Intro to Web Prolog for Erlangers

Torbjörn Lager
Department of Philosophy, Linguistics and Theory of Science
University of Gothenburg
Sweden
torbjorn.lager@gu.se

Abstract

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. The language is based on Prolog, with a good pinch of Erlang added. We stay close to traditional Prolog, so close that the vast majority of programs in Prolog textbooks will run without modification. Towards Erlang we are less faithful, picking only features we regard as useful in a web programming language, e.g. features that support concurrency, distribution and intra-process communication. In particular, we borrow 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. On top of the pengine abstraction we develop a notion of *non-deterministic RPC* and the concept of *the Prolog Web*.

CCS Concepts • **Software and its engineering → Concurrent programming languages**.

Keywords Prolog, Erlang, web programming

ACM Reference Format:
Torbjörn Lager. 2019. Intro to Web Prolog for Erlangers. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang (Erlang '19), August 18, 2019, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3331542.3342569

1 Introduction

As a logic programming language, Prolog represents a programming paradigm which at its core is unique and very different from imperative or functional programming languages. Features such as built-in backtracking search, unification and a built-in clause database form the basis for logic-based knowledge representation and reasoning, and support for meta-programming, user-defined operators, a term-expansion mechanism and grammars is also provided. These are the features that underlie Prolog's reputation as a symbolic AI programming language. Also, they are features that Erlang does not support. In this paper, we present Web Prolog – a language which combines the most important features of Prolog with those of Erlang.

The structure of the paper is as follows. The rest of this section justifies the introduction of yet another dialect of Prolog. Section 2 presents an IDE for Web Prolog, introduces the notion of a *node*, and describes the interaction between the IDE and a node over a WebSocket sub-protocol. Section 3 introduces the language of Web Prolog as such and compares it with Erlang. Section 4 looks closer at the combination of the actor model and the logic programming model, placing a particular focus on non-determinism and backtracking. The notion of a pengine is described in detail, and a notion of non-deterministic RPC is developed. Section 5 describes some earlier work, Section 6 provides a discussion, and Section 7 summarises and suggests avenues for further work.

The paper is written with an audience of Erlangers in mind but some basic knowledge of Prolog is assumed.

1.1 Web Prolog is a Hybrid Programming Language

Web Prolog is not only a logic programming language but also an *actor programming language* as it extends Prolog with primitives for spawning processes and sending and receiving messages in the style of Erlang, i.e. constructs that made Erlang such a great language for programming message-passing concurrency and distribution. Provided we can accept using a syntax which is relational rather than functional it turns out that the surface syntax of Prolog can easily be adapted to express them. Since Prolog and Erlang also share many other properties such as dynamic typing, the use of immutable variables, and a reliance on pattern matching and recursion, creating a hybrid between them seems reasonable. Indeed, it allows us to regard Web Prolog either as a new dialect of Prolog, or as a new dialect of Erlang.

1.2 Web Prolog is a Web Programming Language

There are more than a dozen Prolog systems around, and we do not intend to compete with them, or with Erlang for that matter. Although the proposed hybrid between Prolog and Erlang would likely work as a general-purpose language, this is not what we aim for. Instead, as suggested by the first

Erlang '19, August 18, 2019, Berlin, Germany

ACM ISBN 978-1-4503-6810-0/19/08…$15.00
https://doi.org/10.1145/3331542.3342569

part of its name and in an effort to find new uses for Prolog, we think of Web Prolog as a special-purpose programming language – as an embedded, sandboxed scripting language for programming the Web with logic and for implementing web-based communication protocols in the style of Erlang.

Web Prolog is an embedded 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 [6] but we are convinced it can also be embedded in other Prolog systems, in systems supporting non-Prolog programming languages or knowledge representation languages, and in web browsers through transpilation into JavaScript or compilation into WebAssembly.

Web Prolog is furthermore 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 predicates for file I/O, socket programming or persistent storage, but must rely on the host environment in which it is embedded for such features.

2 Running Web Prolog from a Browser

A proof-of-concept web-based IDE for Web Prolog has been implemented in a combination of HTML, CSS and JavaScript. The IDE is equipped with an editor and a shell. Figure 1 shows them as they appear in a browser window, with the editor to the left and the shell to the right.

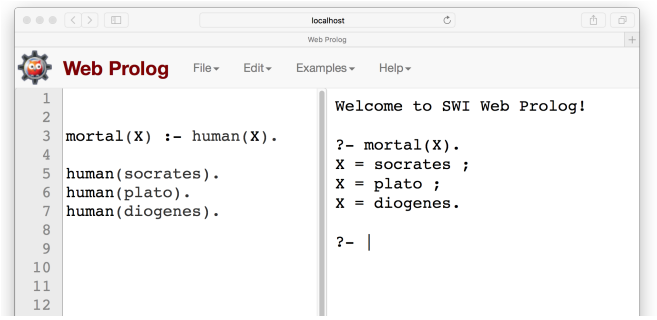

Figure 1. The proof-of-concept IDE.

The tiny program shown in the editor can be assumed to have been written by a programmer who has then entered a query in the shell in order to inspect the results produced one-at-a-time in the usual lazy fashion typical of interactions with a Prolog top-level. The scenario, depicted in Figure 2, also involves a *node*, identified by the URI `http://local.org`, to which the IDE has established a connection.

A node is an executing Web Prolog runtime environment. Its purpose is to host *pengines* and other actors. A pengine is an actor and a programming abstraction modelled on the interactive top-level of Prolog. It can be seen as a first-class Prolog top-level, accessible from Web Prolog as well as from other programming languages.

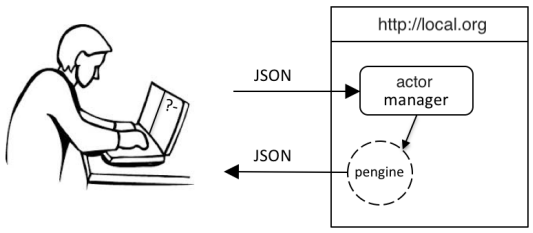

Figure 2. The shell offers mediated communication between a programmer and a pengine hosted by a node.

Despite what Figure 2 may suggest, the programmer may not be alone in interacting with this particular node. Other programmers may be talking to other pengines running there. They are completely shielded from each other, unless 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.

A node is equipped with comprehensive web APIs using 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 by contributions from both. The IDE must be run over the WebSocket protocol.

The task of the node's *actor manager* is to handle the reception of messages sent by clients and arriving over 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.

Crucially, a node may host a Web Prolog *program* – a deductive database, an expert system, a digital assistant, a home control system, or another kind of application – perhaps related to AI and in need of knowledge representation and reasoning. If it does, any pengine running on the node has access to the predicates defined by this program in addition to Web Prolog built-in predicates. The program – which we shall refer to as the *node-resident* program – is typically maintained by the owner of the node. Unless programmers are authorised to do so, they 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, programmers are allowed to *complement* the node-resident program with source code that they themselves have written.

When the programmer enters the URI of the node in the browser's address field, a WebSocket connection is first established between the IDE and the node, and then used to ask the node to spawn a pengine. 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. When talking to a browser, the pengine is instructed to use JSON.

Our proof-of-concept demonstrator of a Web Prolog node is written in SWI-Prolog [6] and can be downloaded from https://github.com/Web-Prolog, installed and taken for a trial run. The IDE is included in the installation. The demonstrator features an interactive tutorial which provides a tour of the language. The editor and shell supports the usual interactive edit-run cycle and allow users to compose and run their own programs.[1]

3 Erlang-Style Programming in Web Prolog

> Reading the code was fun – I had to do a double take – was I reading Erlang or Prolog – they often look pretty much the same.
>
> *Joe Armstrong* (p.c. June 18, 2018)

Most Erlangers are probably aware that Erlang is related to Prolog in more than one way. The first implementation was written in Prolog [1], and syntactically they look rather similar and share a lot of terminology. This is something we try to take advantage of when designing Web Prolog and we even name the predicates which support spawning and messaging after Erlang primitives with similar syntax and semantics. However, as we shall see, the primitives for spawning and messaging in Web Prolog are in some ways more expressive than the corresponding Erlang primitives.

3.1 A Simple Count Server

Just like in Erlang, source code which specifies the behaviour of an actor to be spawned can be written in Web Prolog – the kind of messages it will listen for, and the kind of messages it will send to other actors. Such actors are referred to as *servers* in the Erlang community.

In the editor, a count server can be written as follows and then be loaded by means of injection into the workspace of the pengine to which the shell is attached:

```
count_server(Count0) :-
    receive({
        count(Pid) ->
            Count is Count0 + 1,
            Pid ! Count,
            count_server(Count);
        stop(Pid) ->
            Pid ! stopped(Pid)
    }).
```

This code contains two primitives foreign to traditional Prolog. The predicate `receive/1` is used to select and extract messages appearing in the mailbox of the process running the code. The send operator `!/2` is used to send a message to another process.

[1] In the future, we intend to use a Web Prolog node as a back-end to SWISH [5], a much more mature online IDE for Prolog than the one offered by our demonstrator. See https://swish.swi-prolog.org.

The example demonstrates a programming pattern frequently found in Erlang programs and destined to become very useful in Web Prolog as well: a loop is defined where a call to the receive primitive is used to match a message in the mailbox, do something with it, and then continue looping by making a recursive call. The state of the counter is kept in the argument of `count_server/1`.

The predicate `spawn/2-3` is used to create actor processes and it works almost like the spawn function in Erlang. However, while Erlang is a higher-order language in which the spawn function takes an anonymous function as its argument, Prolog (or Web Prolog) is not a higher-order language in this sense. In Web Prolog, `spawn/1-3` is a *meta predicate* which expects a callable goal to be passed in the first argument. Also, while the spawn function in Erlang exists in more than a dozen variants, Web Prolog has only two, one which takes a list of options and one which relies on their default values. The options are used for the configuration of the actor to be created. Here is how we can spawn the count server from the shell and take it for a trial run:

```
?- spawn(count_server(0), Pid, [
       monitor(true),
       src_predicates([count_server/1])
   ]).
Pid = '8915b2d4'.
?- self(Self).
Self = 'f431a324'@'http://local.org'.
?- $Pid ! count($Self),
   receive({Count -> true}).
Count = 1.
?- $Pid ! stop($Self).
true.
?- flush.
Shell got stopped('8915b2d4')
Shell got down('8915b2d4',true)
true.
?-
```

The `src_predicates` option ensured that the count server source code injected into the workspace of the top-level pengine was also injected into the workspace of the spawned server process. Calling `self/1` determined the identity of the top-level pengine, and `!/2` was used to send the current count back to the client. Calling the utility predicate `flush/0` – also borrowed from Erlang – allowed us to inspect the content of the top-level mailbox, where a message `stopped` as well as a `down` message was found.

Note also the use of another shell utility feature, borrowed from SWI-Prolog, which allows bindings resulting from the successful execution of a top-level goal to be reused in future top-level goals as `$Var`. Together with `flush/0`, this facility comes in handy during interactive programming in the 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 an actor. Furthermore, the node option allows the programmer to spawn an actor process on a remote node instead of locally, and other options allow the caller to monitor the spawned process or to terminate it should the caller die.

Links in Web Prolog are somewhat simpler than in Erlang. In contrast to Erlang's bi-directional links, they are uni-directional. As argued in [4], uni-directional links simplify things and do not harm expressivity. The only kind of link currently supported in Web Prolog is specified by means of an option `link` to `spawn/3` which, if set to `true` (default), causes the 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.

An obvious example of the use of links is that when a programmer closes (or just leaves) the IDE, the top-level pengine which serves the shell as well as any actors that may have been spawned from this pengine are forced to terminate. This might be seen as a supervision hierarchy rooted in the process running the shell.

3.2 Node-Resident Actor Processes

In addition to node-resident source code, the owner of a node may install *node-resident actor processes*. We show an example below which uses `register/2` to give a running count server a mnemonic name:

```
?- spawn(count_server(0), Pid),
   register(counter, Pid).
```

Just like in Erlang, the registered name can be used instead of the pid when sending to the process:

```
?- self(Self).
Self = '51f40b45'@'http://local.org'.
?- counter ! count($Self),
   receive({Count -> true}).
Count = 1.
?- counter ! count($Self),
   receive({Count -> true}).
Count = 3.
?-
```

Contrary to a server injected and spawned by a client, a node-resident server is accessible from any client to the node that knows the registered name of the server. (This explains why 3 rather than 2 was received in the example – another client happened to increment the counter.)

3.3 The Syntax of Send and Receive

The syntax of Web Prolog is ordinary Prolog except that three infix operators (`!/2`, `when/2` and `@/2`) have been declared using `op/3`, which is the predicate for specifying user-defined operators in Prolog.

As shown in the previous section, an actor process uses the receive primitive to extract messages from its mailbox. In Web Prolog, just like in Erlang, this operation specifies an ordered sequence of *receive clauses* 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 query prefixed with the `when` operator. As in Erlang, its role is to make pattern matching more expressive.

To demonstrate the use of the `when` operator and the use of two `receive/2` options that causes a goal to run on timeout, we show a priority queue example borrowed from Fred Hébert's textbook on Erlang [2]. The purpose is to build a list of messages with those with a priority above 10 coming first:

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

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

Below, we test this program 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://local.org'.
?- important(Messages).
Messages = [high,high,low,low].
?-
```

For comparison, here is Hébert's version of `important/0`:

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

Compared to the Web Prolog predicate, the Erlang function is more succinct. There are three reasons for this. First, Web Prolog has a relational syntax which does not allow nesting of calls while Erlang is a functional language where such nesting is the norm.

Secondly, while `op/3` allowed us to define the infix `when` operator, not much can be done about the more complex `receive...after...end` construct. Instead, Web Prolog defines 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 specifying the behaviour around time-outs in the second argument.

Thirdly, 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 a 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. The "pair operator" (`-`) was a sensible choice here, but any valid Prolog term would do.

Three fairly major syntactic differences in such a small program – where at least the first two contribute to the program looking neater in Erlang than in Web Prolog – may seem like a lot for a language that tries to appear as similar as possible to Erlang. However, note that the example was chosen exactly because it highlights many differences in a tiny program. Usually, the differences are less conspicuous.

All in all, we believe that most Erlangers will find the syntax of Web Prolog likeable and easy to work with. Fans of Elixir may be less enthusiastic, at least if the Prolog-ish syntax that Erlang inherited from Prolog is what drew them to Elixir with its Ruby-ish syntax instead.

3.4 The Semantics of Send and Receive

Getting the semantics of `receive/1-2` right is of course more important than getting its syntax right and we believe we have succeeded in doing that. As in Erlang, `receive/1-2` scans the mailbox looking for the first message (i.e. the oldest) that matches a pattern in any of the receive clauses 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. In Web Prolog, just like in Erlang, values of any variables bound by the matching of the pattern with a message are available in the body of the clause.

If no pattern matches a 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. Just like in Erlang, this behaviour is particularly useful if we expect two messages but are not sure which one will arrive first. Note that the implementation of the priority-queue example relies on this behaviour and would not work without it.

Most uses of receive in Erlang can be mechanically translated into uses of receive in Web Prolog that will behave in the same way as in Erlang. However, three differences should be noted. First, Erlang's receive construct is an *expression* (with a value given by the expression on the right hand side of the arrow of a matching rule) rather than a statement (that will succeed or fail as in Prolog, and will not return a value).

Secondly, Erlang enforces purity and efficiency by only allowing a restricted set of primitives 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. In Web Prolog, although only the first solution will be searched for, any query may be used as a guard and values of variables bound by it are available in the body. This makes the receive construct more powerful than in Erlang, but it also means that the programmer is made responsible for keeping guards as simple and efficient as possible and to avoid side effects. Enabling the programmer to condition the matching of a receive clause on the content of the whole Prolog database makes it worth it.

Finally, the receive construct in Web Prolog is a *semi-deterministic* predicate, i.e. it either fails, or succeeds exactly once. As will be shown in Section 4, this is a key property of `receive/1-2` which ensures that backtracking can be handled in an elegant way.

3.5 Concurrent and Distributed Programming

Since the count server in Section 3.1 is running in parallel to the pengine to which the shell is attached and is talking to it using asynchronous messaging, we have already demonstrated the use of concurrency. Below, in a probably more convincing example inspired by a user's guide to Erlang,[2] 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).

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

When `start/0`, defined below, is called, the behaviour of this program exactly mirrors the behaviour of the original version in Erlang.

[2] See http://erlang.org/doc/getting_started/conc_prog.html

```
start :-
    spawn(pong, Pong_Pid, [
        src_predicates([pong/0])
    ]),
    spawn(ping(3, Pong_Pid), _, [
        src_predicates([ping/2])
    ]).
```

Another thing that Web Prolog has in common with Erlang is that spawning and sending work also in a distributed setting. In Web Prolog we can pass the `node` option to the spawn operation to invoke a process on a remote node and subsequently communicate with it using send and receive. For example, if the option `node('http://remote.org')` is passed to any of the above calls to `spawn/3`, the game of ping-pong will be played between two nodes.

3.6 Programming Patterns in Erlang and Web Prolog

As long as languages do deterministic and sequential computation only, i.e. when neither search nor concurrency is involved, functional programming and logic programming are fairly similar in the way they work, and methods used to achieve success with one often transfer to the other. Immutable variables, pattern matching and recursion, for example, typically play important roles in both kind of languages.

When concurrency *is* involved, language designers must choose a good approach and a suitable set of primitives to express it. In a post to the *erlang-programming* mail list, the renowned programming wizard and Prolog and Erlang specialist Richard O'Keefe writes that he "would prefer multi-threading in Prolog to look as much as possible like Erlang".[3] We do agree, but note that it might be too late since at least four Prolog systems have already implemented the proposed ISO Prolog standard for multi-threading.[4] Be that as it may, for a special-purpose dialect such as Web Prolog, Erlang-style concurrency appears to be an excellent choice, especially since it generalises to the distributed case as well.

As a consequence of this choice, Erlang and Web Prolog share not only a great deal of syntax and a lot of terminology but many programming patterns as well, such as the use of concurrency and recursion for maintaining state, the use of protocols, and various abstractions for asynchronous and synchronous communication between processes which may or may not run on the same machine or CPU core. Indeed, as the programs in Section 3 demonstrated, it is usually straightforward to translate Erlang programs into Web Prolog.

The close affinity between Erlang and Web Prolog leads us to believe that behaviours such as those offered by the OTP might be implemented in future versions of Web Prolog. To ensure the uninterrupted service of node-resident actors, for example, the supervisor behaviour would be great to have, and as a way to implement complex protocols in the style of Erlang, the state machine behaviour might be useful.

[3] https://groups.google.com/d/msg/erlang-programming/1jdsnqZ4XfQ/ve9WfFl2YBwJ

[4] https://logtalk.org/plstd/threads.pdf

4 Backtracking beyond Erlang

As we now turn to examples that go beyond what Erlang can easily do, 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 backtracking for example? In the next section we show an example which suggests that the mix is both sound and easy to understand.

4.1 Handling Non-determinism

Suppose the query given in the argument to `spawn/2` has more than one answer, a query such as `?-mortal(Who)` for example. Below, a goal containing this query is called, the first solution is sent back to the calling process, and then `receive/1` is used in order to listen for a message of the form `next` or `stop` before terminating:

```
?- self(Self),
   spawn(( mortal(Who),
           Self ! Who,
           receive({
               next -> fail;
               stop -> true
           })
         ), Pid).
Pid = 'a4b940a8',
Self = 'c4806702'@'http://local.org'.
?- flush.
Shell got socrates
true.
?- $Pid ! next.
true.
?- flush.
Shell got plato
true.
?- $Pid ! stop.
true.
?-
```

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

For an Erlang programmer this particular use of `receive/1` may come as a surprise. After all, the Prolog concepts of

failure and *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. The only way `receive/1-2` will fail is if the goal in the *body* of one of its receive clauses fails. To see how it pans out in a corner case, consider the following two receive calls:

```
receive({m(X) -> true})     receive({m(X) -> fail})
```

The call on the left will succeed if a message matching the pattern `m(X)` appears in the mailbox. The call on the right will fail (and possibly cause backtracking) once a message matching the pattern `m(X)` appears. Only by the left call will the variable `X` be bound. Both calls will remove the matched message from the mailbox.

The tiny examples in this section highlighted a vital feature of the Web Prolog design as they showed how Prolog-style search and Erlang-style concurrency can be integrated and how a non-deterministic query can be supplied with a deterministic interface. This is precisely the point where the logic programming model and the actor programming model – represented here by Prolog and Erlang – interface with each other. This suggests that Prolog's backtracking mechanism is perfectly compatible with, and in fact complements, the proposed Erlang-like mechanisms for spawning actors and handling the communication between them.

The first example above demonstrated an actor adhering to what might be seen as a tiny communication protocol accepting only the messages `next` and `stop`. We need to observe, however, that the goal to be solved was 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 the next section we will describe the API to a full-blown generic pengine abstraction – an actor adhering to a considerably more complex protocol.

4.2 A Pengine is an Actor with a Protocol

In traditional Prolog the top-level is *lazy* in the sense that new solutions to a query are only computed on demand. However, the top-level is not accessible to programs, i.e. a program cannot *internally* create a top-level, pose queries and request solutions on demand. 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 such as JavaScript.

What distinguishes a pengine from other kinds of actors is the *protocol* it follows when it communicates, i.e. the kind of messages it listens for, the kind of messages it sends, and the behaviour this gives rise to. The protocol must not only allow a client to ask queries and a pengine running on a node to respond with answers, it must also allow the pengine to prompt for input or produce output in an order and with a content as dictated by the running Web Prolog program. 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 design behind pengines is in fact 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 semicolon in order to ask for more solutions, or hit return to stop. These are "conversational moves" that Prolog understands. There are even 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. There are only a few moves a client can successfully make when the protocol is in a particular state, and the possibilities can easily be described, by a state machine for example, as we shall do in the next section.

4.3 The Pengine Communication Protocol

Figure 3 depicts a statechart specifying the Pengine Communication Protocol (PCP) – a protocol for the communication between a client and a server (in the Erlang sense of these terms). The server is a pengine running on a node. The client can be any process (including another actor or a JavaScript process) 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.[5]

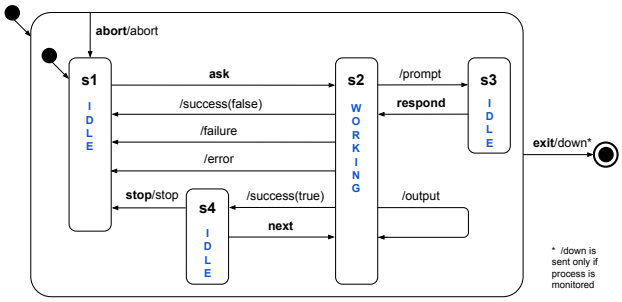

Figure 3. Statechart specifying the PCP for a complete Web Prolog session. The transitions are labeled with *message types*. Types in bold are sent from the client to the pengine, whereas message types with a leading / goes in the opposite direction, from the pengine to the client.

Web Prolog comes with built-in predicates which allow a client to spawn a pengine (`pengine_spawn/1-2`), and send it messages in bold (`pengine_ask/2-3`, `pengine_next/1-2`,

[5]The use of a statechart allows us to show that no matter the current state of the protocol, **abort** will always take it to the state from which a new query can be asked and **exit** will always terminate the pengine process.

pengine_stop/1, pengine_respond/2, pengine_abort/1 and pengine_exit/1). For communication from the pengine to the client, pengine_input/2 and pengine_output/1 are available.

4.4 In Conversation with a Pengine

Below, we show how to create and interact with a pengine process that runs as a child of the current top-level process.

```
?- pengine_spawn(Pid, [
       node('http://remote.org'),
       src_text("p(a). p(b). p(c)."),
       monitor(true),
       exit(false)
   ]),
   pengine_ask(Pid, p(X), [
       template(X)
   ]).
Pid = '7528c178'@'http://remote.org'.
?- flush.
Shell got success('7528c178'@'http://...',[a],true)
true.
?- pengine_next($Pid, [
       limit(2)
   ]),
   receive({Answer -> true}).
Answer = success('7528c178'@'http://...',[b,c],false).
?-
```

There is quite a lot going on here. The node option passed to pengine_spawn/1-2 allowed us to spawn the pengine on a remote node, the src_text option was used to send along three clauses to be injected into the process, and the monitor options allowed us to monitor it. These options are all inherited from spawn/2-3.

Given the pid returned when calling pengine_spawn/1-2, we then called pengine_ask/2-3 with the query ?-p(X), and by passing the template option we decided the form of answers. Answers were returned to the mailbox of the calling process (i.e. in this case the mailbox belonging to the pengine running our top-level). We inspected them by calling flush/0. By calling pengine_next/2 with the limit option set to 2 we then asked for the last two solutions, and this time used receive/1 to view them.

We passed the option exit(false) to pengine_spawn/2, so although the query has now run to completion, the pengine is not dead and we can use it to demonstrate how I/O works:

```
?- pengine_ask($Pid, pengine_output(hello)),
   receive({Answer -> true}).
Answer = output('7528c178'@'http://remote.org',hello).
?-
```

We will not show it here, but input can be collected by calling pengine_input/2, which sends a prompt message to the client which can respond by calling pengine_respond/2.

The pengine is still not dead so let us see what happens when a non-terminating query such as ?-repeat,fail is asked:

```
?- pengine_ask($Pid, (repeat, fail)).
true.
?-
```

Although nothing is shown, we can assume that the remote pengine is just wasting CPU cycles to no avail. Fortunately, we can always abort a runaway process by calling pengine_abort/1:

```
?- pengine_abort($Pid),
   receive({Answer -> true}).
Answer = abort('7528c178'@'http://remote.org').
?-
```

When we are done talking to the pengine we can kill it:

```
?- pengine_exit($Pid, goodbye),
   receive({Answer -> true}).
Answer = down('7528c178'@'http://remote.org',goodbye).
?-
```

Note that messages sent to a pengine will always be handled in the right order even if they arrive in the "wrong" order (e.g. next before ask). This is due to the selective receive which defers the handling of them until the PCP protocol permits it. This behaviour guarantees that pengines can be freely "mixed" with other pengines or actors. The messages abort and exit, however, will never be deferred.

4.5 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. Such calls are synchronous and no explicit concurrency is involved, and this is what makes rpc/2-3 remarkably easy 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.) A Web Prolog client process queries a node by calling rpc/2 with the first argument a URI pointing to the node, and the second argument a query to be run over the predicates offered by the node. Here is a trivial example of its use:

```
?- rpc('http://remote.org', mortal(Who)).
Who = socrates ;
Who = plato ;
Who = diogenes.
?-
```

4.6 An Implementation of rpc/2-3

Below, we show an implementation of `rpc/2-3` which is built on top of a pengine spawned on a remote node and a local loop that waits for answers arriving from it:

```
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)
    }).
```

Note how the disjunction in the body of the third receive clause and the use of `member/2` in the third and fourth clauses turn the deterministic calls made by `pengine_ask/3` and `pengine_next/1` into the expected non-deterministic behaviour of `rpc/2-3`.

4.7 Reducing the Number of Network Roundtrips

Time spent in remote shell-pengine or pengine-pengine interaction can be the dominant factor in the 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 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 a local call. By passing the `limit` option to `rpc/3` (inherited from `pengine_ask/3`) we can make the communication less "chatty" and avoid many roundtrips. Here is an example:

```
?- rpc('http://remote.org', mortal(Who),[
       limit(10)
   ]).
Who = socrates ;
Who = plato ;
Who = diogenes.
?-
```

As the 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 all three solutions. Thus no new request to the node needs to be made. So while the retrieval of the three solutions to the query required three network roundtrips before we applied the option, it will now only require one roundtrip. More generally, 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, or just one roundtrip if $n/i < 1$.

Use of the `limit` option is fine also from a purity point of view – it has nothing to do with logic and the declarative reading of the query, but must be treated as a *pragma* – as a language construct that specifies the granularity with which 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 performance when running the query over a cluster of nodes.

4.8 Shuffling Code and Data Back and Forth

On the internet, the cost of shuffling code and data back and forth across remote boundaries is significant, yet cannot be avoided. But in which direction should the shuffling be made in order to bring down the cost? The answer is most likely that it varies and that programmers should be given a choice.

The obvious way to bring code to the data in Web Prolog is to inject source code into the remote process created by `rpc/2-3`. With the following call, we do just that:

```
rpc('http://remote.org', foo(X), [
    src_text("foo(X) :- mortal(X).")
])
```

The default value of the `node` option for `spawn/2-3` and `pengine_spawn/1-2` is the special-purpose URI `localnode`. 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. Here is an example:

```
rpc(localnode, mortal(X), [
    src_uri('http://remote.org/src')
])
```

The source held by the node at `http://remote.org` is injected into the workspace of the underlying pengine before the query `?-mortal(X)` is tried, thus isolation is provided.

These two examples suggest a useful symmetry which allows 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 options configuring the actor to be spawned, but it can in principle also be decided programmatically at runtime.

4.9 More about the Underlying Web APIs

An 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 a running query. What makes unconstrained mixed-initiative interaction feasible is the support for efficient bi-directional messaging offered by a node thanks to the use of the WebSocket protocol.

Other kinds of web applications may have no need for mixed-initiative interaction. In order to serve such applications, a Web Prolog node offers a stateless HTTP API. Interestingly, it also turns out that since `rpc/2-3` does not produce output or request input, it can be run over HTTP instead of over the WebSocket protocol. In our proof-of-concept implementation this is the default transport.

To retrieve the first solution to `?-mortal(X)` using HTTP, a GET request can be made with the following URI:

```
http://remote.org/ask?query=mortal(X)&offset=0
```

Here too, responses are returned as Prolog or as Prolog variable bindings encoded as JSON. Such URIs are simple, they are meaningful, they are declarative, they can be bookmarked, and responses are cachable by intermediates.

To ask for the second and third solution to `?-mortal(X)`, another GET request can be made with the same query, but setting `offset` to 1 this time and adding a parameter `limit=2`. In order to avoid recomputation of previous solutions, the actor manager keeps a pool of active pengines. For example, when the actor manager received the first request it spawned a pengine which found and returned the solution to the client. This pengine – still running – was then stored in the pool where it was indexed on the combination of the query and an integer indicating the number of solutions produced so far (i.e. 1 in this case). When a request for the second and third solution arrived, the actor manager picked a matching pooled pengine, used it to compute the solutions, and returned them to the client. Note that the second request could have come from any client, not necessarily from the one that requested the first solution. This is what makes the HTTP API stateless.

The maximum size of the pool is determined by the node's settings. To ensure that the load on the node is kept within limits, the oldest active pengines are terminated and removed from the pool when the maximum size is reached. This *may* mean that some solutions to some subsequent calls must be disposed of, but this will not hurt the general performance.

5 Previous Work

Technology-wise, Web Prolog can be seen as an attempt to rethink and redesign our work on `library(pengines)` [3], which is the library serving SWISH [5]. As a library for JavaScript-Prolog communication it works well enough to support SWISH, but it also makes promises that it cannot really live up to, in particular when it comes to concurrent and distributed programming. It is not possible, for example, to spawn two remote pengines and make them play ping-pong with each other. One way to put it is to say that `library(pengines)` fails to implement the actor model. A layer of predicates beneath the pengine abstraction in the form of a small set of programming primitives that support actor-based programming is a much better design, and in addition establishes a clear and direct link to Erlang.

6 Discussion

> Communicating Prolog engines is a great idea – this is more or less what Erlang started as – but I didn't like the idea of backtracking over nodes.
>
> *Joe Armstrong* (p.c. June 16, 2018)

The ability to backtrack over nodes must indeed be seen as part of the essence of Web Prolog. Basing the distribution of Prolog processes on the actor programming model seems to require this ability. If the idea of backtracking over nodes is abandoned, then the idea of backtracking *within* an actor or over actors *within* a node does not seem attractive either. Thus, the move to a functional language with its simpler syntax and deterministic operational semantics becomes the logical next step – a step that unfortunately, as it were, does away with the "logic" in "logic programming" and with a lot of useful features that a language such as Prolog provides.

We hasten to add that in no way should this be seen to imply that we think that Armstrong and the other inventors of Erlang made a *mistake* when they abandoned Prolog in favour of a simple functional language. Given that Prolog is fairly difficult to learn and to use correctly, 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 the right decision. After all, Erlang is a very successful programming language, more successful than Prolog when it comes to industrial uses.

Almost fifty years have gone by since Prolog was introduced as a promising language for AI, and more than thirty years have passed since Erlang was invented. Today, AI is booming again, Prolog has evolved considerably, the internet is much faster and a lot more reliable than it used to be, the multi-core hardware revolution is in full swing, and Erlang-style concurrency has emerged as a sensible way to program such hardware. Perhaps now is the right time for the idea of communicating Prolog processes and backtracking over

nodes to make a comeback. With this paper we are making an attempt to show how this idea might be realised.

6.1 A Hierarchy of Useful Abstractions

In Web Prolog, just like in Erlang, the actor is regarded as the fundamental unit of computation and as the single abstraction that solves the two problems of concurrency and distribution and provides a form of network transparency. In Web Prolog, just like in Erlang, other network-transparent programming abstractions can be built on top of the actor. In Web Prolog, the most prominent and universally useful such abstraction is the pengine, followed closely by the abstraction for making non-deterministic remote procedure calls. These are both abstractions that would not fit easily into Erlang, but which are natural in Web Prolog. Abstractions such as these can be compared to Erlang *behaviours*. They are not always easy to build, but once they are built they can easily be instantiated and tailored to specific tasks.

We note that these three abstractions – the actor, the pengine and the remote procedure call – form a hierarchy where predicates on the higher levels inherit some of their options from predicates on the lower levels. `rpc/3` inherits options from `pengine_spawn/2` and `pengine_ask/3`, and `pengine_spawn/2` inherits options from `spawn/3` in turn.

6.2 Web Prolog and the Programmable Prolog Web

A node has a dual identity. It can be seen not only as a Web Prolog runtime system but also as a node in the network forming what we will refer to as *the Prolog Web*. The traditional Web is distributed, decentralised and open, and these are traits we want the Prolog Web to share. Whereas distribution is nicely conceptualised in the actor model, and nicely handled by an actor programming language such as Erlang or Web Prolog, decentralisation and openness require features we choose to rely on the Web as such to contribute.

Here, the humble URI is a key concept, as it allows us to link a Web Prolog program to another Web Prolog program, a running actor to another running actor, or a Prolog query to its answers, in much the same way as HTML documents are linked to other HTML documents.

Another key feature of the Prolog Web is that communication among nodes relies only on HTTP and the WebSocket protocol. This allows it to pass through firewalls, and provides security-related features such as methods for authentication, HTTPS and CORS (Cross-Origin Request Sharing).[6]

Distributed programming in Erlang typically involves a number of nodes connected into a cluster. Erlang nodes usually rely on TCP/IP for transport and are, for reasons of security, assumed to be operating in a closed, trusted environment where we directly control the machines involved. In other words, when Erlang runs on a cluster, it is a cluster that is *closed*. In comparison, the Prolog Web might be seen as a cluster as well, but one that is as *open* as the Web itself.

6.3 Would the Prolog Web Scale?

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 allowing clients to limit the number of network roundtrips it takes to run a query to completion, by embracing the WebSocket transport protocol with its low overhead per message, by offering also a stateless HTTP 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 our high hopes for the scalability of the Prolog Web are not unfounded.

For the Prolog Web to be able to scale *really* well, nodes must also be able to spawn very many actors, creating and destroying actors must be fast, and the communication among them efficient. Since actors created by the Erlang virtual machine are famous for having exactly those properties, this is certainly yet another reason for us to look closely at Erlang.

An implementation of a Web Prolog node in Erlang might be interesting since it would most probably have a performance profile different from our implementation in SWI-Prolog. Interestingly, there is already Erlog – a fairly complete Prolog implementation in Erlang written by Robert Virding – which might serve as a point of departure.[7] Erlog is an interpreter so the basic Prolog machinery (e.g. unification and backtracking) is likely to be slower in Erlog compared to (say) SWI-Prolog, whereas the super-fast lightweight processes of Erlang have other advantages, probably allowing it to scale better to very many simultaneous clients on a network. For the networking part, we note that 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 for an Erlang implementation of Web Prolog.

The holy grail for a Web Prolog runtime system is a compiler targeting a virtual machine with BEAM-like properties, capable of producing code which when run will create processes as small and efficient as Erlang processes, yet with the useful capabilities that Prolog offers. We do not dare to guess whether building such a virtual machine is feasible.

6.4 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*

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

[6] https://en.wikipedia.org/wiki/Cross-origin_resource_sharing

[7] https://github.com/rvirding/erlog

behind. People both inside and outside the community have at various occasions voiced their fears about the future of Prolog, noting that there are too many incompatible systems around, resulting in a fragmented community and an ISO standard that few systems conform to.

We suggest rebranding as a strategy for reviving Prolog, and offer Web Prolog in the hope that it may serve as a *lingua franca* allowing different Prolog systems to communicate, and possibly aid the "defragmentation" of the community.

The strategy of rebranding as such is not a new idea. In fact, we would suggest that Elixir might be regarded as a rebranded Erlang. Since Elixir appears to be more popular than Erlang, rebranding seems to have worked. However, since we do not propose to change the syntax of Prolog, only its purpose, our approach to rebranding is different.

7 Summary and Future Work

In accordance with our rebranding strategy, we choose to present our summary in the form of two "elevator pitches".

> Imagine a dialect of Prolog with actors and mailboxes and send and receive – all the means necessary for powerful concurrent and distributed programming. Alternatively, think of it as a dialect of Erlang with logic variables, backtracking search and a built-in database of facts and rules – the means for logic programming, knowledge representation and reasoning. Also, think of it as a web logic programming language. This is what Web Prolog is all about.
>
> *Web Prolog – the elevator pitch*

> Imagine the Web wrapped in Prolog, running on top of a distributed architecture comprising a network of nodes supporting HTTP and WebSocket APIs, as well as web formats such as JSON. Think of it as a high-level Web, capable of serving answers to queries – answers that follow from what the Web "knows". Moreover, imagine it being programmable, allowing Web Prolog source code to flow in either direction, from the client to the node or from the node to the client. This is what the Prolog Web is all about.
>
> *The Prolog Web – the elevator pitch*

Our work on the design and implementation of Web Prolog has so far resulted in a somewhat sketchy language specification and a proof-of-concept demonstrator featuring a fairly comprehensive interactive tutorial. As for future work, our next goal is to make sure the demonstrator is robust and secure enough to allow people to play with the language online without having to download anything.

In parallel to investing more work into building something that can be used in production, we are considering making an early attempt to create a standard for Web Prolog, based on a suitable subset of ISO Prolog, but developed under the auspices of the W3C this time rather than ISO, or under a liaison between these organisations. As a first move in this direction, we might create a W3C Community Group,[8] as this appears to be an easy way to find out if enough interest can be generated among people of appropriate expertise.

A realistic but ambitious deadline for a standardisation effort would be to aim for 2022, the year when Prolog celebrates its 50th birthday. We find it difficult, in fact, to think of a better way to celebrate this occasion than to release version 1.0 of such a standard along with software implementing it.

We believe that Erlang technology might have something to contribute to Web Prolog, and, of course, that Prolog technology has something to contribute to Erlang (and by that we mean more than it has already contributed by once upon a time having inspired Erlang). Nowadays, there is not much contact between the Prolog community and the Erlang community. In the best of worlds, the language of Web Prolog might serve to open a new line of communication between the two communities.

Acknowledgments

The author is grateful to late Joe Armstrong for his enthusiastic support. Richard O'Keefe and Markus Triska provided encouraging and constructive comments and suggestions. Jan Wielemaker kindly helped with the implementation of the demonstrator, and he also came up with the original idea on which the stateless HTTP API is based. Anonymous reviewers offered a number of additional useful suggestions.

References

[1] Joe Armstrong. 2007. A history of Erlang. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*. 1–26. https://doi.org/10.1145/1238844.1238850

[2] Fred Hebert. 2013. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA.

[3] Torbjörn Lager and Jan Wielemaker. 2014. Pengines: Web Logic Programming Made Easy. *Theory and Practice of Logic Programming* 14, 4-5 (2014), 539–552. https://doi.org/10.1017/S1471068414000192

[4] Hans Svensson, Lars-Åke Fredlund, and Clara Benac Earle. 2010. A unified semantics for future Erlang. In *Erlang Workshop*, Scott Lystig Fritchie and Konstantinos F. Sagonas (Eds.). ACM, 23–32. http://dl.acm.org/citation.cfm?id=1863509

[5] Jan Wielemaker, Fabrizio Riguzzi, Bob Kowalski, Torbjörn Lager, Fariba Sadri, and Miguel Calejo. 2019. Using SWISH to realise interactive web based tutorials for logic based languages. *Theory and Practice of Logic Programming* 19, 2 (2019), 229–261. https://doi.org/10.1017/S1471068418000522

[6] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.

[8] https://www.w3.org/community