of asking for a third solution. Solutions appear lazily, one by one.

The pengine running the top-level has direct access to some but not all of the knowledge required to come up with an answer (the clauses in the upper left box). The pengine reads the first clause as an *instruction*: "first find an `X` such that `p(X)`, then spawn a pengine on `http://ex2.org` and ask it for a `Y` such that `q(Y)`, and finally, check that `Y` is larger than `X`. If not, backtrack.". This is a procedural reading of the clause, to be contrasted with the declarative reading given in Section 9.2.

To solve the goal `rpc('http://ex2.org',q(Y))` the top-level pengine has spawned the pengine on `http://ex2.org`, which in turn has spawned one at `http://ex3.org`. The pengine at `http://ex2.org` got back to the pengine at `http://ex1.org` with the binding `Y=2` which together with the binding `X=1`, and since `2>1`, constitutes the first solution. When the programmer typed a semicolon, the shell sent a message `next` to the top-level pengine, and got the second solution back.

At this point, the programmer can either ask for a third solution (by again typing a semicolon), or indicate that he has seen enough (by typing Return). Doing the latter will send the `stop` message to the pengine at `http://ex1.org` that will terminate the query, but not the session. The other two pengines will terminate since they are *linked* to the first pengine.

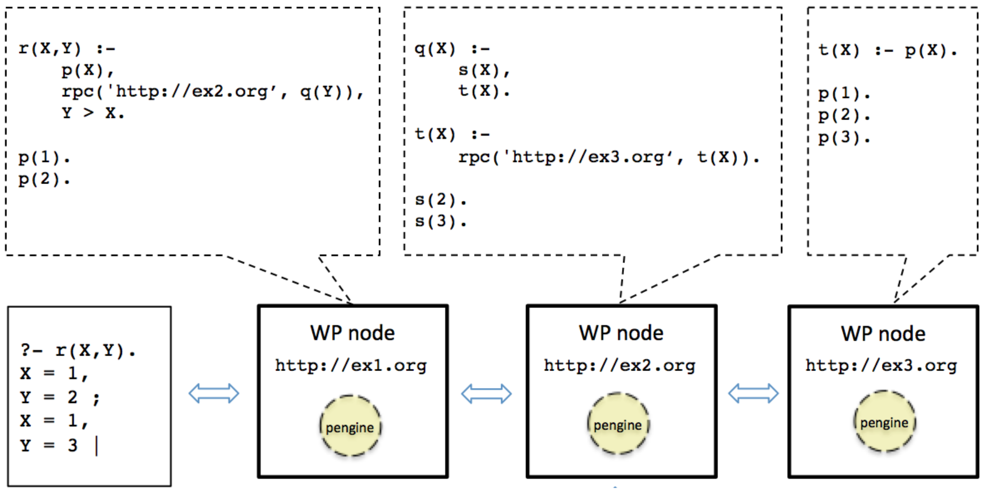


Figure 9.3: A tiny Prolog Web (again). The pengine in the node `http://ex1.org` is running a *session*. This means it will not terminate when the current query has run to completion. The other two pengines will terminate.

What we have here is a simple (and degenerate) supervision tree.

The reason this works is that it is very similar to how pure Prolog explores each branch of a search tree independently from other branches. The analogy is this: Web Prolog, running pure Prolog on the Prolog Web, explores each *node* independently from other nodes. The exploration of a single node typically (but not necessarily) involves the moves that SLD resolution normally goes through when solving a Prolog goal: the depth-first traversal of the search tree and the use of backtracking when failure nodes are encountered. So from the caller's point of view, a call to `rpc/2-3` in the body of a rule is just another subgoal to process, another search tree to traverse, only this time in a Prolog context different from the context of the caller.

One way to sum this up is to say that we are back to normal, well-charted territory, only now we also have an easy way to distribute our programs. There is of course a cost associated with this, one that we encountered and discussed in the context of the non-deterministic but synchronous RPC in Section 3.7 – the caller has to wait idly for the answer bindings to be computed on the remote node and then to arrive over the network. This can also be observed in the example in Figure 9.3. At a given point in time, only one of the three pengines involved is actually working. The others are waiting for something to do. Often, distribution goes hand in hand with concurrency/parallelism, but in this case, it does not. Indeed, when `rpc/2-3` is used, communication between the pengines involved is synchronous by design, and no real parallelism is going on.

As far as the pure Prolog Web in relation to concurrency is concerned, we have to look for it elsewhere. In particular, it is important to remember that when one pengine is waiting for something to do, other pengines on the same node, spawned by other clients connected to it, may be working hard on something. Therein lies a lot of parallelism.

Also, recall that by using spawn, send and receive or the `pengine_*` predicates there are other ways to distribute a program in Web Prolog, while at the same time making good use of any parallellism that can be found and exploited. But this destroys declarativity, is much harder, and involves dealing with concepts such as pengines, pids and messages explicitly. We will return briefly to the concurrent Prolog Web in Section 9.7.

9.4 The power of code injection

Querying and code injection as well as the closely related possibility of bringing code to the data makes the Prolog Web into a web that is *programmable* in a very strong sense of this word, much stronger than the programmability made possible by ordinary web services on the so called Programmable Web. The reason, of course, is that we a dealing with a real programming programming language rather than (say) a relational database.

We have no reason to suspect that code injection – for the purpose of bringing pure Prolog code to the data or data to the pure code – would jeopardise the logical purity of the pure Prolog Web. What matters is the purity of the final composition of the program that is eventually executed by the pengine that is created. This composition is not determined only by the code held by the node, but also by clauses (if any) that the client choose to inject by passing any `src_*` options at pengine creation time. The pure Prolog Web is still programmable. As an example, the program that is listed as belonging to the pengine running at the node `http://ex1.org` in Figure 9.3 may in fact be what the client injected when the pengine was created. Figure 9.3 is simply not clear on this point, and was not meant to be.

9.5 Two levels of abstraction

Looking back at Figure 9.3 there seems to be a sense in which two *levels of abstraction* have appeared. The contents of the boxes with dashed borders form the *level of logic programs*. Below, we find a *level of communicating pengine processes*. Although these levels are clearly related, to the extent that they can be seen as two sides of the same coin, the level of logic programs is more abstract than the level of the multi-pengine system. On the lower level, pengine processes are created and destroyed and while they are alive they send messages to each other. Nothing of this kind takes place on the level of logic programs. On this level, entities such as processes, messages and mailboxes do not exist – they have been abstracted away. (This can be seen most clearly in the definition of `rpc/2-3` in Section 7.6.)

As we see it, the distinction between the two levels is just another way to express the distinction between declarative and procedural that so often appears in the story about Prolog. Or, said differently, "What?" is specified at the level of logic and the corresponding "How?" is handled on the level of abstraction underneath it, i.e. on the level of communicating pengine processes. From a procedural point of view, there is nothing strange about the idea that instructions of the form "in order to do that, do this, then this ... and finally this" may involve talking to a pengine on another node. That such conversations can take place "over the wire" and in a way that preserves the logic as well as the behaviour of Prolog is perhaps the key insight here. It is just a matter of widening the concept of procedural to also cover communication between processes.

9.6 The not-so-pure Prolog Web

When performing reasoning, most pengines use the standard variant of SLD resolution that traverses the search tree depth-first, one branch at a time, using backtracking when it encounters a failure node.[3] Since the exploration of each branch of the search tree can be dealt with in isolation from the other branches, depth-first search is very efficient in its use of computing resources. Pure Prolog is free of uses of assert and retract and other ways to implement global variables. In pure Prolog we can therefore be sure that the only input passed to a procedure is the values of its variable when it is being called. Nothing else can have an impact on the computation. As long as the predicates are present when and where they are needed, this will work also in the distributed case.

Depth-first search may be very efficient but is incomplete since the computation does not terminate if the search space contains infinite branches and the search strategy explores these in preference to finite branches. To ensure its usefulness as a general purpose programming language, a number of impure procedural features have been added to the language and some of them are there to help the programmer to ensure termination.

An important way of optimising and ensuring proper termination of Prolog programs is to explicitly state where the backtracking should be "cut", i.e., committing to a particular choice and never allowing backtracking to choose an alternative, thereby pruning the search space. Sometimes this is sound and does not change the semantics of the program, it just allows us to explain to Prolog when certain things are actually deterministic because of some knowledge we possess but cannot render into Horn clauses; but sometimes it is unsound since it can prune out a correct answer. Thus, cut is an impure operation and it vastly complicates the denotational semantics of what a Prolog program actually means. It also destroys one of Prolog's main attractions, the ability to run programs "forwards and backwards" (e.g. the use of `append/3` not only to concatenate two lists, but also to non-deterministically split one list into two). Having said this, the sequentiality embodied in simple chronological backtracking and the use of the cut for controlling it are likely to be crucial factors in Prolog's success.

The cut is not the only operation contributing to the impurity of full Prolog. The concepts of destructive modification and global variables are alien to the logic programming paradigm, and assert and retract can be used for this purpose. (Erlang has the same issue with the process dictionary, ETS and Mnesia.)

I/O predicates such as `read/1` and `write/1` also contribute to the impurities. It is easy to see that spawn, send and receive is of this kind. Clearly, send and receive, just like the I/O operations in traditional Prolog, do not have any logical interpretation. Indeed, any explicit use of a procedural multi-threading API would break the declarative simplicity of the execution model of logic based languages. This is true of the ISO Prolog Threads standard as well as of our proposed API.

Besides, pengines and other actors are stateful and can be used to implement non-logical destructive assignment, even when avoiding uses of assert and retract.

Resources containing impure constructs such as these takes the Prolog Web further away from pure logic. This by itself does not imply that the Prolog Web is a bad proposition, it only means that it must be judged differently. If the cut, I/O, assert and retract, etc. is what makes Prolog into a practical programming language, then it is also what makes Web Prolog

[3]Note that the term node is here used in a sense different from before.

into a practical web programming language and the Prolog Web into a layer of the Web that is practical.

[TODO: Say something about purity from an internal vs an external point of view (Triska). If we want to, we can implement `append/3` in Python. From an internal point of view it would not be pure and it would not even be Prolog) but from an external point of view it would be pure, and behave *as if* it was Prolog since (if implemented correctly) it would implement the intended relation between three list.]

9.7 Concurrent multi-pengine systems

On the pure Prolog Web, the use of `rpc/2-3` makes processing sequential. This is due to the fact that the required communication with the remote pengine involved is synchronous. As we saw already in Section 3.7 and which is evident in the implementation of `rpc/2-3` in Section 7.6, synchronous communication can easily be built on top of the asynchronous send and the blocking receive.

Asynchronous communication is really the basic mode of communication in Web Prolog and to a large extent asynchronous communication is what allows two or more concurrently running pengines (or other actors) to talk to each other.

Once we allow the spawning of actors and the sending and receiving of messages between them, we open up to the possibility of concurrent programming in the style of Erlang. One approach would be to run an actor that will talk to several other actors, among them pengines, using asynchronous communication. This actor will typically assume the role of a *controller* of the multi-pengine system thus formed. The controller, in turn, is responsible for creating "worker actors" some of which will be executed concurrently with respect to the controller as well as with respect to each other. The traffic between the client, the controller and the worker actors may be guided by a custom protocol, relying not only on standard messages but also on messages sent by means of `!/2` and received by `receive/1-2`.

The worker actors, in turn, may spawn other actors, and thus the resulting multi-pengine system develops into a true (and not just a degenerate) supervision hierarchy.

Of course, another way to run two pengines concurrently is to create and control more than one pengine from the front-end of a web application. We can even make the pengines talk to each other directly if we want, but it may be a bit tricky. It is probably easier to implement a controller of a kind in JavaScript and make the pengines talk to each other only via this controller. If the programs held by the pengines involved are pure, this is indeed a very sensible approach.

More sophisticated controllers, a dialogue manager in a multi-modal dialogue system for example, should probably be implemented in Web Prolog directly, since (possible very complicated) conditions for changing the state of the dialogue manager is much easier and cheaper to check then. Such a conversation will only work if the node that runs the controller is an ACTOR node and if the WebSocket transport is employed.

9.8 Reliability and scalability

In the abstract world of logic, things like execution errors or communication problems are non-existent and there is no need to think about performance. In the real world, where agents such as pengines live, things sometimes do not work out the way they should. Things may be too slow, or the nodes that "house" the pengines may get too crowded at rush hour.

Of course, compared to running a centralised version of a program, there are a few extra problems that may show up when running the same program over the Web, having to do with the nature of the Web (or more generally with the nature of distributed programming) rather than with anything else. We shall have to deal with communication issues caused by high latency, network outages, crashing servers, etc. – problems that occur frequently on the Web. There is not much we can do about them besides raising an error and possibly falling back to a different route. We have made a proposal for how to avoid such issues, based on the `timeout` option, but there should also be a way to raise errors when a 404 occurs, for example.

What is more, time spent in remote shell-actor or actor-actor interaction can be the dominant factor in user-perceived performance of a web application. Some of the backtracking involved in the search for solutions is taking place over the network, and network roundtrips take time – a lot of time in comparison with other computational steps that programs typically perform. Since calling a remote program may involve very many roundtrips during backtracking, the times may add up to a significant slowdown compared to making an in-process call. We need to replace the fine-grained communication with a coarser-grained approach, making the communication less "chatty". By taking advantage of the `limit` option supported by `rpc/3` it is possible to do just that, and thus avoid many roundtrips. A query with n solutions would (normally and by default) require n roundtrips if we wanted to see them all, but if we set `limit` to i, the same query would only require n/i roundtrips.

Use of the `limit` option is fine too from a purity point of view – it has nothing to do with the meaning of the query or the program(s) involved, but must be treated as a *pragma*, a language construct that specifies with which granularity the conversation between the client and the node should be conducted. Passing the option `limit(10)` can be understood as saying: "Send me the answers in chunks of `10`. I will be looking at them one-by-one, but I want them in batches." Although adding the limit pragma to a query will have no effect on the meaning of the query, it can have a significant effect on the performance of running the query over a system of nodes.

A reasonable rule of thumb for setting the value of the limit option is "the higher the latency we expect, the higher the limit should be set". It must not be set too high though, since that might use too much space, or be too generous with CPU time. Either of these will throw an error. By adjusting the settings of the node, the owner can decide *when* such errors should occur, but will not be able to avoid them altogether. Also, latency typically varies with the time of the day on the Web, so to determine the optimal value of the `limit` option is therefore not trivial.

For a multi-pengine system to be able to scale, nodes must be able to spawn very many pengines, creating and destroying pengines must be fast, and the communication among them efficient. A pengine is a kind of actor, and since actors created by the Erlang virtual machine are famous for having exactly those properties, this is certainly one reason (but not the only one) for us to look closely at Erlang, a language which also happens to be very well positioned to take advantage of the so-called hardware multi-core revolution.

Chapter 10

Other Web Prolog profiles

In previous chapters we have extended Prolog into Web Prolog by adding a few extra data types as well as a number of built-in predicates to a subset of ISO Prolog. This resulted in a fairly complete characterisation of what we have referred to as the ACTOR profile of Web Prolog. We have not said much about other profiles, but will do so in this chapter.

We shall introduce, most tentatively, an *hierarchy* of profiles of the Web Prolog language. Three new profiles will be presented and discussed in some detail: ISOTOPE, ISOBASE and RELATION. They differ in both language capabilities and interactional capabilities, and they form an hierarchy in the sense that nodes implementing the ACTOR profile are more capable than nodes implementing the ISOTOPE profile, and nodes implementing the ISOTOPE profile are more capable than nodes implementing the ISOBASE profile. The RELATION profile is the least capable of these four.

10.1 Motivation

As we have explained in previous chapters, and which is demonstrated by the proof-of-concept implementation accompanying this report, Web Prolog nodes having different owners and running at different locations are able to offload computations to each other and to communicate amongst themselves. While this must be seen as an important step towards interoperability, we have only managed to show that two nodes running on the *same* programming platform and having the *same* implementation are capable of this. This is a rather weak form of interoperability. One of our quite ambitious goals with Web Prolog is to increase interoperability between *different* programming platforms supporting *different* host languages. To reach this goal, more work is needed, and some sacrifices will have to be made.

Interoperability in this stronger sense cannot be achieved without *standardisation*. An effort to standardise Web Prolog is indeed also part of the plan (see Chapter 16), but this can only succeed if we can show that at least two independently developed and tested implementations are capable of interoperability. We have a prototype implementation of one system, but sooner or later we are going to need at least one more.

Unfortunately, implementing a Prolog Web node capable of running the full set of example programs given in the previous chapters is not exactly an easy task, and this is what is required of an ACTOR node. Such difficulties have motivated us to define three other profiles which are less powerful and easier to implement than the ACTOR profile. Hopefully, this may encourage developers of Prolog systems to implement nodes capable of contributing to the Prolog Web.

A key insight here is that many web applications can be written with much less support from the Prolog back-end than what is offered by a node implementing the ACTOR profile. Indeed, many web applications are written in HTML, CSS and JavaScript and have only need for what amounts to a "logic server". Since node acting as logic servers are a lot easier to implement than ACTOR nodes, this should send us off in the right direction.

10.2 Overview

An overview of the proposed profiles is presented in Figure 10.1. The figure should be interpreted as a Venn diagram. For example, the reason that owner-defined predicates and the HTTP API are located inside the rectangle marked RELATION is that it signifies that those are features that are supported by *all* profiles, and the fact that WebSocket is placed in the outermost rectangle is intended to imply that the WebSocket API is only available in the ACTOR profile.

10.3 The ACTOR profile – a summary

The capabilities of the ACTOR profile have already been described and thoroughly demonstrated in previous chapters, and this chapter describes the other and less capable profiles mainly in terms of what we can no longer expect from nodes that offer them. A brief summary of the capabilities of an actor node is in order, though.

As has been demonstrated up till now, a Web Prolog node that implements the ACTOR profile allows a (Prolog or non-Prolog) client to send along a program when creating a pengine running a session on the node. The injected code (if any) complements (and may override some predicates in) a node-resident program installed by the owner of the node. The client is then able to query the resulting program using arbitrarily complex goals that may produce, not only answers in the form of variable bindings (or failure- or error messages), but also prompt for input or produce output in an order and with a content as dictated by the program. The ACTOR profile even lends an external client the power to create more than one pengine (or other actor) on a node, and run them concurrently. Again, this is overkill for most applications.

10.4 The ISOTOPE profile

In terms of interactional as well as language capabilities, the "distance" from the ACTOR profile to the ISOTOPE profile is quite big. Neither proper strings nor dicts are supported by an ISOTOPE node. A term such as `"hello"` will be interpreted as in ISO Prolog, and the dict notation will cause syntax errors. Not having to support these data types also means that built-in predicates for working with them do not have to be made available.

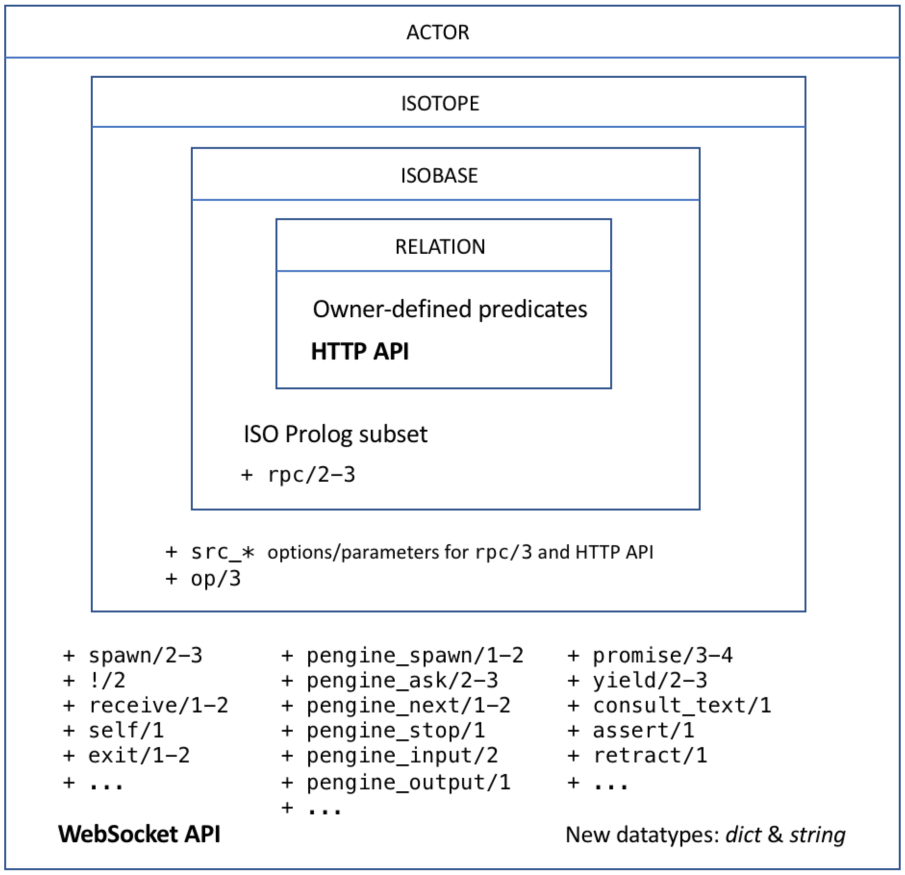


Figure 10.1: A diagram depicting the hierarchy of tentative Web Prolog profiles.

A node that offers the ISOTOPE profile lacks the capabilities that has to do with spawning and messaging. Predicates such as `spawn/2-3`, `!/2`, `receive/1-2`, `pengine_spawn/1-2`, `pengine_ask/2-3`, `pengine_next/1-2`, `pengine_input/2` and `pengine_output/1` are simply not defined by a node with this profile. An attempt to call any of these predicates, in an injected program or in a query, will result in a runtime error.

An ISOTOPE node does not support the WebSocket protocol and therefore none of the interactional capabilities that rely on stateful and asynchronous inter-node communication can be offered by such a node. Only the stateless and synchronous communication offered by the RESTful HTTP API is available, and an ISOTOPE node therefore does not allow a client to create a session on the node – only one-shot queries are possible.

However, source code injection is still permitted, and an ISOTOPE node should therefore offer POST in addition to GET. A POST request such as

```
POST http://isotope.org/ask
query=p(X)
```

```
offset=1
limit=2
src_text=p(a). p(b). p(c).
```

would produce the following JSON formatted response:

```
{
  "type":"success",
  "pid":"anonymous",
  "data":[{"X":"b"},{"X":"c"}],
  "more":false
}
```

As shown in Section 7.6, `rpc/2-3` can be implemented on top of the RESTful HTTP API. So from any node that offers the `rpc/2-3` predicate (and all of them except the RELATION node are required to do so), we can make a non-deterministic remote call to an ISOTOPE node in the following way:

```
?- rpc('http://isotope.org', p(X), [
       limit(10),
       src_text("p(a). p(b). p(c).")
   ]).
X = a ;
X = b ;
X = c.
?-
```

As seen in the above call, the `limit` option as well as the `src_*` options are supported by this profile. Indeed, as we shall see, when calling `rpc/2-3` the `limit` option is always available, regardless of profile, but the `src_*` options are not available in profiles that are less capable than the ISOTOPE profile. Since it requires the WebSocket protocol, the only `rpc/2-3` option not supported by the ISOTOPE profile is the `transport` option.

Recall that `rpc/2-3` implements a form of synchronous RPC. Asynchronous RPC using `promise/3-4` and `yield/1-2` can also be made to an ISOTOPE node, but only from an ACTOR node. The reason for imposing this restriction is that we want to ensure that the less capable profiles can be implemented in languages that do not support concurrency, and that an implemention of `promise/3-4` and `yield/1-2` seem to require some kind of concurrency at the client side.[1]

Note that `consult_text/1` is not available. The only way to inject source code into a remote process is to use the `src_*` options provided by `rpc/2-3` and `promise/3-4`. Predicates for manipulating the dynamic database – `assert/1` and `retract/1` – are not available either. Using them would make little sense anyway, since a session cannot be created.

10.5 The ISOBASE profile

In terms of capabilities, the distance from ISOTOPE to ISOBASE is quite small. There is no change in the data types supported, and only a few built-in predicates that are present in the

[1]This is not strictly true. JavaScript is not a concurrent language but can still support so called *futures*, which is

ISOTOPE profile are lacking from the ISOBASE profile.

For a non-Prolog client only the HTTP API is available here too, and its powers are somewhat reduced since the query parameters for code injection are not supported. Therefore, an ISOBASE node does not need to support the HTTP POST method and a client should use GET. Because of this lack of support for source code injection from the underlying HTTP transport, `rpc/2-3` does not support the `src_*` options either:

```
?- rpc('http://isobase.org', foo(X), [
        src_text('foo(a). foo(b).')
    ]).
Error: The ISOBASE profile does not support code injection
?-
```

An ISOBASE node can, since it does not have to load data or programs dynamically, pull a few tricks in order to optimise query processing, tricks that may be slow to play, but which can be performed offline if dynamic loading is not permitted. (Compiling Web Prolog code into C, for example.) In contrast, a node that implements the ACTOR or ISOTOPE profile, and thus offers code injection capabilities, must do the updating quickly and must probably avoid some of the slower ways of optimising code.

An ISOBASE node does not support `op/3`, the ISO Prolog predicate allowing a programmer to define prefix, postfix or infix operators and specify their precedences. After all, `op/3` is normally used as a directive, so to offer it in a profile that does not also support the injection of source code makes little sense. More importantly, user-defined operators makes writing a parser for Web Prolog more difficult, and the main motivation for having profiles in the first place is to make it easy to implement the less capable profiles.

10.6 The RELATION profile

In terms of language capabilities, the distance from the ISOBASE profile to the RELATION profile is again considerable. A node that supports the RELATION profile is not required to define any built-in predicates *at all*. All predicates that are not explicitly exported by a node-resident program will be treated as undefined. Note that not even control predicates such as `,/2` or `;/2` are available.

A node supporting the RELATION profile would typically be set up to handle very basic queries of the form ?-$functor(arg_1,..,arg_n)$, where $functor$ is a valid Prolog functor and arg_i is a Prolog term.

In terms of interactional capabilities, a RELATION node offers a non-Prolog client the same RESTful HTTP API as an ISOTOPE node, complete with options such as `offset` and `limit`. Consequently, a Web Prolog client can query a RELATION node using `rpc/2-3`.

But since a RELATION node does not offer any built-in Web Prolog predicates, the following will not work even if `p/1` and `q/1` are defined by the node-resident program (and unless `;/2` is defined by the owner):

```
?- rpc('http://relation.org', (p(X);q(X))).
ERROR: Undefined procedure: ;/2
?-
```

just another name for promises.

The main motivation for allowing a node to offer clients as little as this is to make it relatively easy to implement a Prolog Web node in just about any programming language. At the very least, an implementation must be able to parse Prolog terms, as well as to be able to create bindings for the free variables in a query. It does not necessarily need to do any backtracking or recursion but it must *behave* as if it does.

10.7 Some alternatives

The DYNAMIC profile. Here is a synopsis:

```
POST /spawn
...

GET /ask?pid={Pid}&query={Query}&...

GET /next?pid={Pid}&...

GET /stop?pid={Pid}

GET /abort?pid={Pid}

GET /exit?pid={Pid}

POST /send
pid={Pid}&term={Term}
```

We begin the demonstration of the stateful HTTP API by showing that we can use the `src_text` option here too in order to inject source code into the workspace of a pengine when we spawn it:

```
POST /api/pengine_spawn
options=[src_text("foo(a). foo(b)."), format(json)]
```

Unless the source code contains errors, the response is a JSON structure containing a pid:

```
{ "type":"spawned",
  "pid":"f635d35c-d667-11e6-bd95-f36eabd0168b"
}
```

The pid must be sent along when querying the pengine. This will ensure that the same pengine is going to be used during the lifetime of the query and the session:

```
GET /api/pengine_ask?pid=f635d35c-d667-11e6-bd95-f36eabd0168b&query=foo(X)&offset=1
{ "type":"success",
  "pid":"f635d35c-d667-11e6-bd95-f36eabd0168b",
  "data":[{"X":"b"}],
  "more":false
}
```

Since we control a whole Web Prolog *session* rather than just a single query, we can use `assert/1` to assert a clause that will outlive the query:

```
GET /api/pengine_ask?pid=f635d35c-d667-11e6-bd95-f36eabd0168b&query=assert(bar(1))
{ "type":"success",
  "pid":"f635d35c-d667-11e6-bd95-f36eabd0168b",
  "data":[],
  "more":false
}
```

Indeed, after having asserted a clause, we may choose to retract it again:

```
GET /api/pengine_ask?pid=f635d35c-d667-11e6-bd95-f36eabd0168b&query=retract(bar(X))
{ "type":"success",
  "pid":"f635d35c-d667-11e6-bd95-f36eabd0168b",
  "data":[{"X":1}],
  "more":false
}
```

Should we suspect that the query passed to the session will not terminate we can abort the query like so:

```
GET /api/pengine_abort?pid=f635d35c-d667-11e6-bd95-f36eabd0168b
{ "type":"abort",
  "pid":"f635d35c-d667-11e6-bd95-f36eabd0168b"
}
```

Note that this will terminate the query, but not the session. If we want to terminate the session, we can choose to exit the process altogether:

```
GET /api/pengine_exit?pid=f635d35c-d667-11e6-bd95-f36eabd0168b
{ "type":"exit",
  "pid":"f635d35c-d667-11e6-bd95-f36eabd0168b"
}
```

If the client does not exit the remote process, the node may terminate it after a set time with no activity. The pid will no longer be valid and any changes made to the workspace of the process will be lost. At least this is one way of handling it.

But here is also a chance for a node owner to support persistency of a simple kind. Dependent on a node setting, just before terminating all code residing in the workspace of the process can be saved to a file (i.e. “marshalled”) or to another form of persistent storage (e.g. under a name identical to the pid) before the process is finally terminated. Should later (possibly much later) a query request targeting this pid be made, the node can spawn the process again (possibly give it the same pid again?) and inject it with the content of the stored file. This seems to point in the direction of a scheme where a client can hang on to a particular pengine over an extended period of time and still not demand too much resources from the node.

—

A possible advantage with the ISOBASE and ISOTOPE profiles is that they, due to their lack of operations for manipulating the dynamic database and for general I/O, encourage a good programming style focusing on the declarative, logical aspects of Prolog. Those profiles do not rule out impure constructs altogether though, since the cut and non-pure arithmetic are still available.

To encourage an even greater focus on declarative programming techniques, we could imagine a profile – perhaps to be named PURE – that allows only a completely pure subset of Prolog to be used in queries or programs, thus leaving out constructs such as the cut and non-pure arithmetic. We note that the capabilities of such a profile would be orthogonal to the capabilities of the ISOBASE and ISOTOPE profiles. In particular, we see no reason why a pure profile should not be able to support source code injection. A more complex hierarchy of profiles would be the result.

10.8 Discussion

A RELATION node that offers an HTTP API with a fixed set of owner-defined predicates may be compared to a traditional web API and to variants of remote procedure calls on the Web such as XML-RPC, JSON-RPC and SOAP that only offer what amounts to a particular call to a node-resident program. At first blush, they seem to be equally inflexible, but only until we realise that when querying a RELATION node we are dealing with a *relational* language.

When querying predicates with an arity greater than 1, we can ask queries such as `?-r(a,b)`, `?-r(a,Y)`, `?-r(X,b)` and `?-r(X,Y)`. In other words, provided the mode of `r/2` is unrestricted, the API allows us to ask different *kinds* of questions. This is the power of a relational query interface, a power that a more traditional HTTP API cannot provide clients with. Likely, a traditional HTTP API would force us to introduce different endpoints for different kinds of queries, something that makes an API vastly more complicated.

Unless a RELATION node offers a predicate that does not (always) terminate in finite time, it also safeguards against non-terminating queries, a protection that nodes offering the other profiles do not give. Furthermore, making sure that all predicates that are offered by the node are deterministic ensures that no application state has to be maintained by the node and thereby eliminates the need for a cache.

Capabilities offered by an ISOTOPE or ISOBASE node can be compared to *remote evaluation* in the sense of Stamos (1986) and Stamos and Gifford (1990), Thus, remote evaluation is not as restrictive as the querying supported by a relation node, but it is still a useful restriction. The owner of a node may be prepared to accept a remote evaluation over a spawn – a remote evaluation is a one-shot finite computation (made finite by a tight time limit) while a spawn may (legitimately) execute indefinitely (although each query may not).

In addition, such requests have the following properties:

- No outside party, including the client who originally submitted the request, can communicate in any way with the remote computation once it has begun evaluation.
- A process executes in an environment where all capability for intra- and inter-node communication, except for uses of `rdf/2-3`, is not possible, unless the node-resident program offers predicates that allow it.
- They are *safe*, which means they are intended only for information retrieval and should not change the state of the node. In other words, they should not have side effects, beyond relatively harmless effects such as logging or caching.[2]
- They are *idempotent*, i.e. the request can be submitted multiple times without changing

[2] `https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Safe_methods`

the response beyond the one resulting from the initial request.[3]

None of these properties can be guaranteed by the ACTOR profile.

Note that the owner of a node that offers the ISOTOPE, ISOBASE or RELATION profile must take full responsibility for the management of the node's resources. A client cannot terminate a running query, so a mechanism that does that automatically after a set time (or when certain other conditions are fulfilled) must be employed.

RESTfulness has served the Web well, yet the world is moving from mostly stateless connections to increasingly more stateful ones. Having "ongoing conversations" are getting more and more important. Supported by the WebSocket protocol, the ACTOR profile provides us with a way to handle also those actors that do not produce any results, or produce more than one result, i.e. actors that are not really compatible with the request-response orientation of HTTP. Compared to previous attempts to "web enable" Prolog (see Section 1.8) this might be seen as a big step forward.

Another (but less important) reason for defining an hierarchy of profiles is that it allows the owner of a node to offer clients *less* than the node is actually capable of. As long as "important" clients are happy with what is on offer, the node owner may lack incentive for enabling other clients in the wild to run Web Prolog processes on the node. The idea is to enable the owner of (say) an ACTOR node to change a setting so that the node will present itself as less capable than it actually is. The motive for doing so may vary, but taking control over the management of resources (i.e. CPU cycles and memory capacity) might be one of them, security concerns another.

Furthermore, a node owner not willing to offer the full power of Web Prolog queries may want to announce the node as offering the RELATION profile and make sure that only the predicates that are actually needed by a particular application are defined. In other words, it allows a node to *disguise* as a node with lesser capabilites.

[3]https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol#Idempotent_methods_and_web_applications

Chapter 11

Settings and node information

A Web Prolog node is controlled by a number of *settings* which only its owner can change. This section discusses the meaning of such settings.

11.1 Settings for timeouts

It turns out that the PCP protocol can be used to specify at least some settings, and in particular the ones that have to do with various forms of timeouts.

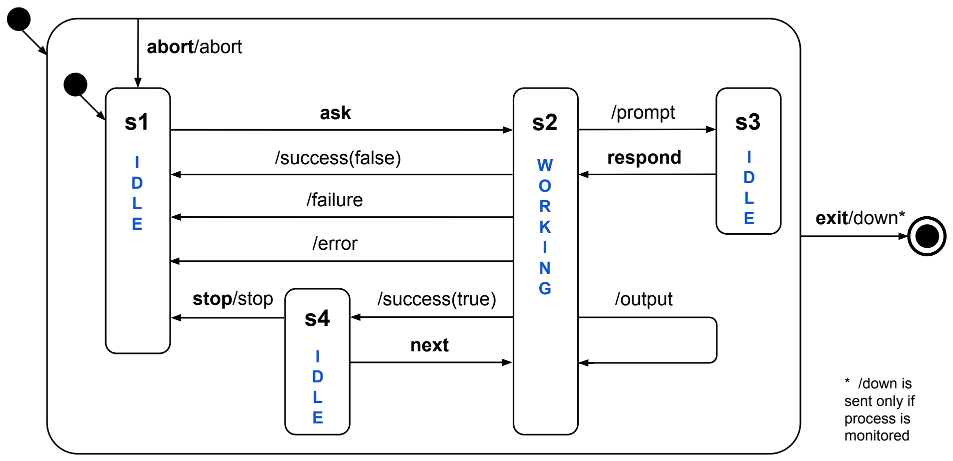


Figure 11.1: Statechart depicting the Pengine Communication Protocol (PCP) for a complete Web Prolog session.

In only one out of four states is the pengine actually doing some real work. The diagram refers to this state as **s2** or the *working* (or running) state. It is obvious that we cannot allow the pengine to remain in this state for too long, so this is where one of our timeouts comes in. If this timeout is exceeded, an error message `timeout_exceeded` is sent back to the caller.

The owner of a node may not want to allow an engine to remain in an *idle* state (also known as a *suspended* state) for too long either. However, should this form of timeout be exceeded it should not lead to an error message, but to exit and termination of the pengine process, and, if the engine is monitored, to a `down` message to this effect being sent to the client.

11.2 Settings for controlling concurrency

An ACTOR node allows a client to spawn an in principle arbitrary number of concurrently running pengines and other actors on a node. This call for a way to limit this number, and a setting seems appropriate.

11.3 Settings for changing profiles

Chapter 12

Prolog Web ≈ Semantic Web?

The conventional web, already in its most primitive form as a Web of Documents (also known as Web 1.0), is *distributed* (running on more than one machine), *decentralised* (running at different geographical locations and/or having different owners) and *open* (anyone can use and contribute). These are some of the traits that the Prolog Web inherits from the conventional web. Since the Pure Prolog Web adds logic and reasoning to the conventional web, we appear to be approaching something resembling the *Semantic Web*, a vision of a machine readable web first proposed in May 2001 in an article by Tim Berners-Lee, James Hendler and Ora Lassila published by Scientific American (Berners-Lee et al., 2001), and in part captured by the following three quotes from this article:

> The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation.
>
> For the semantic web to function, computers must have access to structured collections of information and sets of inference rules that they can use to conduct automated reasoning.
>
> Adding logic to the Web – the means to use rules to make inferences, choose courses of action and answer questions – is the task before the Semantic Web community at the moment.

There has been a lot of activity in the Semantic Web field in the years following the publication of the article. The Semantic Web vision is many-faceted and has prompted research and development in many areas of computer science. A plethora of languages for knowledge representation, query languages and dedicated programming languages have been designed and some of them have been standardised by the W3C. Components such as reasoners, web services, service discovery facilities, user interface frameworks, agent platforms and other infrastructure for the Semantic Web have also been designed and developed but not yet standardised.

As such, the ideas behind the Semantic Web may be both sound and powerful, but the W3C recommendations are complex, and it is often difficult to see how they fit together and how to make use of them when building practical applications.[1] The Prolog Web is much simpler, and we might ask ourselves if it could, in at least some cases, serve as an easier to use replacement for the Semantic Web. Unfortunately, for some Semantic Web use cases, we are not quite there yet. While logic, automated reasoning, openness and decentralisation may be necessary ingredients in a truly semantic web, they surely are not sufficient.

What seems to be lacking is a purely logical approach to the building of ontologies. For this task, Prolog does not have the right expressivity. On the other hand, there appears to be a common understanding now in the Semantic Web community that the word "Semantic" as in "Semantic Web" is a little over ambitious and evokes the wrong expectations. For the baseline functionality and majority of use cases, ontologies may not be needed.

12.1 To have the cake and eat it too

The architecture of the Semantic Web is layered, one layer of languages above another, where each layer exploits the capabilities of the layers below. The layers at the bottom serve also the conventional web. A commonly used illustration of this idea is the so called "Semantic Web Layer Cake", one variant of which is depicted in Figure 12.1.

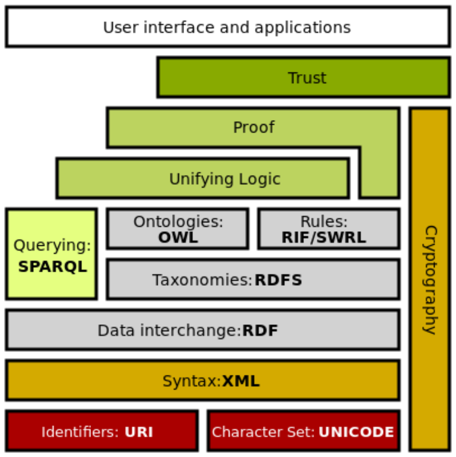


Figure 12.1: The Semantic Web Layer Cake

A diagram of this kind can only give us a vague and somewhat impressionistic view of what the Semantic Web involves, but can at least be used to lend structure to a comparison with the Prolog Web. Starting from the bottom of the diagram, the Uniform Resource Identifier (URI) has played an important part in the development of the Web already from its inception, at first only as a way to locate documents, but on the Semantic Web it can be used to name anything. The Prolog Web, as described so far, may use it it this latter way too, but also to locate programs as well as to uniquely identify answers to queries. Not surprisingly, URIs

[1]This has caused some to give up on the Semantic Web. See `http://inamidst.com/whits/2008/ditching` and `https://www.linkedin.com/pulse/why-semantic-web-has-failed-kurt-cagle`

are normally represented as *atoms* in Web Prolog, but there are also ways to pick them apart and access their components.[2]

Unicode, another necessary part of the conventional Web, may also be seen as a naming mechanism, as it aims to uniquely identify all the characters in all the written languages of the world. Support for Unicode is increasing throughout the world of computing, as the benefits of one common character set are overwhelming when programs are used in a global environment such as the Web. Not all Prolog systems are capable of working with Unicode, but it seems clear that a world wide Web Prolog has to be.

The Semantic Web cake usually has a layer indicating a relience on XML, seen as the syntax of the Semantic Web. In the context of the Prolog Web we wish to somewhat downplay the importance of XML in favour of the Prolog *term* data structure and the JSON format. Our impression is that most researchers would agree that XML is not fundamental to the Semantic Web but should rather be viewed as a *data interchange format* (and, in fact, one out of many). A particular Prolog Web node may well be able to parse and generate XML documents, but a node is not strictly required to be able to do so.

Above the XML layer of the Semantic Web cake we find the layers devoted to semantics proper, where RDF lays the very foundation. From a Prolog point of view, RDF is a predicate with three arguments. [TODO: More here about the work of Wielemaker et al.]

ClioPatria is an RDF-application development framework developed by Wielemaker et al and built on top of SWI-Prolog (Wielemaker et al., 2016). The Cliopatria white paper[3] describes the benefits as follows:

- The single primitive `rdf(Subject, Predicate, Object)` (`rdf/3`) suffices to realise all the basic graph-pattern matching that can be done in SPARQL.
- We can give a name to a query and build complex queries from simple ones. This greatly simplifies maintenance of complex queries.
- Optimisation and unfolding can be used to achieve optimal performance at small cost.
- Instead of the predefined SPARQL functions and conditions, we can apply any Prolog predicate as a condition or function anywhere in the query. We can also introduce other relations (e.g., from an RDBMS) into our predicates.
- The RDF store is tightly connected to Prolog. This allows for arbitrary reasoning with and exploration of the RDF graph at low cost.

To our knowledge, no other general-purpose programming language besides Prolog provides such a clean interface to RDF. We would even encourage the view of the syntaxes of the different RDF serialisation formats as variants of the syntax of (a subset of) Prolog. Here we give an example of triples represented in Prolog taken from the RDF 1.1 primer document:[4]

```
rdf('http://example.org/bob#me',
    'http://...rdf-syntax-ns#type',
```

[2] As it seems independently useful, we may want to introduce the namespace mechanism available in Cliopatria in all Web Prolog profiles.

[3] `http://cliopatria.swi-prolog.org/help/whitepaper.html`

[4] `http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/#section-n-triples`

```
        'http://xmlns.com/foaf/0.1/Person').
    rdf('http://example.org/bob#me',
        'http://xmlns.com/foaf/0.1/knows',
        'http://example.org/alice#me').
    rdf('http://example.org/bob#me',
        'http://schema.org/birthDate',
        literal(type('http://...#date', '1990-07-04'))).
```

In the next layer, OWL, based on description logics, is a subset of predicate logic only partly overlapping with the Horn clause logic subset, and is used for establishing the fixed, controlled ontology vocabularies deemed so important for the Semantic Web. The Prolog Web lacks the concept of ontologies. It is not that we cannot build ontologies in pure Web Prolog. We can, but due to the less expressive Horn clause logic they will less useful. While so called ISA hierarchies can easily be represented – already Kowalski (1986) demonstrated this – features such as range restrictions et cetera are more problematic. (Since Web Prolog is a general-purpose programming language, we can of course use it to write programs capable of reasoning with OWL, but this is a different matter.)

One thing to notice here is that the Semantic Web community makes a clear distinction between knowledge representation languages (such as RDF, RDFS and OWL) and query languages (such as SPARQL), Prolog (including Web Prolog) can, at least to some extent, fill both of these roles. Granted, as a knowledge representation language, Prolog is not very expressive – there are a lot of things we simply cannot say in Horn clause logic, and there are certainly queries that cannot be asked in Prolog used as a query language. However, the point is that the *same* language can be used for *both* knowledge representation and querying. This is a unique and very valuable feature of Prolog, not shared by any other (major) programming language.

Reactivity is of concern also to the Semantic Web and various formats for condition-action rules and event-condition-action rules have been proposed by W3C working groups. In Chapter 8 we described how such rules can be implemented by means of Web Prolog's receive primitive and how they can be used on the Prolog Web. Although they may not be as sophisticated as the ones specified by the W3C, they are most likely much simpler to use and it is immediately obvious how they relate to the rest of the Web Prolog language.

The Semantic Web community does not have much to say about the cake layers labelled Proof and Trust, and nor have we. Recall, however, that as one way to increase (one particular kind of) trust in a Web Prolog code resource, we can at least generate *proof trees*, and we have already demonstrated one way of doing this in Section 7.11. A common criticism agains such proofs is that they tend to grow too big and detailed. In some domains this may be true, but in other domains it would probably work to the satisfaction of its users. As we saw in Section 7.11 a very similar approach might hide what is going on inside a node, and only show the bigger picture.

Furthermore, we would like to suggest that the notion of trust can be interpreted in a very mundane way, where one way to build and maintain trust in a Web Prolog resource is to offer a way to inspect the Web Prolog code stored on remote nodes, e.g. through a browsing or listing facility. The use of good predicate names and good variable names are also important.

As we have already indicated, many variants of the Semantic Web layer cake has been put forward. It appears that some variants are more in line with the theory behind logic programming than others. The two-tower architecture of Horrocks et al. (2005) depicted in

Figure 12.2 presents two possible instantiations of the Semantic Web layered architecture.

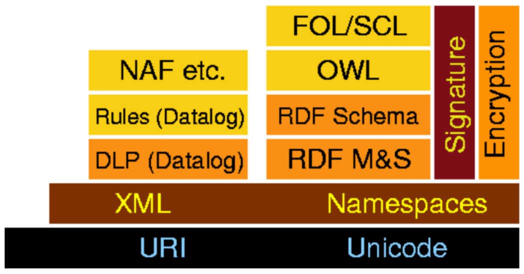


Figure 12.2: The Two-Tower Semantic Web Layer Cake

The tower to the right basically captures the same idea as the cake in Figure 12.1. The tower to the left is different, and is more firmly grounded in the logic programming paradigm proper. Here DLP stands for Description Logic Programs, a subset of First Order Logic which includes the Datalog fragment, but which also has some of the expressive power found in OWL, useful for building ontologies. On top of DLP rules can be formulated and extended with Negation As Failure (NAF) etc. The "etc" here is significant as it indicates other possible extensions of the Horn clause fragment, and in the world of logic programming research there are indeed a lot of those around. In the same spirit as the Two Tower paper, Kifer et al. (2005) argue that a realistic architecture for the Semantic Web must be based on multiple independent, but interoperable, stacks of languages, and they put special emphasis on a stack similar to the tower to the left in Figure 12.2, arguably closer to logic programming than the tower to the right.

It would be interesting to see if the Prolog Web could be developed into a Logic Programming Web more generally. Perhaps the implementation of ProbLog (in Java) can be extended into a node capable of answering ProbLog queries in a Web Prolog compatible way, thus providing probabilistic logic programming. [For demonstration purposes, perhaps Poole's stuff can be used. Or Samer's implementation of Horn Clause Abduction.]

12.2 Prolog Web reasoning engines

The Semantic Web community usually makes a distinction between Semantic Web *languages* and Semantic Web *reasoning engines* (or just "reasoners"). This distinction makes a lot of sense, especially since different reasoners can be used for the same language, and the distinction can of course be made for Web Prolog too. Any machinery for computing with the pure subset of Web Prolog may well be seen as a *Prolog Web reasoner*. Indeed, the humble pengine should be viewed as a reasoning engine for the Prolog Web, although it is not only that, but an *interaction engine* as well, taking care of the necessary communication with other processes.

As we have argued already, logic and protocol form two separate aspects of Prolog. Given the abstractness of the protocol (cf. Section 5.2) it should be expected that it can be given a concrete manifestation not only for Prolog but also for other logic programming languages, and not only for pengines, but also for other reasoning engines.

Recall that since general I/O is non-pure, the only communication possible with a pengine running a pure Web Prolog program is the sending of queries and the reception of answers that the pengine is sending back in response. This is a restricted form of I/O.

There are several possibilities here. One is to keep the restricted communication capabilities of a pengine, but to strengthen its reasoning abilities. This can be done without introducing more language features; adding the ability to do abduction in Horn clause theories would be one example. Using less expressive languages that are still compatible with Prolog is of course also possible. Datalog is one example. Datalog reasoners (based on e.g. the magic sets algorithm) have properties that are interesting in certain use cases while the communication with such reasoners can be done over the Pengine Communication Protocol, and in fact over a subset of it.

12.3 Prolog Web Services and Service Discovery

Semantic Web Services, like conventional web services, are the server end of a client-server system for machine-to-machine interaction via the Web. As we have already seen, the approach we take with the Prolog Web is to specify a set of generic web service APIs that covers everything that we could possibly ask for.

Web Prolog nodes are designed to serve not only the front-ends of our applications but also to be part of an ecology – a section of the Programmable Prolog Web. This means exposing our data and programs to others, to be queried and copied and integrated, often even without our explicit consent or even our knowledge. Others are thus invited to use our APIs and to treat our node as just another part of the Web's infrastructure, as a world-wide assortment of Web Prolog predicates to call in order to get things done.

In contrast to the Semantic Web, the so called *Programmable Web* has made its mark in a big way.[5] Users expect web applications to call out to third-party services as necessary in order to better serve them, and web application developers are no longer content with limiting their applications to users who act with it directly. By allowing code injection the Programmable Web is getting even *more* programmable. Most web APIs only let you look at the data being served in a particular way, usually the way that the hosting site looks at it. By injecting code, a client can choose to look at it in a very different way. TODO: Cite Aaron S.

For locating Semantic Web Services the need for *Service Discovery* becomes apparent. From the article by (Berners-Lee et al., 2001) again:

> Many automated Web-based services already exist without semantics, but other programs such as agents have no way to locate one that will perform a specific function. This process, called service discovery, can happen only when there is a common language to describe a service in a way that lets other agents "understand" both the function offered and how to take advantage of it. Services and agents can advertise their function by, for example, depositing such descriptions in directories analogous to the Yellow Pages.

[5] *ProgrammableWeb* is also the name of a website that publishes a repository of web APIs and has documented over 17,000 open web APIs. See `https://www.programmableweb.com/category/all/apis`.

One can easily imagine a Prolog Web Service Discovery for Prolog Web nodes. An owner of a node may choose to register it in a central service of a kind, nodes can be "crawled" and capabilities, defined predicates, documentation, etc. extracted and registered. In this report we stop short of trying to actually design and implement such a registry, however.

12.4 Prolog Web user interfaces

For the Semantic Web vision to succeed, is generally agreed that research on novel user interfaces is essential (Hachey and Gasevic, 2012). Proposals are for example based on the idea that an ontology can more or less automatically be mapped to a user interface equipped with all kinds of bells and whistles for querying or updating data defined in the terms of this ontology. Our vision for the Prolog Web is far less ambitious, but user interfaces are of course needed here too. We have three things to say about this: 1) An editor in combination with a traditional Prolog top-level shell is certainly not novel, but still a superb user interface to the Prolog Web, 2) Web Prolog in combination with standard web technologies such as HTML, CSS and JavaScript, and communication with the back-end processes (e.g. pengines) performed over HTTP or WebSocket, can be used to fairly easily build novel user interfaces adapted to particular kind of applications, and 3) this is relevant not only for building user interfaces to the Prolog Web, but also to the Semantic Web, as the work on Cliopatria has shown.

12.5 The Prolog Web comes with an architecture

The Prolog Web comes with programs, processes and a flexible infrastructure for computing on the Web. This is something that is lacking from the Semantic Web proposal, or at least does not exist in a form that has been generally agreed. This is worrying to some:

> Without a distributed process infrastructure, Semantic Web applications are left with the typical server/client-download philosophy of the World Wide Web. For data intensive algorithms, this is an inefficient use of resources as it requires the movement of large amounts of data to the algorithms executing machine(s).

Others see some progress:

> Accounting for the decentralized nature of the Semantic Web [...] one can observe an ongoing shift from localized to federated semantic data processing, where independent endpoints provide data, and semantic data applications utilize both local repositories and remote data sources at the same time to satisfy their information needs. In response to this paradigm shift, different federated RDF processing strategies – targeted at different use cases and application scenarios – have been proposed.

With the Prolog Web, this has to a large extent been sorted out. A distributed process infrastructure consisting of nodes (serving pengines) serving answers to queries has been specified in great detail and implemented, and an approach allowing Web Prolog source code to flow in either direction, from the client to the node or from the node to the client, has been worked out. The choice between bringing data to the code, or code to the data, is left to the programmer to make on a case by case basis. Recall, however, that if the remote program is not self-contained, the choice is not even there.

12.6 Steps towards a Semantic Web Programming Language

A commonly held assumption in the Semantic Web community seems to be that for tying all the different Semantic Web technologies together, any general-purpose programming language such as Java or Python would do. There are however well-known problems with this view, due to the so called "impedance mismatch" between imperative (object oriented) programming languages and relational languages (Wielemaker et al., 2016). Considerations such as this have made people suggest that we need dedicated semantic web programming languages.

As for basing such a language on Prolog, searching the Web for information about previous work, we find people that "seriously think about combining SPARQL and Prolog into some nice semantic web scripting language".[6] However, our personal view is rather that Web Prolog does not need SPARQL and that Web Prolog, possibly enhanced with primitives to deal with RDF and friends, can stand strong as a semantic web programming language all by itself.

Interestingly, a W3C community group dedicated to the task of designing Semantic Web programming languages has been formed:[7]

> **The Semantic Web Programming Languages Community Group** A community focused on the adoption of Semantic Web concepts within contemporary and new programming languages. These will incorporate W3C Semantic Web standards for Ontology, Linked data and representations as integral parts of the development tool chains. Particularly the group will aim to 1. Develop new semantically-aware programming languages, 2. Modify existing languages to be semantically-aware 3. Develop design patterns for semantically-aware programming. 4. Develop Ontologies for computer programming concepts to allow inter-lingual sharing of basic and domain-specific algorithms.

We cannot help thinking that Web Prolog fits the aims of this group quite well. Developing an RDF profile for Web Prolog may be one way to push it in this direction. (In Section **??** we propose something like this – not a profile, but a *flavour*.) On the other hand, since Web Prolog pretty well succeeds in *hiding* the Semantic Web behind Prolog, such a step may not be necessary. The most important step was to adapt Prolog to the Web, and this seems to have already provided us with a fairly decent Semantic Web Programming Language.

[6]`https://www.w3.org/wiki/SemWebProgrammingLanguage`

[7]`https://www.w3.org/community/semwebprog/`

12.7 The importance of communities

Programming language communities are typically very devoted to their language of choice, and although the Prolog community is not very big, the people involved in it are quite committed. Our impression is that it is harder to form a dedicated user community around something like the Semantic Web. This may be because the technologies are programming language neutral. It may well be that a love for Web Prolog and the Prolog Web will be easier to grow.

12.8 Attempting a summary

Writing this chapter has been difficult, and there were times that we wanted to avoid doing it altogether. But by proposing something like the Prolog Web we find ourselves forced to consider the relation to its "younger but bigger brother", the Semantic Web. We saw no way of ducking out of it.

It should be clear by now that we do *not* aim for the Prolog Web to *replace* the Semantic Web, or for Web Prolog to replace the current Semantic Web languages and the technologies surrounding them. On the contrary, we hope and believe that we will be able to carve out a place for the Prolog Web alongside the Semantic Web, and that the languages involved can live happily together. This also appears to be consistent with the multi-stack architecture for the Semantic Web that is proposed in (Horrocks et al., 2005) and (Kifer et al., 2005). Interestingly, Kifer et al. (2005) uses the following analogy when arguing against a single-stack architecture for the Semantic Web:

> While a single-stack architecture would hold aesthetic appeal and simplify interoperability, many workers in the field believe that such architecture is *unrealistic* and *unsustainable*. For one thing, it is presumptuous to assume that any technology will preserve its advantages forever and to require that any new development must be compatible with the old. If this were a law in the music industry (for example) then MP3 players would not be replacing compact disks, compact discs would have never displaced tapes, and we might still be using gramophones.

The authors do have a point, and we would like to draw on the gramophone analogy even further. Quite a few people (usually referring to themselves as "audiophiles") *are* still using gramophones, for the good sound and perhaps for the nice smell of a vinyl record.[8] We also note that, for reasons of compatibility and interoperability with more modern technology, the gramophones sold today can usually be plugged into a computer via a USB port. We are thus tempted to think of programs written in Web Prolog as the vinyl records of the logic programming world, and of pengines and nodes as the gramophones – updated for compatibility and interoperability with the Web – on which to "play" the programs. And to continue

[8]According to Wikipedia, "From the 1990s to the 2010s, records continued to be manufactured and sold on a much smaller scale, and were especially used by disc jockeys (DJs), released by artists in some genres, and listened to by a niche market of audiophiles. The phonograph record has made a niche resurgence in the early 21st century - 9.2 million records were sold in the U.S. in 2014, a 260% increase since 2009. Likewise, in the UK sales have increased five-fold from 2009 to 2014."

with the analogy, there are things that you can do with vinyl records and gramophones, like *scratching*[9] for example, which simply is not possible with newer technologies, and there are things you can do with Prolog, like *programming* for example, which the usual Semantic Web stack of languages does not allow. So while it is true that Prolog is old technology, and while Prolog's tight integration of knowledge representation, querying and general purpose programming comes with its own set of problems, this integration is at the heart of Prolog. Having it any other way, it would no longer be a Prolog Web.

[9]https://en.wikipedia.org/wiki/Scratching

Chapter 13

Ways of implementing Web Prolog

A major motive behind the proposed design of the Web Prolog language and the architecture for the Prolog Web is that we see it as a way to allow the many Prolog systems currently in existence to talk to each other. As an extension of this idea, we also propose that, as long as we can wrap an interface consistent with Web Prolog around them, such conversations might involve not only Prolog systems, but also many non-Prolog systems, such as (deductive) databases, other logic programming systems, as well as systems running other programming languages such as Erlang, Java or JavaScript.

This is what the diagram in Figure 13.1 tries to convey.

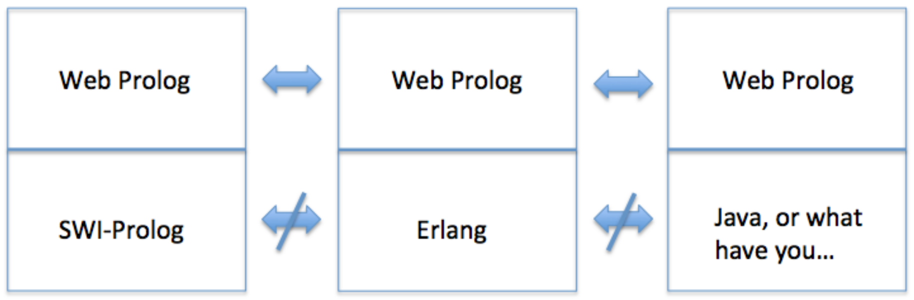


Figure 13.1: Three platforms. [TODO: This diagram should perhaps include another Prolog system as well as JavaScript too. Also, clear up the confusion between wrapping and embedding!]

One big advantage with our proposal is that if you are a Prolog implementor, most of the necessary implementation work has already been done. Some really excellent Prolog implementations exist out there, and a majority of them more or less conform to the subset of ISO Prolog that Web Prolog is based upon. Wrapping them in code that implements a Prolog Web node is then all that is needed. Depending on from where you start, targeting an AC-

TOR node may still be a lot of additional hard work, while a node with ISOTOPE or ISOBASE capabilities should be easier.

Using Web Prolog as an embedded scripting language would allow these systems to exchange information using Web Prolog as an interlingua, allow clients to script pengines and other actors running on them, and allow us to integrate all these systems into a truly programmable (or should we say "scriptable") Prolog Web.

Like many other scripting languages, Web Prolog lacks some of the capabilities of general-purpose programming languages, such as file I/O, access to the OS and a foreign language interface, but since a Prolog Web node is implemented on top of a general purpose programming language it should be straightforward to delegate the need for such operations to this language. A programmer wishing to implement an application that needs to perform e.g. file I/O needs to be knowledgeable in the host language. For some nodes, it probably makes sense to offer an interface between Web Prolog and the host language. In the case of SWI-Prolog, such an interface is obviously already there.

There are consequences for the portability of some applications. An application written in Web Prolog is portable between different node implementations. Indeed, an application written in pure Web Prolog will not only be portable in the normal sense, but we will likely also be able to run it on a node that we do not own nor manage. However, an application written in a combination of (say) Erlang and Web Prolog may not be easy to port to (say) a node written in SWI-Prolog. And since an Erlang powered back-end may have been chosen for a good reason, porting it to a node written in Erlang may not be a good idea, even if it was possible.

The conformance demands that we should place on Web Prolog systems need to be very strict. Porting programs between platforms takes place offline, and with a bit of work problems can usually be taken care of. Interoperability is more of a runtime phenomenon. Conformance issues that arise at runtime cannot be easily fixed.

13.1 Porting the SWI-Prolog implementation

In this report, we have sketched an implemention of a Web Prolog ACTOR node in SWI-Prolog. Unfortunately, most of the code is basically non-portable to other Prolog systems. Many of the required libraries and features are shared with at least one other Prolog implementation, but none is capable to support the full range.

Below we summarise the main problems.

- The scale of the involved Prolog libraries demands for a closely compatible Prolog module system. Probably only SICStus and YAP can be used without significant rewrites.
- The HTTP server libraries are heavily based on C code that interacts with the SWI-Prolog foreign language interface to Prolog streams. YAP has copied the low-level libraries and is capable to run (an old version of) these libraries.
- As far as we know, SWI-Prolog is the only Prolog system that offers a WebSocket library.

- The primitives for spawning pengines and other actors depend on the multi-thread interface. The SWI-Prolog thread API is also provided by YAP and XSB.

From the above list it should be clear that a fully functional port of the SWI-Prolog ACTOR node to another Prolog system is not immediately feasible. YAP probably comes closest but still requires a significant amount of work.

13.2 Implementing specifications from scratch

We need to switch our attention from the porting of the SWI-Prolog implementation of a Web Prolog node to an implementation of the *specification* of the Web Prolog language and its runtime system. Also, recall that there is more than one specification – one for each profile – although they are related to each other. To implement one of these specifications should be possible to do in more than one way. Here we sketch a few alternatives:

- Some Prolog systems do not use the ISO Threads standard to implement concurrency. SICStus Prolog and Ciao Prolog are cases in point. Presumably, at least some of the Web Prolog profiles can still be implemented using alternative constructs provided by these dialects. [Clark et al. (2001), section 7 compare communication in Qu-Prolog with that of SICStus-MT, BinProlog, CIAO, Erlang, Mozart-Oz and April.]
- Some (most?) Prolog systems could probably be deployed on the Web using OS based sandboxing where every client talks to a private copy. There are several language neutral environments that would let us do this. At least some profiles of Web Prolog could probably be implemented using this approach.

13.3 Implementation for browsers

As it now stands, as a web programming language Web Prolog cannot compete with JavaScript in one important respect: web browsers support JavaScript, but not Prolog. This could change of course. Building an implementation that transpiles a subset of Web Prolog into JavaScript, or compiles it into WebAssembly,[1] is most likely feasible.[2] It needs to be able to spawn pengines or other actors on external nodes, and it must be able to communicate with them using send and receive in a completely asynchronous fashion. Note that we probably do not want Web Prolog running in a browser to act as a full-blown node, i.e. no external entity should be allowed to spawn a process on it. In all likelihood, we would need a completely new specification of a browser profile.

```
<html lang="en">
 <head>
  <script type="javascript" src="/web-prolog.js">
  <script type="text/x-web-prolog">

    :- initialization(run).
```

[1]https://en.wikipedia.org/wiki/WebAssembly
[2]Here is an implementation of Prolog in JavaScript: http://tau-prolog.org/

```
      run :-
          ...
          receive({
              ...
          }).

   </script>
  </head>
  <body>
    ...
  </body>
</html>
```

In this way, we can avoid the explicit use of WebSockets and XMLHttpRequest altogether.

13.4 Implementation on top of other programming systems

Dedicated Prolog systems such as SWI-Prolog or SICStus Prolog are usually written in fairly low-level languages such as C or C++ but one occasionally come across Prolog implementations also in more high-level languages such as Scala, Python, Haskell, Erlang, Ruby, Lisp or Scheme.

Such implementations may be nothing more than the result of programming exercises – perhaps by a student or a hobbyist wishing to implement a couple of somewhat challenging algorithms such as unification and backtracking. But they may also be the results of a real need felt by a programmer confronted with a problem – the need for a “rule engine” or “logic engine”. Then, rather than move an application to Prolog, the programmer decides to implement Prolog in the language of his application. Some do it for the fun of it.

Interestingly, there is already a Prolog implementation in Erlang, written by Robert Virding (who seems to fall under the do-i-for-fun category of programmers), one of the people behind Erlang.[3].

An implementation of a Web Prolog node in Erlang would be interesting since it would most probably have a performance profile different from the implementation in SWI-Prolog. Prolog is likely to be slower in Erlang than in the implementation in SWI-Prolog, whereas the super fast lightweight processes in Erlang have other advantages, making it scale better to very many simultaneous users on a network. Besides, Erlang is particularly famous for extremely efficient implementations of Web-related technologies such as Web servers (e.g. Yaws and Cowboy) and this could also be a distinctive advantage of an Erlang implementation.

[3] `https://github.com/rvirding/erlog`

Chapter 14

How to revive and rebrand Prolog?

The wordings of the heading of this chapter seem to make the (in our opinion) false presupposition that Prolog is not already in many ways a great programming language. But if greatness of a programming language is measured in number of practitioners, then it is, unfortunately, true. In this chapter we will take a stab at the problem of popularising Web Prolog. We have quite some way to go. Estimates have it that there are 18.5 million software developers in the world (including 7.5 million hobbyists).[1] Let us be optimistic here, and assume that a couple of thousand developers are using Prolog. This is nowhere near greatness.

Also note that while overall the number of developers have grown dramatically since the mid-nineties, the number of Prolog developers have fallen. However, the fact that the language refuses to die should tell us something and give us hope for a future revival.

When we guess that there may be perhaps a couple of thousand programmers in the world, we do not count people using Prolog as part of their education in university computer science courses. It is of course a good thing that teachers still see Prolog as interesting enough to teach it for a couple of weeks in order to introduce a new and different paradigm, but students taking such courses should not be counted as developers. They do not learn much anyway, since most courses are too short.

Is it too difficult to learn?

While it may be true that not all people are wired for Prolog, we do think that some are. Likely, they can be found among the very best of JavaScript programmers. Therefore, it might be a good idea to actively seek out and educate these people, showing them what can be done with a combination of JavaScript and Web Prolog.

It ought to be possible to make Prolog popular again. A nice thing about popularity is that it can be measured.

[1] `https://www.infoq.com/news/2014/01/IDC-software-developers`

14.1 Some recent statistics

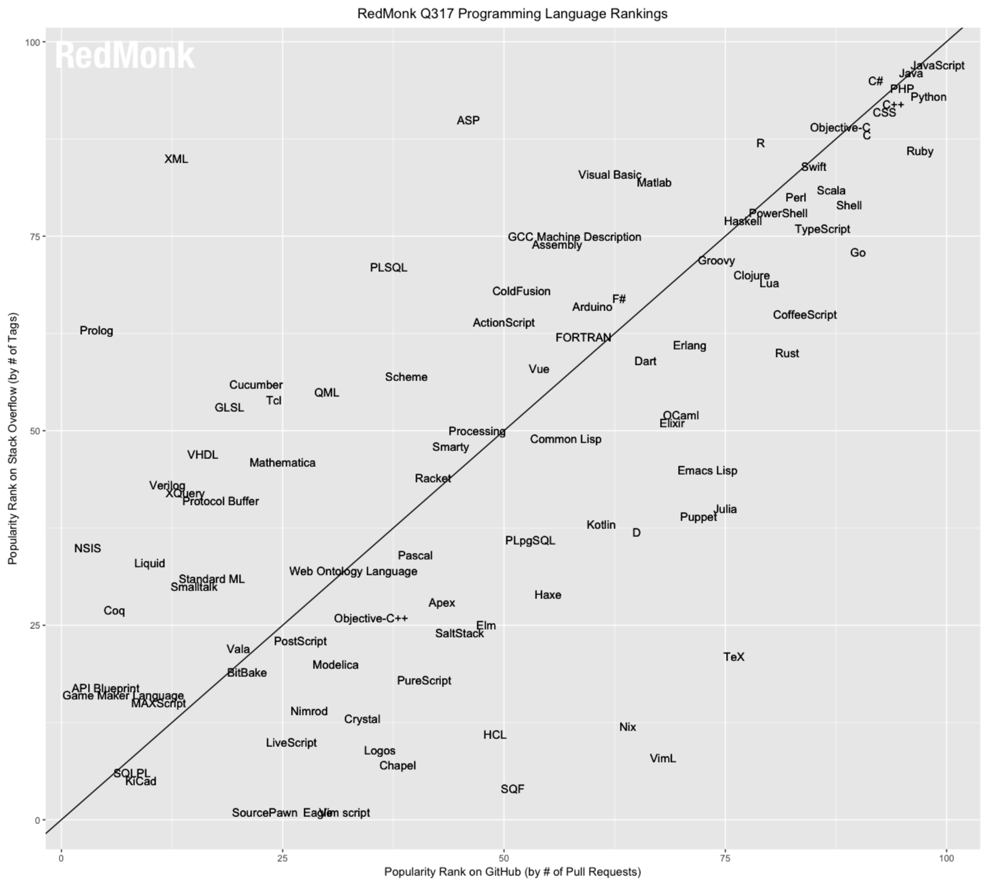


Figure 14.1: The Redmonk language ranking – June 2017

14.2 The crisis of Prolog

Van Roy (1994) surveyed the major developments in sequential Prolog implementation during the period 1983-1993. In a section about their role in the marketplace, he writes:

> As measured by the number of users, commercial systems, and practical applications, Prolog is by far the most successful logic programming language. Its closest competitors are surely the special-purpose constraint languages. But it is true that logic programming in particular and declarative programming in general remain outside of the mainstream of computing. Two important factors that hinder the widespread acceptance of Prolog are:

> - Compatibility. Existing code works and investment in it is large. Therefore people will not easily abandon it for new technology. Therefore a crucial condition for acceptance is that Prolog systems be embeddable. This problem has been solved to varying degrees by commercial vendors.
> - Public perception. To the commercial computing community, the terms "Prolog" and "logic programming" are at best perceived as useful in an academic or research setting, but not useful for industry. This image is not based on any rational deduction. Changing the image requires both marketing and application development.
>
> The ideas of logic programming will continue to be used in those application domains for which it is particularly suited. This includes domains in which program complexity is beyond what can be managed in the imperative paradigm.

Today, the claim that declarative programming in general remain outside of the mainstream of computing is probably false. At least Haskell, Scala, Clojure, Erlang and Elixir are fairly mainstream and must be counted as completely declarative (Haskell), or as at least having a core that is declarative (the rest). Indeed, functional programming is having something of a heyday.

It is furthermore not certain that it is as important that Prolog systems are embeddable as it used to be. For many applications, calling Prolog over HTTP or Websocket would work just fine, and for web applications it is of course the only option. And when embedding still is important, nowadays Prolog can be embedded in C, C++, or other languages to provide logic based services in a larger application.

Thirteen years after the publication of (Van Roy, 1994), in a piece entitled *A Wake Up Call for the Logic Programming Community* published in the December 2007 issue of the ALP Newsletter,[2] Tom Schrijvers wrote:

> In the six years that I have been doing research in the Logic Programming community, I have met a lot of nice people and heard about a lot of interesting research ideas. However, the community itself isn't exactly thriving and LP has a serious PR problem. It's my fear that if we happily continue along our current path, then there will be little left of LP and its flagship Prolog in a couple of years.

A year later, at ICLP 2008, (Schrijvers and Demoen, 2008) continue:

> The concerns raised in the ALP Newsletter have been known for years, but it takes repeated raising of voices to make the community more aware of the issues: one big problem is that the Prolog community has been too fragmented. This is true at several levels: at the system side one notes that there are too many incompatible Prolog systems around, their structure and libraries are different, their implementation technology is very diverse, and their aims are difficult to reconcile (research or industrial deployment).

[2]`https://dtai.cs.kuleuven.be/projects/ALP/newsletter/dec07/content/Articles/tom/paper.pdf`

Now, ten years later, the situation has perhaps improved a little. But only a little. The Prolog community is still fragmented. The challenge still exist.

14.3 Some attempts at solving the crisis

Happily, Schrijvers and Demoen (2008) could also report that progress had been made, that rekindled their faith in a prosperous future for Prolog. One way of uniting the Prolog community – the Prolog Commons[3] – had already been tried, with some success. Alas, at the time of writing the present report this initiative seems to have lost momentum.

Another important stab at the problem of improving the portability of Prolog programs was taken by Wielemaker and Costa (2011). Although advice important to any effort directed towards making any Prolog system conform to other systems were provided, the focus of the authors was on improving the compatibility between Yap Prolog and SWI-Prolog.

With his LogTalk system, Paulo Moura seems to have had ambitions similar to ours. Logtalk provides a compatibility layer with portable libraries supporting B-Prolog, CxProlog, ECLiPSe, GNU Prolog, Lean Prolog, Qu-Prolog, SICStus Prolog, SWI-Prolog, XSB, and YAP. Through his work on LogTalk, Moura has found hundreds of portability problems and bugs in many Prolog compilers and encouraged developers to correct them. The problem with his approach from our perspective is that LogTalk may not be suitable as a web programming language. What is missing is an approach to distribution and the communication necessary for interoperability between Prolog platforms.

14.4 Our own proposal

Instead of focusing on the portability of as large a fragment of Prolog as possible, we suggest that the community should agree on a somewhat smaller fragment, and aim also at interoperability in the sense that the present report suggests. If we can make the many different Prolog systems talk to each other over the Web, we believe that the Prolog sub-communities will start talking to each other too, more than they currently do.

Note that the approach that we suggest will allow Prolog systems to *stay incompatible* on many levels and in many ways, as long as they stay compatible on the level of Web Prolog. All sorts of extensions to be made to a Prolog system are allowed, as long as they do not interfere with the Web Prolog layer. Through calls to the host language, a Prolog Web node based on such a platform will still be able to offer the functionality of such extensions and their presence might be what makes a developer choose it over other nodes.

14.5 Key factors for popularity

According to one source on the Web, these are the key factors to consider:[4]

1. Being a default language for a popular ecosystem;

[3]http://prolog-commons.org/

[4]`http://programmingzen.com/so-you-want-your-programming-language-to-be-popular/`

2. Being backed and promoted by a tech giant;
3. Having a large standard library and/or targeting a popular VM (e.g., JVM, unless its a system language);
4. Fully embodying a new paradigm shift;
5. Being very useful in a particular domain, like Ruby did through Rails back in 2005;
6. Having great documentation, guidance for newcomers, tools, editor integration, and in general removing any instrumentation obstacle that makes getting started with your language difficult;
7. Fostering a welcoming community that encourages sharing, marketing, and evangelism;
8. Having a somewhat familiar syntax that is easy to read, write, and teach;
9. Being active and developed for several years;
10. Providing technical innovations that lead to productivity and more maintainable code;
11. Luck.

As the author of these key factors points out, the first two are out of the question for most languages, and even then, few languages will meet all of the remaining requirements. Even embracing just a few of these should, however, be enough to grant a language a chance at becoming mainstream. So how well do Web Prolog satisfy these requirements?

As for being the default language for a popular ecosystem, we are trying to *build* an ecosystem, namely the Prolog Web, for which Web Prolog is meant to be the default language. Whether the Prolog Web will become popular remains of course to be seen.

We do not at this time foresee Web Prolog being backed by a tech giant. But who knows, at least one leading engineer at Google thinks we need more web programming languages.[5] With languages such as Elm[6] and ClojureScript[7] functional programming is moving in this direction. When it comes to web logic programming languages, there isn't much competition. It does seem to be necessary to create a version of Web Prolog that runs in a web browser however.

Web Prolog does not target a specific VM. Any VM or any programming language that can possibly support an implementation of a profile of Web Prolog is targeted. After all, this is the whole idea behind a language aiming at interoperability between different programming systems that may have different VMs or no VM at all.

As for fully embodying a new paradigm shift, we could possibly claim that web logic programming might be counted as a new paradigm, based on the two older paradigms of logic programming and actor programming. However, we actually think that it is better to think of Web Prolog as a multi-paradigm programming language.

[5] `https://www.pcworld.com/article/2362500/google-engineer-we-need-more-web-programming-languages.html`

[6] `http://elm-lang.org`

[7] `https://clojurescript.org`

We also believe that Web Prolog, just like Ruby on Rails was, is very interesting for the web domain. After all, creating a domain-specific variant of Prolog is our main focus.

Documentation of the Web Prolog built-in predicates can be pooled from the documentation provided by existing Prolog systems. Guidance for newcomers in the form of textbooks, many of which are free, exists. As for removing other obstacles that makes getting started with Web Prolog difficult, SWISH must be mentioned.

Currently, the most active Prolog community is arguably the one that has formed around SWI-Prolog. It is indeed a welcoming and encouraging community.

As for having a somewhat familiar syntax that is easy to read, write, and teach. Whether this applies to Web Prolog depends on where you are coming from. For an Erlang programmer, Web Prolog will present few problems. More on this will be said in Section 14.7.

SWI-Prolog has been developed over a period of thirty years.

As for providing technical innovations that lead to productivity and more maintainable code, this is what Prolog and technologies such as constraint logic programming is for.

And yes, we would certainly need luck as well, but as the Spanish proverb says, luck comes to those who look after it.

To these key factors, we are going to add another one, namely the existence of a standard. We shall spend the whole of Chapter 16 discussing the prospect of a new standardisation effort.

[TODO: Paul Graham has written about popularity too]

14.6 Rebranding Prolog

> *Rebranding* is a marketing strategy in which a new name, term, symbol, design, or combination thereof is created for an established brand with the intention of developing a new, differentiated identity in the minds of consumers, investors, competitors, and other stakeholders. – *Wikipedia*

Here is an interesting quote from a Hacker News user that calls himself “gruesom”:[8]

> With programming languages, I’ve come to the conclusion that ”If X is so awesome, why is nobody using it?” is the wrong question to ask. Or rather, it’s a fair question, but the answer doesn’t depend on X. It is overwhelmingly a matter of network effects and social proof. What matters is what everyone else is using.
>
> A programming language has a window of opportunity to gain mindshare while it’s new. Once that window passes, the odds are hugely against it. It’s true there are a few technical factors, for example that older implementations lack the infrastructure to work with newer technologies, but such things can be improved. The dominant factor, the real barrier, is social psychology. This is obscured by our tendency to retrofit plausible-sounding reasons to our choices (”X is slow” or ”it doesn’t have libraries” or whatever).
>
> This does suggest a strategy for reviving an old language: rebranding. First, you need a new implementation. Trying to convince people to use the old one

[8]https://news.ycombinator.com/item?id=3900047

> is like trying to get them to watch an old movie. George Lucas didn't promote Kurosawa, he made Star Wars. Second, there has to be a hook, something to convince us that this really is a new language, modern and with the times. So, light sabers. That's critical for opening a fresh window of opportunity instead of getting pegged to the old, long-closed one. It needn't be the most important thing, but it can't be fake either; it must be real enough or no one will take the new brand seriously. Without a convincing reason for why we weren't using X before, the new window will never open. Once it opens, then and only then do the beauty and power of X get their chance to bond with the user. In the Star Wars analogy, that would be the characters and the story the deeper things that came from Kurosawa and ultimately from ancient archetypes. They'd never have gotten the chance to kick in if Star Wars hadn't got the audience in the theater. Once they did, though, then you had Star Wars fans for life.
>
> With Clojure, the hook was Java libraries. Actually, there were three hooks. The others were concurrency and sequences. But Java libraries is the one that worked. And now there's a modern Lisp again!

With Web Prolog, the hook is the Prolog Web with its promise of interoperability between different Prolog systems. Not every new or rebranded programming language comes with a novel extension of the Web, so from a marketing perspective it makes sense emphasise both. Actually, there are three hooks here too. The others are pengines and a concurrency programming model more sensible than the current one – a model with a proven track record thanks to Erlang and Elixir. Also, as soon as web developers start to see how simple it is to use Prolog instead of a relational database, this might turn out to be another hook.

14.7 Learning Web Prolog

People already familiar with both Prolog and Erlang will have an easy task picking up Web Prolog. Prolog programmers will be able to use Web Prolog out of the box – they only need to learn new things if they want to delve into concurrent programming in Web Prolog, and there are Erlang textbooks from which the principles as well as some of the practice can be learned. Erlang programmers not familiar will Prolog will have some new things to learn, but the close affinities between the two languages are likely to be of help here too. Furthermore, there are plenty of really good Prolog textbooks around, and most of them can be downloaded for free.

This is our line of reasoning: The closer Web Prolog appears to traditional Prolog, the easier it should be to raise some interest for the language in the Prolog community, as long as it can also offer something over and above what traditional Prolog offers (and it can – there are those hooks). In other words, we must not lose our current users in an attempt to find new ones. In addition, staying close to traditional Prolog also makes it possible to reuse some of the old textbooks written for teaching Prolog. A lack of teaching material most likely delay the acceptance and adoption of a language and likely hamper the growth of the community surrounding it.

By the same kind of reasoning, the closer it appears to Erlang, the easier it will be to raise some interest for the language in the Erlang community. Hence our decision to borrow as much from Erlang as possible, notably the syntax and semantics of selective receive. Luckily, it turned out that by just defining an infix operator for the `when` guard, a syntactically pleasing version of the receive operator could be designed. We also decided to use the send operator `!/2` in the same way as Erlang is using it, by defining it as another infix operator.

Presenting the language in a web based IDE such as SWISH, allowing people to play around with examples right in the browser, is also likely to accelerate the adoption of it.

14.8 Web Prolog for teaching advanced CS concepts

Some programming languages have a reputation for being easy to learn and are for this reason advocated as teaching languages. When it comes to basic CS concepts, Python is probably the most used language for the purpose of teaching imperative and object-oriented programming.

For teaching functional programming and advanced CS concepts such as concurrent and distributed programming, Erlang seems to be used at quite a few university CS departments, and for teaching logic programming, traditional Prolog is the only game in town.

We note that Web Prolog can be used to teach *both* logic programming and actor-based concurrent and distributive programming. Being able to do it in the same online development environment must be seen as an advantage.

14.9 We Are the Prolog Web!

As pointed out by (Schrijvers and Demoen, 2008), the many different Prolog systems has resulted in a situation where they all have their own dedicated user communities. This is unfortunate, and the question thus becomes: How can we defragment the Prolog community?

So far in this report, the term “Prolog Web” has been used only to name the technological artefact built from nodes running Web Prolog. This should not stop it from also acquiring another meaning. In the sprit of *We Are the Web*[9] we must acknowledge the fact that the Web is much more than a network of machines and the software running on them. The Prolog Web is also a web of Prolog *users*. The Prolog community could be strengthen a lot if systems allowed us to easily share not only programs, but also processes running them as well as web-based demonstrations to show them off. Other tool for sharing include chat systems, presentation tools and blogging frameworks, tools that can be written in a combination of HTML, CSS and JavaScript and served by a Web Prolog node.

It becomes social computing, not only as it appears in multi-pengine systems, but also in the Prolog community. In times like these, when people share ideas, share programming tricks and share code on the Web, we do believe that the willingness of the people in the community to also share computing power may also be there.

In short, we believe that the Prolog community may be able to defragment itself by having not only a common Web Prolog language but also a Prolog Web in common – a social network of Prolog programmers and of pengines and other actors. The dependency is mutual, because if the Prolog community is not willing and able to show that the Prolog Web is useful, then no one can.

[9]`https://www.wired.com/2005/08/tech`

This would presumably work as a service to the logic programming community at large, many members of which will be able to present their work through APIs that expose their research systems to the Prolog Web for others to easily try out and perhaps also use, regardless of if it is written in Prolog or not. Indeed, it might inspire logic programming researchers to develop proof-of-concept implementations that work well with the Prolog Web, and maybe some of the things that they do can be further standardised in new profiles of Web Prolog. In the best of worlds, this might cause the Prolog Web to evolve organically.

Announcing his new Web Science Research Initiative (`http://www.webscience.org`), Tim Berners-Lee declared he wished to attract multidisciplinary researchers to study the Web as a technological *and* social phenomenon.

[Check out Social Machines]

Chapter 15

User groups and use cases

15.1 Database driven web applications

Prolog is an elegant language for database queries. In fact if one constrains Prolog programs to use only atoms, integers and real numbers (no lists or complex terms) and disallows recursive definitions, one gets a database language that is equivalent to a powerful subset of SQL. Therefore, in at least some circumstances, Prolog can replace a relational database.

15.2 A web-based conversational agent

A *conversational software agent* is a program that is able to interact with a user, through typed text, spoken language and/or perhaps other means, and is capable of responding in an intelligent way to the user's requests. Building interesting and useful conversational agents is not an easy task as there are quite a few requirements that we want to impose on them. Since they are *agents* we expect them to interact with the environment, and perform (communicative) actions either on their own initiative (i.e. being proactive) or in response to (communicative) events which occur externally (i.e. being reactive). Since they are *conversational* agents we require them to understand a language, could be Prolog or it could be a natural language such as Dutch, English or Swedish. Of course, understanding a natural language comes with other requirement, such as the ability to parse and to semantically and pragmatically interpret the user's (often fragmented or 'ill-formed') contributions as well as to generate sensible natural language responses in the course of a dialogue. Like with any *software* we expect implementations to be modular and easily portable from one domain and/or task to another.

Conversational Agent development environment. A tool for developing spoken dialogue systems/chat bots – including speech recognition and synthesis (using the web speech API), and perhaps an animated face that can be controlled from Prolog. Conversational agents aka chat bots are quite popular these days and DCG and KR in Prolog are powerful means for building them.

For the Semantic Web, conversational agents have been on the table all the time, witness `http://www.nature.com/nature/webmatters/agents/agents.html`

JavaScript is the foundation of all interaction programming for the web.

15.3 Defining customised protocols

The idea would be to spawn a Prolog process from JavaScript running in the browser. This process is not a pengine, just an ordinary actor. It would serve as dialogue manager (DM), the component of a dialog system responsible for the state and flow of the conversation, and often the only component that is stateful.

```
prompt_and_listen(ClientPid) :-
    ClientPid ! speak('What can I do for you?'),
    receive({
        recresult(NbestList) ->
            tokenise_and_parse(NbestList, ClientPid);
        noinput ->
            ClientPid ! speak('I didn't hear anything'),
            prompt_and_listen(ClientPid);
        nomatch ->
            ClientPid ! speak('I didn't understand')
            prompt_and_listen(ClientPid)
    }).
```

Due to the ambiguities inherent in natural languages, NLP components such as speech recognition, syntactic parsing, and semantic interpretation typically return more than one solution for many inputs. Most speech recognisers, for example, return an n-best list of recognition results (in the simplest case consisting of just an ordered list of strings).

```
tokenise_and_parse(NbestList, ClientPid) :-
    maplist(tokenize_atom, NbestList, TokenLists),
    member(TokenList, TokenLists),
    rpc('http://parser.org', phrase(s(Semantics), TokenList), [
        limit(10),
        src_
    ]),
    process(Semantics, ClientPid).

s(need(user, drink)) --> [i, need, a, drink].
```

15.4 Managing dialogue with state machines

Sometimes an event arrives at a particularly inconvenient time, when a state machine is in a state that cannot handle the event. In many cases, the nature of the event is such that it can be postponed (within limits) until the system enters another state, in which it is better prepared to handle the original event.

Selective receive: we specify which messages we are willing to accept. the receiver decides the order of handling messages Implicit deferral: messages that we do not explicitly receive remain in the message queue until handled

Messages that arrive too early in a finite state automaton would give us a undefined state. The implicit deferral give us a very simple description of a finite state automaton where all messages are allowed to arrive too early.

15.5 Managing dialogue with theorem provers

”The central mechanism of our architecture is a Prolog-style theorem-proving system. The goal of the dialogue is stated in a Prolog-style goal and rules are invoked to “prove the theorem” or “achieve the goal” in a normal top-down fashion. If the proof succeeds using internally available knowledge, the dialogue terminates without any interaction with the user. Thus it completes a dialogue of length zero. More typically, however, the proof fails, and the system finds itself in need of more information before it can proceed. In this case, it looks for so-called “missing axioms,” which would help complete the proof, and it engages in dialogue to try to acquire them.”

15.6 Building games

Multiplayer games requires the server to send periodic snapshots of the world state to the client.

15.7 Building a Prolog Web of Things

Polling on web services for home control.
https://www.manning.com/books/building-the-web-of-things?a_bid=16f48f14&a_aid=wot

```
:- module(sensor, [
       sensor_data/1
   ]).

:- use_module(library(http/http_open)).
:- use_module(library(http/json)).

sensor_data(SensorData) :-
    URI = 'http://devices.webofthings.io/pi/sensors/',
    setup_call_cleanup(
        http_open(URI, Stream, [
            request_header('Accept'='application/json')
        ]),
        json_read_dict(Stream, SensorData, []),
        close(Stream)).

?- sensor_data(SensorData).
SensorData = _G6823{humidity:_G6751{description:"A humidity sensor.",
frequency:5000, name:"Humidity Sensor", timestamp:"2016-11-03T09:49:41.998Z",
type:"float", unit:"percent", value:30.1}, pir:_G6809{description:"A passive
infrared sensor. When true someone is present.", gpio:20, name:"Passive
Infrared", timestamp:"2016-11-03T09:15:36.468Z", type:"boolean", value:false},
temperature:_G6687{description:"A temperature sensor.", frequency:5000,
name:"Temperature Sensor", timestamp:"2016-11-03T09:49:41.997Z", type:"float",
unit:"celsius", value:24.8}}.
?-

test :-
    self(Self),
    spawn(status(Self), Pid),
    receive({
        temperature(T) when T > 25 ->
            do_something ;
        temperature(T) when T < 20 ->
            do_something_else ;
    }}.

status(Parent) :-
    repeat,
    sleep(600),
    sensor_data(Sensordata),
    Parent ! temperature(Sensordata.temperature.value),
    fail.
```

Chapter 16

Standardisation

I still believe that standardization is vital for the future of Prolog as a programming language. But the current ISO process is the wrong way to do it. – *Paulo Moura, 2009* [1]

A good programming language standard makes writing portable programs possible – provided that the standard covers all the features used by the program, and that implementations conform to it. The core ISO standard for Prolog, while a significant achievement, is limited in its scope and does not address a number of features that real applications need. It has thus failed both in the sense that few existing Prolog systems of today are conformant, and that most systems go far beyond the standard in some respects. SWI Prolog, SICStus Prolog, YAP Prolog, Ciao Prolog, GNU Prolog, XSB Prolog, BinProlog, Jekejeke Prolog – each of them has unique features that the others do not. This has made porting Prolog programs somewhat difficult. In a discussion on Stackoverflow,[2] Jan Wielemaker proposes that

> Ideally we should sit together and assemble a new system that combines the speed of YAP with the tabling of XSB, the constraints of ECLiPSe and the environment and interfaces of SWI-Prolog.

That would helpt, but is not likely to happen. Instead, we propose that a more realistic ideal would be to have YAP Web Prolog, XSB Web Prolog, ECLiPSe Web Prolog and SWI Web Prolog, etc. – all capable of talking to each other on the Prolog Web. This is not easy to achieve, but it is doable. By defining a system of profiles, and by sticking to just a subset of ISO Prolog (for the least advanced profiles) the Prolog community may be able to move in this direction.

SWI-Prolog is one of the systems that does not really conform to the ISO standard, a fact lamented by some people, witness the following quote from a mail message sent to the SWI-Prolog mailing list from Markus Triska:

[1]`http://blog.logtalk.org/2009/10/stepping-down-as-editor-of-iso-prolog-standardization-proposals`

[2]`https://stackoverflow.com/questions/35668475/prolog-vs-erlang-and-other-functional-languages`

> I am extremely reluctant to depart from ISO conforming syntax when there are acceptable alternatives. Many of us work in environments where using anything that is not codified in industrial standards is completely out of the question for legal and insurance reasons alone, so I try to stick with ISO wherever possible, also in the interest of portability between different Prolog systems. For many of us, there will come days where we wonder if our whole program would not be much faster in YAP or SICStus, and it is often useful to have an alternative system in reach. Currently, the dict syntax in SWI breaks even the canonical syntax. If you work in a highly regulated environment, a system with such a property will not be taken seriously, even if it is very useful at places that are less regulated.

This is the response by Jan Wielemaker:

> The ISO core is great to have, but too small, too much bound by poor decisions from the past (e.g., numbers and text) and impossible to evolve. Ideally we'd have a more flexible and wider adopted standardisation effort. All attempts in that direction failed due to lack of commitment and time as well as a wide variation in the focus of the various systems.

Here is our take on this: If we really need to conform to a standard in order to attract users "working in a regulated environment", and if the current ISO standard is impossible to evolve, then it seems we need a *new* standard which could eventually, in theory at least, satisfy the demand for a standard posed by lawyers and insurance companies, and at the same time allow us to evolve the Prolog language. For industrial users, the *prospect* of a new standard may be enough to increase both trust and interest.

Regarding portability, let us look at a piece of the quote from Markus Triska again:

> I try to stick with ISO wherever possible, also in the interest of portability between different Prolog systems. For many of us, there will come days where we wonder if our whole program would not be much faster in YAP or SICStus, and it is often useful to have an alternative system in reach.

Web Prolog is aiming not only for portability, but also for interoperability. It is a well-known fact that different Prolog implementations are good at different things, so in principle, and if and when YAP and SICStus and ECLiPSe have implemented their own Web Prolog systems, it would allow a SWI Web Prolog developer who is not happy about the performance of tabling in some of his code to hand it over to YAP, or if a constraint problem takes too long, it might be faster to perform the CLP computation on a SICStus or ECLiPSe node, even when taking the cost of the node-to-node communication involved into account. In other words, thanks to enabling communication amongst different Prolog systems, it would allow them to cooperate.

So if the ISO process was the wrong way to do it (as claimed by Paulo Moura), then what would be the right way? Inspired by the previous discussion and as food for thought we would suggest something along the following line: 1) a sharper focus, 2) a different process, and 3) a smaller standard.

16.1 A sharper focus

The focus we argue for in this report should be evident by now. We would like to see a Prolog standard more focused on the Web than the current one, which does not have anything to say about the Web (and understandably so, since the ISO standards predates the Web). The goal should be to build a better Web, and Prolog should be seen as a means to that end rather than an end in itself. The Web thrives on standards and cannot work without them, so with a better Web in sight, the development of a standard does in fact becomes *necessary*.

16.2 A different process

We suspect that most readers are aware that if we choose to go this way there is a suitable standardisation organisation in place, namely the *World Wide Web Consortium* (W3C).[3] We have the impression that creating a W3C standard is somewhat easier than creating an ISO standard. All that is needed is a good design, lots of tests, and two implementations (from different organisations) that pass the tests. If design mistakes or errors of omission were made, a new version may be released that corrects them. Still, we must not underestimate the difficulties – standardisation is hard and often frustrating – but W3C does at least provide those who want to try with a suitable point of entry in the form of *W3C Community Groups*:[4]

> A W3C Community Group is an open forum, without fees, where Web developers and other stakeholders develop specifications, hold discussions, develop test suites, and connect with W3C's international community of Web experts.

The idea, illustrated in Figure 16.1, is that *any* individual who has an idea for a standard can 1) share it with other people (that is what we are doing right now), 2) propose a group (if sharing works out ok, this is something we might do). If the group is able to reach some sort of agreement they can 3) publish a report, and if this report is able to raise some interest within the W3C, then 4) the necessary steps towards forming a proper official W3C working group can be taken.

16.3 A smaller standard...

The *size* of a standard for Web Prolog can, at least initially, be kept quite small. We admit that what we have so far proposed in this document is by no means a small language. But there is no need to standardise the whole lot in one go. We can *start* small, with the RELATION, ISOBASE and ISOTOPE profiles. Confident that it may later be extended, we can begin by focusing on the subset of ISO Prolog which in combination with the predicate `rpc/2-3` and the HTTP API makes interoperability both possible and worthwhile. A standard for the ACTOR profile can appear later.

Another reason to keep the standard small initially is to encourage implementation of it. Without at least two implementations, there will be no standard, since at least two implementations passing all tests is a hard W3C requirement.

[3] `https://en.m.wikipedia.org/wiki/World_Wide_Web_Consortium`
[4] `https://www.w3.org/community`

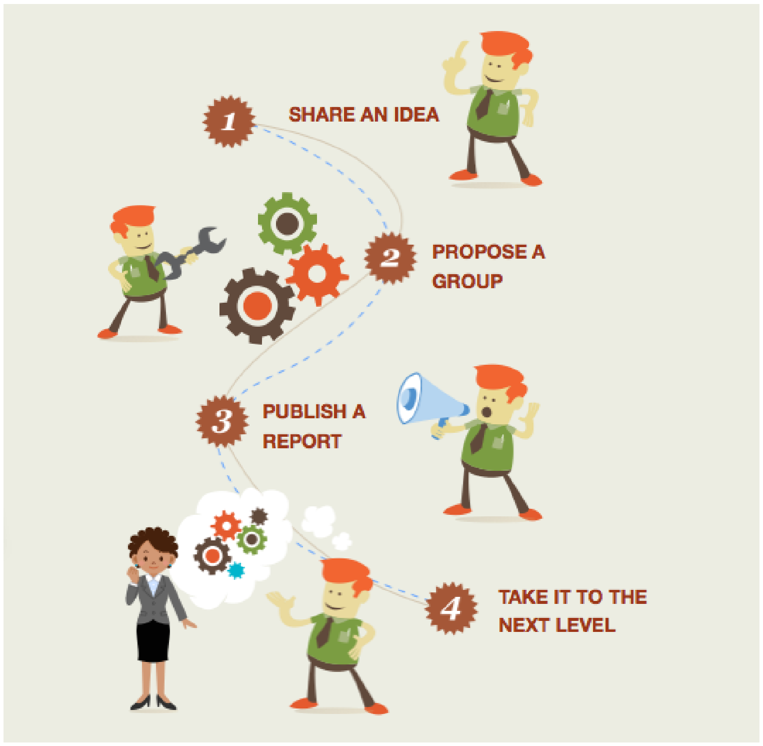


Figure 16.1: W3C Community Group

Here is a sketch of what needs to be in a standard for Web Prolog:

- The *media type* of Web Prolog is `text/web-prolog`. Similar to other media types such as HTML, it must be defined very precisely. Fortunately, to a large extent we can rely on the ISO Prolog standard here.

- We would need to standardise not only the language but the runtime – the minimal node – as well. For each profiles we need to specify each web API that a conforming node must support. A node with ACTOR capabilities must support both the HTTP and WebSocket transport, the other profiles must support HTTP. This set of HTTP endpoints (and possibly a set of WebSocket channels as well) forms the essence of what it means to be a node on the Prolog Web.

- The communication protocol

- The exact structure and content of each of the standard messages (`down`, `spawned`, `ask`, `next`, `stop`, `abort`, `exit`, `success`, `failure`, `error`, `input` and `output`) must be specified. Note that for the basic profiles, only `success`, `failure` and `error` must be specified.

- Clients must be able to query a node's settings, so standardising settings – their names and permissible values – will probably turn out to be important as well. A closed set of typed and named values would be appropriate here, we think.

16.4 ...but extensible

Ease of implementation has so far served as a guiding principle for the choice of language features to be included in the basic profiles. It must be fairly straightforward, or at least not very difficult, to implement the least powerful profiles of Web Prolog in any reasonable general-purpose programming language. With such profiles out of the way, we can turn to more exotic features that may or may not be included in more advanced profiles, those that probably will be supported only by few platforms. When doing so, we should let go of our guiding principle and allow features that are known to be difficult to implement.

Compared to the work required for the less capable profiles, standardising the ACTOR profile is likely going to be more difficult. Some data types in the ACTOR profile – proper strings, support for Unicode, and record-like structures — are not part of the ISO standard. It does not mean that we have to start form scratch, however, since some Prolog systems *do* support strings, and SWI-Prolog supports dicts. In the best of worlds, the ISO committee would be willing to add these data types to the ISO standard. This may not be easy though, since (as noted by Markus Triska), the dict syntax in SWI-Prolog breaks even the canonical syntax as specified in ISO Prolog. Still, we do not think we should hesitate to make the changes necessary in order to include these data types, even at the price of losing some of the conformance with ISO Prolog.

We may also want to introduce *open* JSON terms and *JSON unification* as an operation over such structures. This is something that does not appear in other Web standards, but which would fit in very well in a web-enabled Prolog dialect such as Web Prolog. Supporting JSON terms would show that Web Prolog cares enough about web formats to treat JSON as a first class citizen. This is likely to look good in the eyes of the W3C, since JSON unification is something which seems to both fit in very well with Prolog and to be very useful when manipulating JSON.

The concurrent programming model chosen by Web Prolog is definitely at odds with the model on which the ISO Prolog Threads draft proposal is based. But there is no competition – we do not propose that our model should replace the ISO model. (It would be far too late for this, anyway.) Although Web Prolog tries to piggyback on core ISO Prolog, it is important to keep in mind that Web Prolog and ISO Prolog are *different* programming languages. No one should therefore expect them to support the same concurrency model. We do believe that the proposed model is the better choice for a concurrent *web* programming language such as Web Prolog, but for general-purpose Prolog, the ISO model, complete with threads and shared memory and mutexes and all that, may well be the best choice. We would prefer to avoid taking a stand on this matter.

16.5 Summing up

On the Web, standards are not only “nice to have”, but necessary. The Web just would not work without them. In the case of Web Prolog and the Prolog Web a standard must cover not only computation, i.e. the syntax and semantics of programming primitives, but also the pragmatics of communication, i.e. the protocols and the APIs for using them. Fortunately, the committee behind the ISO Prolog standard has done a great job specifying the syntax of Prolog and the semantics of built-in predicates, which means that most of the effort described in this report could be directed towards the pragmatics of client-to-node and node-to-node communication.

Of course, we cannot be sure that we will succeed in creating a new standard, but here are some points that suggests that we might. Prolog and Erlang are mature programming languages that have stood the tests of time. Tens or hundreds of millions of lines of code have been written in Prolog, and in case of Erlang, they can most likely be counted in the billions.

As it is based on the ISO standard of Prolog, the *de facto* standard of Erlang, and a number of well-established web standards, Web Prolog does not aspire to much originality or novelty, except perhaps in the ways underlying concepts, models and mechanisms are *combined* in our ambition to introduce Web Prolog as an embedded and sandboxed scripting language for the Web.

When it comes to standardisation, a lack of originality and novelty may actually be an advantage, since it makes it more probable that a standardisation effort might actually succeed. Few hitherto unknown problems are likely to surface, so by as much as possible borrowing rather than inventing we may be able to create a viable and trustworthy standard much quicker than would otherwise have been the case.

The real test for a specification is when different groups of implementers can create interoperable applications following the specification without talking to each other. Only through independent implementations and interoperability testing are we likely to end up with a complete specification.

Chapter 17

Final words

An attempt to make Prolog popular again may seem like a good cause... but probably only if you already are a Prolog programmer. Motives matter, and the rest of the world – which is huge and mostly addicted to other programming languages – may not care much for this particular motive of ours.

Twenty-five years in the making, the World Wide Web is a formidable success and the biggest distributed programming system ever constructed, with billions of daily users and millions of contributors. The Web is great, but not perfect by any means. So here is a better motive: let us make an attempt to ensure that Prolog can be used for building an even greater web, a web endowed with logic and reasoning capabilities, easy to use and easy to contribute to. Let the Prolog Web be our bid to make the Web into a better and more interesting place. If successful, the greatness and popularity of Prolog will surely follow.

Before getting carried away along this avenue of thought, we need to ask ourselves if the Web really needs Prolog? Perhaps the Web needs more logic than Web Prolog is able to give? The ideas behind the Semantic Web seem to suggest that this is the case, and that there is need for a logic of a different kind than Prolog is able to provide. Maybe the Web does not require any formal logic at all? Maybe a non-logic programming language such as JavaScript is the only language that the Web needs – one language to rule them all? Some would probably say that the popularity of JavaScript suggests that this may be the case.

The Semantic Web has not really, at least not yet, delivered what it set out to deliver, and is no longer the latest and hottest among AI-related technologies. Non-symbolic methods such as deep learning has taken over. JavaScript has improved considerably over the years, and although it has borrowed a lot of features from "real" functional languages it still is an imperative language at its core.

17.1 Contribution: Enhancing the World Wide Web

Looking back at the quote from Chapter 1 once again, a commitment to the Web seems actually to be what the organisers of the workshop was aiming for (our emphasis):

> This workshop is the second in a series intended to explore the elective affinities between Logic Programming and Internet technologies with *emphasis on*

> *enhancing the World Wide Web* with knowledge, deductive abilities and superior forms of interactive behavior. With the paradigm shift to highly inter-connected computers and programming tools, logic programming languages have *a unique opportunity to contribute* to practical Internet application development. Simplicity, remote executability, robustness, automatic memory management, are among the features some LP languages share with emerging tools like Java. Superior meta-programming and high-level distributed programming facilities, built-in grammars and dynamic databases, declarative semantics are among their competitive advantages. [...]

Note the emphasised phrases. The organisers of the workshop obviously saw logic programming as means to an end – the building of a better Web. This is a spirit worth cultivating. If we did not think that Prolog has something to contribute to the Web that was not there before, we should not be doing this.

More than twenty years after the quoted passage was written, we believe that the unique opportunity for logic programming to contribute to the Web is still there. Of course, logic programming has already done so, in particular through contributions to Semantic Web research. However, since the Semantic Web has focused more on logic than on programming, as a programming language Prolog has only been able to add very little. We are convinced that Prolog, as the first mature and most popular logic programming language, has a lot more to give, and we think that the approach sketched in this report might be one way to realise it.

17.2 A bird's eye view

In this report, we have focused on three concepts in particular: the concept of Web Prolog as a domain-specific programming language, the concept of the actor as the locus of computation and communication, and the concept of the Prolog Web as the environment in which actors are born, act and die.

We have described Web Prolog as a simplification and specialisation of the general-purpose programming language Prolog, enhanced by features borrowed from Erlang and (to a lesser degree) from JavaScript. We have presented actors, and pengines in particular, as simple forms of agents, and the pure Prolog Web has been described as a simple form of web based on logic, at least somewhat related to the Semantic Web.

Web Prolog is a language for representing the knowledge available to actors running on a node. Web Prolog is furthermore a language for programming the pengines and other actors which will make use of this knowledge. The owner of a node uses it to write node-resident programs and to create node-resident actors. Clients uses it to inject source code into an actor to be created. Non-Prolog clients use Web Prolog to communicate with actors and actors use it when communicating amongst themselves. Messages contain queries or commands, and they are all couched in the language of Web Prolog.

This has placed us in a position to argue that the relational dependencies between the language of Web Prolog, the actors, and the Prolog Web are much more precise and elaborated than the dependencies between the languages that served as our inspiration, the more general concept of a software agent, and the vision that has inspired the Semantic Web.

Human agents are able to communicate with artificial agents (i.e. actors) "living" in

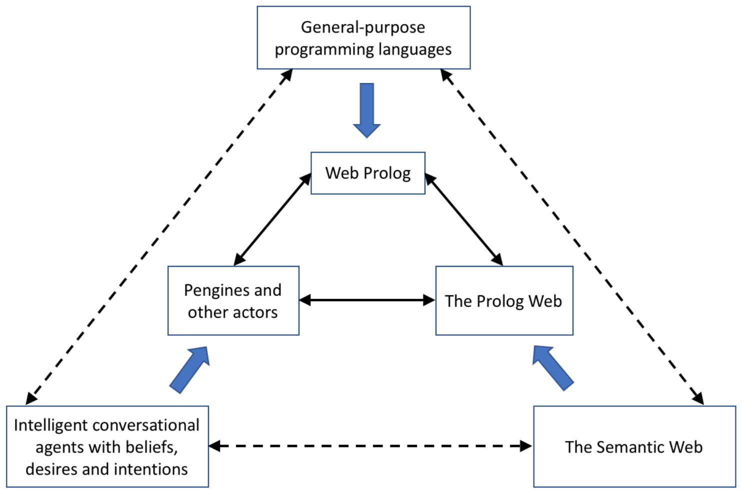

General-purpose programming languages
Web Prolog
Pengines and other actors
The Prolog Web
Intelligent conversational agents with beliefs, desires and intentions
The Semantic Web

Figure 17.1: The big picture. The different kinds of arrows can be interpreted in the following way: Thick and blue stands for *simplification*; solid black for *precisely defined dependencies*; dashed lines for *vague dependencies*.

this environment (i.e. in the Prolog Web), and these artificial agents are in turn capable of enter into conversations with other agents living there. The language that they use for communication is Web Prolog. The beliefs that an agent possesses are represented in the language of Web Prolog too. Web Prolog can be described as the "language of thought" (also known as "mentalese") of such agents.[1]

17.3 What kind of programming language is Web Prolog?

Prolog is different, but not that different – *Richard O'Keefe*

Paraphrasing O'Keefe, we might also say about Web Prolog that it "is different, but not that different". It is clearly not that different from traditional Prolog, and indeed not that different from Erlang either. It is even, in certain ways, not that different from an embedded and sandboxed scripting language for the Web such as JavaScript. Family resemblances abound, although most features of Web Prolog are clearly inherited from traditional Prolog.

A common way to categorise programming languages is in terms of the programming paradigm(s) that they support. In the taxonomy of paradigms by Van Roy (2009), depicted in Figure 17.2, it is indeed possible to pinpoint exactly where Web Prolog fits in. We have enclosed the relevant boxes in boxes with rounded corners.

With access to predicates for asserting and retracting facts in the dynamic database, traditional Prolog has always been able to support imperative programming (and later additions such as `setarg/3` has of course amplified this). Thus Prolog has never really been a single-

[1]https://en.wikipedia.org/wiki/Language_of_thought_hypothesis

paradigm programming language. With the addition of support for the actor programming model, Web Prolog becomes a much clearer as well as (in our view) a more interesting case of a multi-paradigm programming language.

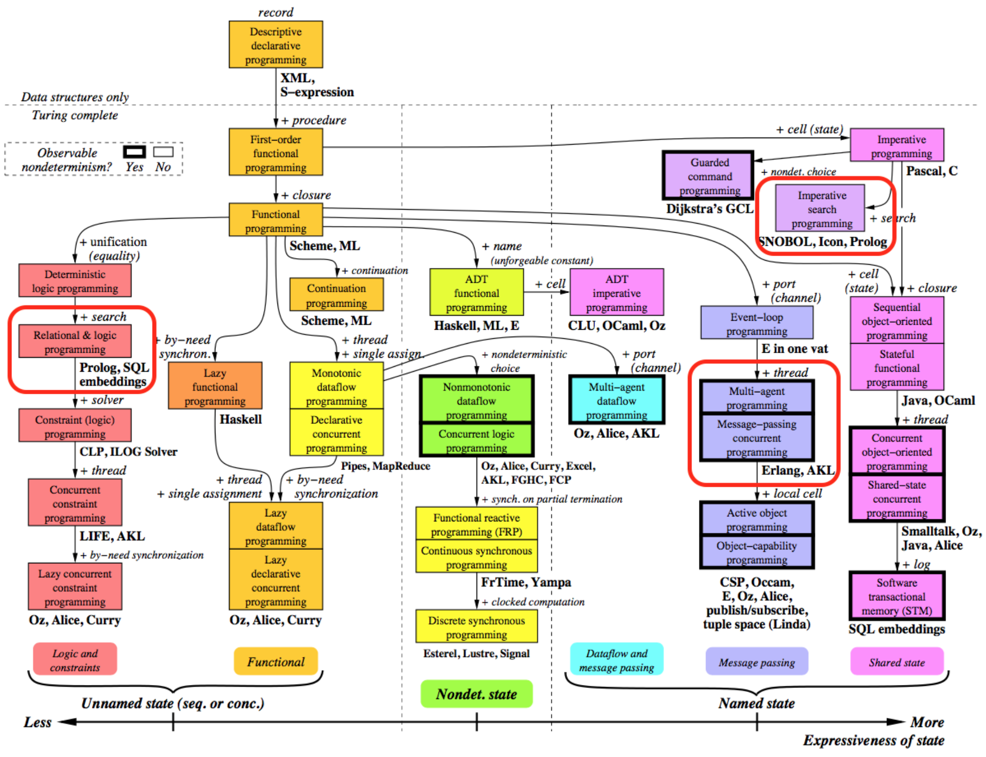


Figure 17.2: A taxonomy of programming paradigms due to Peter van Roy.

However, paradigms alone hardly capture all there is to a practical programming language. Languages may also be related by having a special purpose in common, or by constraint imposed upon them by a particular area of application. They are related by family resemblances rather than anything else, and connected by a series of overlapping similarities, where no one feature is common to all languages. So below, rather than looking for the essence of Web Prolog, we are looking for resemblances with other programming languages, other programming models and other areas of use:

Web Prolog is a logic programming language Based on formal logic, a subject dating all the way back to the antiquity and tried and tested by generations of logicians and philosophers, logic programming forms a paradigm of it own. Even in its pure form, as a set of Horn clauses, Web Prolog is Turing complete. Adding negation as failure gives us an expressive form of non-monotonic logic. Thus, Web Prolog represents a programming paradigm which at its core is unique and very different from JavaScript and other imperative, object-oriented or functional languages that aspire to become web programming languages.

Web Prolog is a dialect of Prolog Prolog can be characterised as a logic programming language with imperative and procedural features added to ensure its usefulness as a general purpose programming language. Prolog is still the best known and most frequently used logic programming language and forty-five years of logic programming research has not been able to replace it. We know of no other logic programming language mature enough to aspire to the role that we want to give Prolog on the Web. It is furthermore the only logic programming language that has been standardised, giving us something to build on in our attempt to adapt Prolog to the Web and create a W3C standard for it.

Web Prolog is a dialect of Erlang We admit that seeing Web Prolog as a dialect of Erlang may be stretching the notion of a "dialect" a bit too far, but we think it's possible to think of it as a variant of the Erlang language where we have "plugged out" its sequential part and "plugged in" sequential Prolog instead. Consider the syntactic, semantic and pragmatic similarities between Web Prolog and Erlang: dynamic typing, assign-once variables, reliance on pattern matching and recursion, and also the fact that the difference between a function and a relation is not all that big, since the former is a special case of the latter. Consider also the Web Prolog primitives for spawning and messaging that are borrowed straight from Erlang. Admittedly, Erlang has a different focus than Web Prolog – a focus on fault tolerance and very efficient concurrent programming – but the fact still remains: while it would be silly to claim that (say) JavaScript is a dialect of Erlang, claiming this for Web Prolog is much more reasonable.

Web Prolog is an extension of Erlang Seen as an Erlang dialect, Web Prolog has many features not found in standard Erlang, features that it inherits from traditional Prolog: built-in backtracking search, unification, logic-based knowledge representation and reasoning, meta-programming, user defined operators, a term expansion mechanism and definite clause grammars. Furthermore, Web Prolog tries to be more open than Erlang – open in the sense that the Web is open. Indeed, if it was not for the fact that Web Prolog resembles Prolog a lot more than it resembles Erlang, we might have named it "Web Erlang".

Web Prolog is an embedded language Just like JavaScript, and in contrast to traditional Prolog and Erlang, Web Prolog is an embedded language, a language designed to be implemented in a host language running in a host environment. We have embedded our Web Prolog proof-of-concept implementation in SWI-Prolog, making good use of the libraries that this platform provides, but we are convinced that it can also be embedded in other Prolog systems, in systems supporting non-Prolog programming or knowledge representation languages, and possibly also in web browsers through transpilation into JavaScript or compilation into WebAssembly.

Web Prolog is a sandboxed language Just like JavaScript, in support of openness and in contrast to traditional Prolog and Erlang, Web Prolog is a sandboxed language, open to the execution of untested or untrusted source code, possibly from unverified or untrusted clients without risking harm to the host machine or operating system. Therefore, Web Prolog does not include any predicates for file I/O, socket programming or persistent storage, relying for these upon the host environment in which it is embedded. Access to the node's host environment can only be granted by the owner who can implement new predicates in the host language and import them into the program space of the node. The author of such code must take responsibility for any security issues that may arise as a consequence of such imports.

Web Prolog is a scripting language Just like JavaScript, and in contrast to traditional Prolog and Erlang, Web Prolog sometimes plays the role of a scripting language. The best example of scripting probably is when a client injects a (usually small) chunk of Web Prolog source code into a pengine or other actor created on a remote node, thus adding to the context in which subsequent queries by this client are to be evaluated. Note that in contrast with JavaScript, the client is scripting the server, rather than the other way around. Of course, this is exactly what is involved in "bringing code to the data". Doing it the other way around, i.e. "bringing data to the code" is also possible. Indeed, there is an useful symmetry here, allowing Web Prolog code to flow in either direction, from the client to the node or from the node to the client. The choice is determined by the programmer's selection of combinations of options that will configure the actor to be created, but the choice can in principle also be made programmatically at runtime.

Web Prolog is an interactive-mode language Similar to other scripting languages, Web Prolog is an *interactive-mode language*, allowing programmers to enter queries one at a time in a shell, and to see the result of their evaluation immediately. This is already a well-known and much appreciated feature of traditional Prolog. In a Web Prolog shell the interactive-programming mode has been extended with `flush/0` for inspecting the content of the mailbox of the top-level pengine to which the shell is attached, a feature that it borrows from Erlang.

Web Prolog is an AI programming language People usually remember Prolog as a language touted as a tool for Artificial Intelligence, and the actor model was also motivated by AI. It is therefore reasonable to think of Web Prolog as an AI programming language, although at the same time we must admit that the hottest topic in AI today seems to be *deep learning*, a pure neural-net approach to machine learning showing great promise when applied to problems such as image recognition, speech recognition and machine translation – for which Prolog is not a suitable implementation language. On the other hand, many researchers agree that deep learning must be combined with symbolic, knowledge-based methods, for which Prolog, and therefore also Web Prolog, may well be suitable.

Web Prolog is a knowledge representation language Similar to most other scripting languages, Web Prolog can be used for purposes for which "scripting" is not really the right word. Using Web Prolog, the owner of a node is for example able to build knowledge bases consisting of millions of clauses – facts as well as rules. (This, obviously, depends a lot on the actual capability of the node in question. For our SWI-Prolog implementation it would be true.) However, as a knowledge representation system, although it subsumes relational as well as deductive databases, Prolog (and Web Prolog) as such it is rather weak in terms of expressiveness. But there is also negation as failure – an important extension allowing non-monotonic knowledge representation

– and with its core based on logic, Web Prolog is rather well suited for interfacing to other logic-based languages of the knowledge representation world such as languages for constraint logic programming, abductive logic programming, probabilistic logic programming and disjunctive logic programming.

Web Prolog is a semantic web programming language For knowledge representation, the latest and web scale experience is surely to be found in the Semantic Web. As we have shown, Prolog is both an RDF query language and a general purpose programming language which due to its relational nature avoids the infamous object-relation impedance mismatch, and therefore provides a perfect platform for semantic web research and applications. While Web Prolog may not be the *best* possible knowledge representation language, the *best* possible query language or the *best* possible general programming language for the Semantic Web, it is the *combination* that might win semantic web researchers over, just as the combination of tools offered by Swiss army knives has won people over. (The analogy is a little bit misleading though, since the combination of tools offered by Web Prolog is arguably more coherent than the combination of tools offered by a Swiss army knife.)

Web Prolog is an ultra-high-level programming language Programming languages are sometimes characterised in terms of their location on a spectrum from "low-level" to "ultra-high-level". Traditionally, Prolog is seen as a *high-level programming language*, and scripting languages are sometimes described as *very-high-level programming languages*. We may therefore view Web Prolog as an *ultra-high-level programming language*, an epithet that we think it deserves due also to the way options are used to "configure" pengines (or other actors) to be run. Some of them are there to inject source code into the actor to be run, while others are pragmas that are there to influence the way the processing and communication are performed.

Web Prolog is an interaction programming language As much as we like the quote in Section 17.1, we have a hard time understanding how the logic programming tools of that time could have contributed "superior forms of interactive behavior" to a Web when HTTP was the sole communication protocol, when Ajax was known only as a brand of cleaning products or as a Dutch football team, and when other acronyms such as REST and JSON did not mean what they mean today. Superior forms of interactive behavior just was not possible at the time, but now, with the advent of WebSockets and the means for programming interaction that the Erlang-ish send and receive brings to the table, the time is ripe. Interactivity implies reactivity, and the receive primitive in particular has proven to be a wonderful tool for reactive programming and for building interpreters for reactive rules and event-driven state machines.

Web Prolog is an actor programming language With the addition of predicates for Erlang-style message-passing concurrency it is reasonable to claim that Web Prolog is an *actor programming language*. In any case, we think that if you are prepared to say that Erlang is an actor programming language (and some may hesitate to do that), then you must be prepared to say that so is Web Prolog. All the machinery that matters for programming systems of actors are there, behaving in a way which is consistent with how it works in Erlang, yet implements a very comprehensive Prolog dialect as well.

Web Prolog is a distributed programming language The advent of massive concurrency through multi-core computer architectures has revived interest in the actor model. Through Erlang (and Elixir) this model has been shown to work extremely well in practice. The actor model, providing us with a single abstraction as a solution to the two problems of concurrent and distributed programming, is a key ingredient in our proposal. The "high-level distributed programming facilities" was mentioned in the quote in Section 17.1, and they may have existed at the time (Linda? April?), but, as we see it, it is not until now that we, thanks to the WebSocket protocol, can hope to implement something that is both adapted to the Web as well as to the future of networked multi-core hardware.

Web Prolog is an agent programming language The notion of an agent is rather fuzzy but there are at least three properties that most theorists would agree that a software agent must possess: it must be a process of a sort, only loosely connected to other processes, it must be stateful, thus have a kind of memory of its own, and it must be capable of interacting with the world external to it. Note that under this definition, any stateful actor would qualify as an agent, and even Erlang might be seen as an agent programming language. A pengine is a kind of actor which in addition to the properties listed above has two other traits that we intuitively tend to associate with agenthood: it is capable of reasoning and capable of giving answers to queries – answers that follow logically from what it believes. So if Erlang is an agent programming language, then this must certainly be true of Web Prolog as well. However, if pengines and other actors are considered agents, then surely they are agents of a very primitive, simple kind. It may be argued that a for a "real" agent – a "soft robot" living in and getting on with its business within the computer's world – the properties listed are not sufficient. A "real" agent must also have desires and intensions, and be capable of making rational decisions based on planning as well as intelligent thinking, something that is implied by the so called *belief-desire-intention* (BDI) software model. A more reasonable stand might be to say that Web Prolog is a *simple* agent programming language, and a language on top of which sophisticated agent programming systems can be built, more easily than with other languages. Since we believe that the future of the Web belongs to agents, we think that the importance of this aspect of Web Prolog will increase with time.

Web Prolog is an agent communication language All communications between a client and a node is couched in the language of Web Prolog, allowing us to think of it as an *agent communication language* (ACL). Compared to standardised ACLs such as FIPA-ACL and KQML, which both define a set of *communicative acts* inspired by speech act theory, Web Prolog is of course much cruder. But the act of asking for the first answer to a query (using `pengine_ask/2-3`) and the act of asking for the next answer to the same query (using `pengine_next/1-2`) may certainly remind us about communicative acts and so do the distinction between Yes/No questions just checking if what is stated is true or false, and WH questions asking for variables to be bound. In the field of expert systems, Why and How questions asking for explanations have also played a role.

Web Prolog is a multi-agent programming language If we choose to look upon a pengine as a simple kind of agent, then it is of course also reasonable to view Web Prolog as a language for programming a *simple* kind of multi-agent systems (MASs). However, we prefer to use the term *multi-pengine systems*[2] instead, and although it resembles a multi-agent language, we are therefore tempted to describe Web Prolog as *multi-pengine programming language* instead.

Web Prolog is a natural language processing programming language Prolog was created as a language intended for natural language processing (NLP) and still is the only (major) programming language with a built-in unification-based grammar formalism. A particular kind of agents for which Web Prolog due to its roots in natural language processing might be a suitable implementation language for is the so called conversational agents.

17.4 What kind of web is the Prolog Web?

The Prolog Web is different, but not that different. We may want to think of it as a layer on top of the current Web, or as something that wraps it. However, the Web as we have known it for a long time shines through – URIs and HTTP are still parts of the foundation of the Prolog Web.

The Prolog Web is a declarative web This is true only to the extent that it is correct to refer to Prolog as a declarative language, i.e. it is not *always* true. `rpc/2-3` is a meta predicate for doing synchronous, non-deterministic remote procedure calls with as query (in the second argument) scoped to the node pointed to by the URI (in the first argument). When the query and the source code held by this node is pure Prolog it is reasonable to speak of a pure and declarative Prolog Web.

The Prolog Web is a procedural web Portions of the Prolog Web is likely to lack purity, either because the programs involved make use of Prolog's impure features, or because they make explicit use of asynchronous Erlang-style concurrency which is not compatible with purity.

The Prolog Web is a wrapper over the traditional web Just as we can describe a node on the Prolog Web as a wrapped software system, we can think of the Prolog Web as a whole as a wrapper around the Web. But as a wrapper, it is transparent, and here and there the Web shines through.

The Prolog Web is an embedded web The Prolog Web is embedded in a multi-pengine system running on virtual network of peer-to-peer nodes acting as Web Prolog runtime environments. On this level, only a procedural understanding of the Prolog Web is possible, an understanding that must involve also the communication between a non-Prolog clients and a node, as well between two or more nodes.

The Prolog Web is a scalable web By adopting a computational model capable of scaling out not only to multiple cores on one machine but also to multiple nodes running on multiple machines, by introducing an option that allows clients to limit the number of

[2]The other alternative term *multi-actor systems* seems to be taken already. See e.g.:

network roundtrips it takes to run a query to completion, by embracing the WebSocket transport protocol with its low overhead per message, by allowing also for an optional RESTful HTTP web API, and by leaving ample room for old as well as new forms of caching on both clients, nodes and intermediaries, we have made our best to ensure that our high hopes for the scalability of the Prolog Web layer are not unfounded.

The Prolog Web is a programmable web In its original form, as a hypertext Web of Documents, the Web was not programmable at all, but this changed fairly early with the introduction of CGI-scripts running on a server that generated documents dynamically, and with JavaScript running on the client. Later, remote procedure calling (XML-RPC, JSON-RPC, SOAP) became prevalent, but the term "programmable web" was not used until individuals or organisations started to publish "open" (often RESTful) web APIs, inviting programmers to combine them into so called "mashups", using JavaScript as glue. While the Prolog Web allows all this, it takes the idea a couple of steps further by allowing the owner of a node to offer clients a complete Web Prolog runtime environment scriptable (another word for 'programmable") by more or less complex queries or by the injection of source code into the pengines or other actors running on the node. The node may or may not be equipped with node-resident programs defining predicates that in addition to the built-in ones may be called by the client. This is another form of programming of the Prolog Web, performed by the owner of a node, rather than by a client.

The Prolog Web is a surfable web Just like many other types of documents on the Web, a Web Prolog program may contain URIs serving as links to other Web Prolog resources. The Prolog Web can thus be surfed (by programmers), or crawled (by software). The editor component of an IDE should probably allow a programmer to (say) left-click such a link in order to bring up a menu allowing the programmer to choose between inspecting the program pointed to by this link, reading its documentation, testing available demonstrations, or browsing a presentation of its author(s).

The Prolog Web is a social web The Prolog Web is not only a technological artefact, but also a social web of Prolog programmers and other users. We believe that this might help to defragment the Prolog community.

The Prolog Web is an open cluster The Prolog Web can perhaps be seen as a Web Prolog alternative to an Erlang cluster.

17.5 Farther into the future

Looking far into an hypothetical future, when the three-pillar foundation consisting of Web Prolog, the Prolog Web and the actors has matured and has been fully standardised, we might want to consider developing extensions in all three directions: towards a more general-purpose Web Prolog language, towards a more sophisticated agent programming framework, and towards a more sophisticated web of logic. When considering such extensions, we must make it a priority to keep the precision and the coherence that we have gained by simplifying

`https://www.tudelft.nl/en/tpm/about-the-faculty/departments/multi-actor-systems/`

things the way we have done.

Similar to how JavaScript has evolved from a small scripting language into a more general-purpose programming language to be used also outside web browsers, we can imagine Web Prolog being developed into a more general-purpose programming language, where file I/O perhaps gets back into the language again, and where fault-tolerance in the spirit of Erlang may be emphasised.

The simple agents may be developed into more sophisticated ones, perhaps acquiring mobility, perhaps desires and intentions. Most likely, sophisticated agents are built from components that are themselves actors running concurrently, allowing agents, as it were, to think, speak and act at the same time.

The Prolog Web would be getting closer to a real semantic web, a rich environment for the agents to reason about. Exactly how this is going to work is far too early to say, but DLP may have a role to play here.

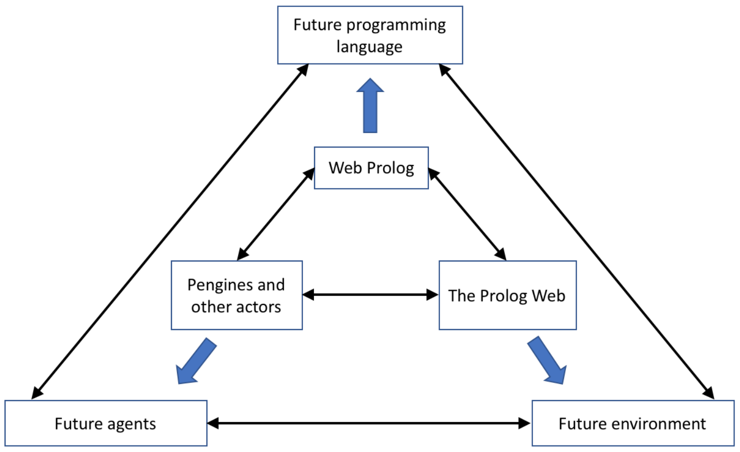


Figure 17.3: The big picture

Bibliography

Alferes, J. J., Damasio, C. V., and Pereira, L. M. (2003). Semantic web logic programming tools. In *In International Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR03*, pages 16–32.

Armstrong, J. (2003a). Concurrency oriented programming in erlang. *Invited talk, FFG*.

Armstrong, J. (2003b). *Making reliable distributed systems in the presence of software errors*. Phd thesis, Royal Institute of Technology, Stockholm.

Armstrong, J., R. Virding, S., and C. Williams, M. (1995). Use of prolog for developing a new programming language.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Bry, F. and Eckert, M. (2006). Twelve theses on reactive rules for the web. In *Proceedings of the 2006 International Conference on Current Trends in Database Technology*, EDBT'06, pages 842–854, Berlin, Heidelberg. Springer-Verlag.

Bry, F. and Marchiori, M. (2005). Ten theses on logic languages for the semantic web. In *Principles and Practice of Semantic Web Reasoning: Third International Workshop, PPSWR 2005, Dagstuhl Castle, Germany, September 11-16, 2005*.

Byrd, L. (1980). Understanding the control flow of Prolog programs. In Tarnlund, S., editor, *Logic Programming Workshop*, pages 127–138, Debrecen, Hungary.

Clark, K., J. Robinson, P., and Hagen, R. (2001). Multi-threading and message communication in qu-prolog. 1:283–301.

Hachey, G. and Gasevic, D. (2012). Semantic web user interfaces: A systematic mapping study and review.

Hebert, F. (2013). *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973*, pages 235–245.

Horrocks, I., Parsia, B., Patel-Schneider, P., and Hendler, J. (2005). *Semantic Web Architecture: Stack or Two Towers?*, pages 37–41. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kifer, M., de Bruijn, J., Boley, H., and Fensel, D. (2005). *A Realistic Architecture for the Semantic Web*, pages 17–29. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kowalski, R. (1986). *Logic for Problem-solving*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands.

Lager, T. and Myrendal, J. (2012). Exploring the web of coined catchy phrases. In *Proceedings of WWW2012: Web Science Track*.

Lager, T. and Wielemaker, J. (2014). Pengines: Web logic programming made easy. *TPLP*, 14(4-5):539–552.

Loke, S. W. (2006). Declarative programming of integrated peer-to-peer and web based systems: the case of prolog. *Journal of Systems and Software*, 79(4):523–536.

Loke, S. W. and Davison, A. (2001). Secure prolog-based mobile code. *Theory Pract. Log. Program.*, 1(3):321–357.

Lombardi, A. (2015). *WebSocket: Lightweight Client-server Communications*. O'Reilly Media, Inc.

Merritt, D. (1989). *Building Expert Systems in Prolog*. Springer-Verlag.

Piancastelli, G. and Omicini, A. (2008). *A Multi-theory Logic Language for the World Wide Web*, pages 769–773. Springer Berlin Heidelberg, Berlin, Heidelberg.

Schrijvers, T. and Demoen, B. (2008). *Uniting the Prolog Community*, pages 7–8. Springer Berlin Heidelberg, Berlin, Heidelberg.

Stamos, J. W. (1986). *Remote Evaluation.* PhD thesis, Massachusetts Inst of Tech Cambridge lab for Computer Science.

Stamos, J. W. and Gifford, D. K. (1990). Remote evaluation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(4):537–564.

Svensson, H., Fredlund, L.-Å., and Earle, C. B. (2010). A unified semantics for future erlang. In Fritchie, S. L. and Sagonas, K. F., editors, *Erlang Workshop*, pages 23–32. ACM.

Tarau, P. (2011). *Coordination and Concurrency in Multi-engine Prolog*, pages 157–171. Springer Berlin Heidelberg, Berlin, Heidelberg.

Tarau, P. and Majumdar, A. (2009). Interoperating logic engines. In *Practical Aspects of Declarative Languages*, pages 137–151. Springer.

Van Roy, P. (1994). 19831993: The wonder years of sequential prolog implementation. 19-20:385–441.

Van Roy, P. (2009). Programming paradigms for dummies: What every programmer should know. *New computational paradigms for computer music*, 104.

Wielemaker, J., Beek, W., Hildebrand, M., and van Ossenbruggen, J. (2016). Cliopatria: A swi-prolog infrastructure for the semantic web. *Semantic Web*, 7(5):529–541.

Wielemaker, J. and Costa, V. S. (2011). On the portability of prolog applications. In *PADL*, pages 69–83.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2012). SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96.

Appendix A

Comparing Web Prolog with the Pengines library

The early pengines library already gave us most of what is offered by the combination of the HTTP libraries and the plain `thread_*` predicates, but also took away some of the flexibility. As a language, Web Prolog strikes a better balance between ease of use and flexibility, in particular when it comes to concurrent and distributed programming. As a runtime environment, a Prolog Web node offers a generic framework that makes it a lot easier to develop Web applications.

- A pengine is always linked to its children in a master-slave relationship. Among other things, this means that if a parent process terminates, then the children do too. In Web Prolog, the monitor and link options of the `spawn/2-3` predicate allows a more flexible regime.
- A pengine is only able to send data to either its parent process (using `pengine_output/1`), or (by calling `pengine_respond/2 or by using the corresponding web API`) to a child process that asks for it (by calling `pengine_input/3`). With `!/2`, any message can be sent to any Web Prolog process, as long as its pid is known.
- Using the old library(pengines), you can ask a pengine server to create a pengine. This pengine is only known to you. And you can create multiple slaves, but they only talk to you, not to each other. They also go to a well defined set of states, and are not prepared to accept arbitrary commands in arbitrary states. So, you can create and control a tree of master/slaves, where you are the root. But, this is a pure tree. In Web Prolog, no such restrictions are present. You *can* create multiple slaves that talk to each other. Also, the deferring mechanism *does* make it possible for a pengines and other actors to accept arbitrary commands in arbitrary states. The commands are simply deferred until later.
- The predicate `pengine_event_loop/2` was put forward as the suggested way to receive and handle messages arriving at a process. There is also `pengine_event/2`, but it is more or less just a copy of the functionality that `thread_get_message/2` offers.

We believe that `receive/1-2` is the best way to provide the necessary functionality. It is significantly more high-level than `pengine_event/2` but more low-level than `pengine_event_loop/2`. As an added bonus, `receive/1-2` should also be a lot more familiar to people coming to Web Prolog from Erlang or Elixir.

Having something along the lines of `pengine_event_loop/2` is not a bad idea though, and we suggest that a future version, perhaps inspired by Erlang's *gen_event behaviour*,[1] can be built on top of `receive/1-2` in the form of a library. We suspect that an implementation of SCXML may be a better fit for Web Prolog, however.

- In Web Prolog, the layer of distribution has been moved from the pengines library into a layer of its own, the language of Web Prolog. It is this layer that is responsible for location tranparency.

- Relying on a uni-directional transport layer such as HTTP, the implementation of `library(pengines)` is not as efficient as it could be. In the case of Web Prolog, the move to a bi-directional transport layer – websockets that is – was both necessary and did also greatly increase the performance of the distribution layer.[2] Based on HTTP, programs doing I/O were particularly inefficent when running on a pengine since an extra network roundtrip was needed each time time the pengine outputted (i.e. "wrote") something to the client. In Web Prolog, the profiles that support an HTTP API do not support I/O. Web Prolog only supports I/O (send and receive) in profiles that use WebSocket as transport.

- Based on the actor abstraction, the design of pengines in Web Prolog is much clearer than the design of the first version of pengines. The ideas behind the first version were rather shallow. The ideas behind pengines in Web Prolog are deeper, yet easier to understand.

- Concurrent programming with `library(pengines)` is too difficult. Sure, one could always claim that concurrent programming is *supposed* to be hard, but the problem with pengines is that the difficulties multiply since we are dealing with a language (or a way of programming) that has not been *validated* as a sound approach to concurrent programming. We would be much better of if we can somehow make it probable that concurrent programming with pengines is a sound idea, and if we can provide design patterns that work. Again, we feel that emulating constructs from a programming language such as Erlang, that was designed for concurrent programming and has been validated and battle-tested as such is a better approach.

- The name of the predicate for non-deterministic RPC has changed from `pengine_rpc/2-3` into `rpc/2-3` as a reflection of the fact that we no longer consider the link with pengines essential. The declarative semantics of `rpc/2-3` can certainly be understood without involving pengines, and it is easy to imagine an implementation of `rpc/2-3` that is not based on pengines.

[1]`http://erlang.org/doc/man/gen_event.html`

[2]See `http://blog.arungupta.me/rest-vs-websocket-comparison-benchmarks/` for an interesting comparison between the efficiency and scalability of the two transfer protocols.

Having said all this, we hasten to add that we learned a great deal from working with pengines and that there are some good things that we have kept. The focus on options is one of them.

Appendix B

Open questions and future work

B.1 Additional options for some predicates

Other possible options for `spawn/2-3`:

- `src_quote(+Quasi_quote)`
  Injects the local predicates denoted by `Quasi_quote` into the process.

In addition, we foresee the need for options to `spawn/2-3` dealing with authentication.

Other possible options for `pengine_spawn/1-2`:

- `reply_to(+Pid)`

Other possible options for `pengine_ask/2-3`:

- `once(+Boolean)`
  Make `Goal` succeed at most once. This is *not* the same as running a goal `once(Goal)`. Passing the option `once(true)` will produce an answer containing the maximum number of solutions suggested by the `limit` option, whereas the goal `once(Goal)` will only provide one solution. The purpose, in both cases, is to avoid leaving unwanted choice points around.
- `sorted(+Boolean)`
  Note that each chunk of solutions is sorted separately. Sorting is therefore meaningful only when `limit` > 1.
- `timing(+Boolean)`
  Include timing information in the answers. Default is `false`.

B.2 Additional information in answers

Messages of type `spawned` currently only has one argument for holding the pid. In the case of the JSON representation, this is a property `pid` that takes a pid as its value. For the purpose of bringing back information about the actor to the client that spawned it, this message will likely receive at least one more argument/property in the future. Such information could include a listing of the settings of the node. The information can be used to update a predicate `actor_property/2` accessible from the client. .

In messages of type success, failure and error, we also foresee the need for an argument/property holding statistical information such as timing information, should the client ask for such information.

Other properties that may be considered are: 1) `sending-time` (i.e. the time of the event manager of the Web site sending the message), 2) `reception-time` (i.e. the time at which a site receives the message), 3) `sender` (i.e. the URI of the site where the event has been raised), 4) `recipient` (i.e. the URI of the site where the event has been received), and 5) `id` (i.e. a unique identifier given at the recipient Web site). Note that all times must be according to the local clocks, since no globally synchronised clock can be assumed on the Web.

B.3 Porting Erlang/OTP behaviours

The close affinity between Erlang and Web Prolog leads us to believe that behaviours other than the pengine behaviour, adhering to protocols other than the PCP, may be implemented in future versions of Web Prolog. For example, it may well be that some of the behaviours offered by Erlang/OTP can fairly easily be reimplemented in Web Prolog.

A generic event handling behavior Erlang/OTP features a generic event handling behaviour referred to as *gen_event*. It consists of a generic event manager process with any number of event handlers that can be added and deleted dynamically.

Here is a sketch of how to make a closure act as a handler for messages. Note that if done in this way, messages will not be deferred.

```
handle_actor_messages(Closure) :-
    receive({
        Message ->
            ignore(call(Closure, Message)),
            handle_actor_messages(Closure)
    }).
```

Here is is how this abstraction might be called in a partial implementation of `rpc/2-3`:

```
    ...
    handle_actor_messages(myhandle).
```

And here is the definition of `myhandler/3`:

```
myhandle(failure(Pid), _, Pid).
myhandle(error(Pid, Exception), _, Pid) :-
    throw(Exception).
myhandle(success(Pid, Solutions, true), Query, Pid) :-
    (   member(Query, Solutions)
    ;   pengine_next(Pid)
    ).
```

```
myhandle(success(Pid, Solutions, false)), Query, Pid) :-
    member(Query, Solutions).
```

A generic state machine behavior An implementation of SCXML in Web Prolog would be fairly straightforward to build. This would tie two W3C areas together.

State Machine Notation for Control Abstraction.

Would work well also in a browser implementation.

The APIs are given. `pengine_spawn/1-2` will work. An system with scxml capabilities would be able to invoke a state chart like so:

```
?- pengine_spawn(Pid, [
       src_uri('http://ex.org/sm.scxml')
   ]).
```

where the `.scxml` extension should take care of the transpilation.

Our own ETS/Mnesisa variant Implement our Mnesia as RDF dB. Add triples to it. Two or more processes running on the same node may share a dynamic database. Information about updates of the shared database is propagated as messages to all processes that share it.

Support for functional notation? Would support for functional notation be useful when porting Erlang code to Web Prolog? (Such support can quite easily be provided.)]

Can a Prolog parser be modified so that Erlang programs can be parsed?

A better approach to the specification of profiles? Perhaps a better approach to the specification of profiles can be found? Perhaps a profile can be understood and specified as a *set of features* instead of just atoms?

Formalisation A formalization of our approach would not be out of the question since formalisations of (subsets) of Prolog and Erlang (Svensson et al., 2010) have been made.

Flavours **rdf** This profile might promote a view of RDF and SPARQL as nothing more than alternative notations for Prolog facts and queries, respectively. This can be seen as an attempt to refine and make more precise the view conveyed by (Wielemaker et al 2015).

xpath

clp

tabling

persistency Marshalling...

Appendix C

A slightly larger program

Below, we have ported an Erlang program[1] into Web Prolog that uses seven concurrently running actor processes to solve the so called *Dining Philosophers problem.*[2]. This program is also available and can be run in the tutorial accompanying the report.

```
sleep :-
    Time is random_float/10,
    sleep(Time).

doForks(ForkList) :-
    receive({
        {grabforks, {Left, Right}} ->
            subtract(ForkList, [Left,Right], ForkList1),
            doForks(ForkList1);
        {releaseforks, {Left, Right}} ->
            doForks([Left, Right| ForkList]);
        {available, {Left, Right}, Sender} ->
            (   member(Left, ForkList),
                member(Right, ForkList)
            ->  Bool = true
            ;   Bool = false
            ),
            Sender ! {areAvailable, Bool},
            doForks(ForkList);
        {die} ->
            io:format("Forks put away.")
    }).

areAvailable(Forks, Have) :-
    self(Self),
    forks ! {available, Forks, Self},
```

[1]href="https://github.com/acmeism/RosettaCodeData/blob/master/Task/Dining-philosophers/Erlang/dining-philosophers.erl

[2]href="https://en.wikipedia.org/wiki/Dining_philosophers_problem

```
    receive({
        {areAvailable, false} ->
            Have = false;
        {areAvailable, true} ->
            Have = true
    }).

processWaitList([], false).
processWaitList([H|T], Result) :-
    {Client, Forks} = H,
    areAvailable(Forks, Have),
    (   Have == true
    ->  Client ! {served},
        Result = true
    ;   Have == false
    ->  processWaitList(T, Result)
    ).

doWaiter([], 0, 0, false) :-
    forks ! {die},
    io:format("Waiter is leaving."),
    diningRoom ! {allgone}.
doWaiter(WaitList, ClientCount, EatingCount, Busy) :-
    receive({
        {waiting, Client} ->
            WaitList1 = [Client|WaitList], % add to waiting list
            (   Busy == false,
                EatingCount&lt;2
            ->  processWaitList(WaitList1, Busy1)
            ;   Busy1 = Busy
            ),
            doWaiter(WaitList1, ClientCount, EatingCount, Busy1);
        {eating, Client} ->
            subtract(WaitList, [Client], WaitList1),
            EatingCount1 is EatingCount+1,
            doWaiter(WaitList1, ClientCount, EatingCount1, false);
        {finished} ->
            processWaitList(WaitList, R1),
            EatingCount1 is EatingCount-1,
            doWaiter(WaitList, ClientCount, EatingCount1, R1) ;
        {leaving} ->
            ClientCount1 is ClientCount - 1,
            flag(left_received, N, N+1),
            doWaiter(WaitList, ClientCount1, EatingCount, Busy)
    }).

philosopher(Name, _Forks, 0) :-
    io:format("~s is leaving.", [Name]),
    waiter ! {leaving},
    flag(left, N, N+1).
```

```
philosopher(Name, Forks, Cycle) :-
    self(Self),
    io:format("~s is thinking (cycle ~w).", [Name, Cycle]),
    sleep,
    io:format("~s is hungry (cycle ~w).", [Name, Cycle]),
    waiter ! {waiting, {Self, Forks}}, % sit at table
    receive({
        {served} ->
            forks ! {grabforks, Forks}, % grab forks
            waiter ! {eating, {Self, Forks}}, % start eating
            io:format("~s is eating (cycle ~w).", [Name, Cycle])
    }),
    sleep,
    forks ! {releaseforks, Forks}, % put forks down
    waiter ! {finished},
    Cycle1 is Cycle - 1,
    philosopher(Name, Forks, Cycle1).

dining :-
    AllForks = [1, 2, 3, 4, 5],
    Clients = 5,
    self(Self),
    register(diningRoom, Self),
    spawn(doForks(AllForks), ForksPid),
    register(forks, ForksPid),
    spawn(doWaiter([], Clients, 0, false), WaiterPid),
    register(waiter, WaiterPid),
    Life_span = 20,
    spawn(philosopher('Aristotle', {5, 1}, Life_span)),
    spawn(philosopher('Kant', {1, 2}, Life_span)),
    spawn(philosopher('Spinoza', {2, 3}, Life_span)),
    spawn(philosopher('Marx', {3, 4}, Life_span)),
    spawn(philosopher('Russel', {4, 5}, Life_span)),
    receive({
        {allgone} ->
            io:format("Dining room closed.")
    }),
    unregister(diningRoom).
```

—