Web Prolog and the Prolog Web – the (draft) manual

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

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

Contents

1 Web Prolog Predicate APIs

1.1 Built-in Predicates for Programming with Actors

Predicate: `self/1` ACTOR

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

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

Predicate: `spawn/2-3` ACTOR

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

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

- `node(+URI)`
  Creates the process on the node pointed to by the URI. Default is `localhost`.
- `monitor(+Boolean)`
  If `true`, sends a `down` message to the parent process when the spawned process terminates. Default is `false`.
- `link(+Boolean)`
  If `true`, terminates all child processes (if any) upon termination of the spawned process. Default is `false`.
- `timeout(+Integer)`
  Terminates the spawned process (or the process of spawning a process) after `Integer` milliseconds.
- `src_list(+ListOfClauses)`
  Injects a list of Web Prolog clauses into the process.
- `src_text(+Atom)`
  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.
- `type(+Atom)`
  Indicates the type of the source to be injected into the process. Default is `'web-prolog'`. Note that some `src_*` options may not be compatible with other values of this option.

Predicate: `!/2` ACTOR

```
+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 Web Prolog term except a bare variable. The sending is asynchronous, i.e. `!/2` does not block waiting for a response but continues immediately. If any of the arguments is uninstantiated, an instantiation error is thrown. If a process named `Pid` does not exist, nothing happens.

Predicate: `raise/1` ACTOR

```
raise(+Message) is det.
```

Sends `Message` to the mailbox of the current process. Defined as

```
raise(Message) :-
    self(Pid),
    Pid ! Message.
```

Predicate: `return/1` ACTOR

```
return(+Message) is det.
```

Sends `Message` to the mailbox of the process that spawned the current process. Defined as

```
return(Message) :-
    '$return_to'(Pid),
    Pid ! Message.
```

Predicate: `receive/1-2` ACTOR

```
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 body of the receive clause is called. 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 *deferred*, i.e. left in the mailbox without any change in their contents or order. Valid options:

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

Note 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 two corner cases, consider the following 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.

Predicate: `exit/1-2` ACTOR

```
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 with a known pid or registered name, but only by its owner. If the process is monitored, `Reason` (which may be any ground term) will appear in the second argument of the `down` message sent to the parent of the process.

Predicate: `register/2` ACTOR

```
register(+Name, +Pid) is det.
```

Registers the process `Pid` under the name `Name`.

Predicate: `unregister/1` ACTOR

```
unregister(+Name) is det.
```

Removes the registered name `Name` associated with a process identifier.

Predicate: `flush/0` ACTOR

```
flush is det.
```

This is a utility predicate that allows a programmer to inspect the contents of the toplevel pengine to which the shell is attached.

1.1.1 Down messages

If `monitor(true)` is passed to `spawn/2-3`, the spawned process sends a message of the form `down(Pid,Status)` to the parent when it terminates. `Pid` is the pid of the child, and `Status` is one of:

- `false` if the `Goal` of the process has been completed and failed.
- `true` if the `Goal` of the process has been completed and succeeded.
- `exited(Term)` if the `Goal` of the process has been terminated using `exit/1` with `Term` as argument, or `exit/2` with `Term` as second argument.
- `exception(Term)` if the `Goal` of the process has been terminated due to an uncaught exception.

1.2 Built-in Predicates for Programming with Pengines

1.2.1 The Pengine Communication Protocol

Figure 1 depicts a statechart specifying the Prolog 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.[1]

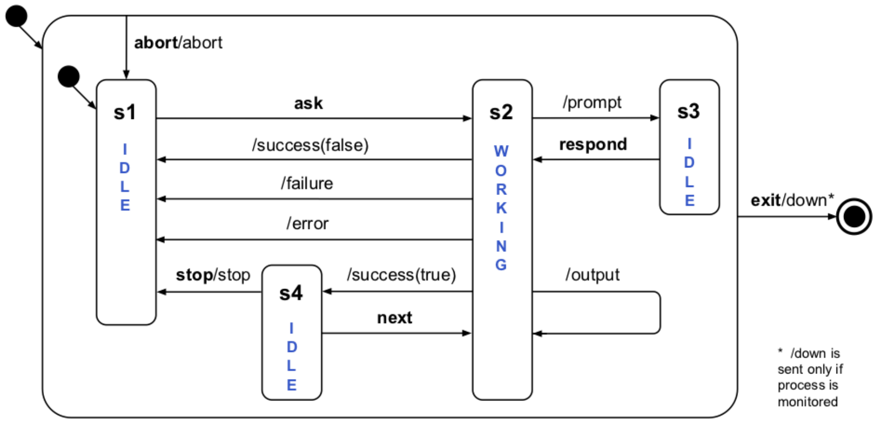

Fig. 1. Statechart specifying the PCP for a complete Web Prolog session. The transitions are labeled with *message types*. Types in bold are sent from the client to the pengine, whereas message types with a leading / goes in the opposite direction, from the pengine to the client.

Web Prolog comes with a handful of predicates that allows a client to create a pengine and then start talking to it. Several such predicates will be demonstrated in the next section.

1.2.2 Built-in Predicates for Pengines

Predicate: `pengine_spawn/1-2` ACTOR

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

Spawns a pengine and binds `Pid` to its pid. With just one exception, all options that can be passed to `pengine_spawn/2` are inherited from `spawn/3`. The only new option is `exit`.

[1] 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.

- `exit(+Boolean)`
  Determines if the pengine session must exit after having run a query to completion. Defaults to `true`. If set to `false`, more queries may follow.

Predicate: `pengine_ask/2-3` ACTOR

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

Asks the pengine `Pid` for solutions to `Query`. 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 query.
- `limit(+Integer)`
  By default, `pengine_ask/2-3` requests that *all* solutions to `Query` be computed and returned as a list of solutions embedded in an answer term of type `success`. By passing the limit option, the length of this list can be restricted to `Integer`.
- `return_to(+Pid)`
  Sends the answer to `Pid`. Default is the parent.

Variables in `Query` will not be bound. Instead, solutions and other kinds of output will be returned in the form of answer messages delivered to the mailbox of the process that called `pengine_spawn/2-3`.

- `success(Pid, Data, More, Info)`
  `Pid` refers to the pengine that succeeded in solving the query. `Data` 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`. `Info` is a structure containing optional extra information such as timing information.
- `failure(Pid)`
  `Pid` is the pid of the pengine that failed for lack of (more) solutions.
- `exception(Pid, Data)`
  `Pid` is the pid of the pengine throwing the exception. `Data` is the exception's error term.
- `output(Pid, Data)`
  `Pid` is the pid of a pengine running the goal that called `pengine_output/1`. `Data` is the term passed in the argument of `pengine_output/1` when it was called.
- `prompt(Pid, Data)`
  `Pid` is the pid of the pengine that called `pengine_input/2` and `Data` is the prompt.

Note that nothing stops a pengine from sending messages of a form different from the above to the target.

Predicate: `pengine_next/1-2` ACTOR

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

Asks pengine `Pid` for more solutions to `Query`. Valid options:

- `limit(+Integer)`
  By default, the value of the `limit` option is the same as for `pengine_ask/2-3`.
- `return_to(+Pid)`
  Send the answer to `Pid`. Default is the parent.

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` ACTOR

```
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` ACTOR

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

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

Predicate: `pengine_exit/1-2` ACTOR

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

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

Predicate: `pengine_output/1` ACTOR

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

Sends a message `output(Pid,Data)` to the parent process. `Pid` is the pid of the current process. This predicate is defined as follows:

```
pengine_output(Data) :-
    self(Pid),
    return(output(Pid, Data)).
```

Note that this is just a convenience predicate. A pengine, just like any other actor, may use `return/1` directly in order to send *any* term to its parent, or `!/2` to send any term to any process to which it has a pid.

Predicate: `pengine_input/2` ACTOR

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

Sends a message prompt(Pid,Prompt) to the parent process and waits for its input. Prompt may be any term (i.e. even a compound term). Pid is the pid of the current process. Data will be bound to the term that the parent process sends using pengine_respond/2.

Predicate: pengine_respond/2 ACTOR

```
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.

1.3 Built-in Predicates for Remote Procedure Calling

Predicate: rpc/2-3 ISOBASE

```
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.
```

The following options are valid:

- limit(+Integer) ISOBASE
  By default, rpc/2-3 will only make one trip to the remote node at URI in which it will (try to) compute all solutions to Query in order to cache them at the client. A query with n solutions and limit set to 1 would require n roundtrips if we wanted to see them all. With limit set to i, the same query would only require $ceiling(n/i)$ roundtrips.
- timeout(+Integer) ISOBASE
  Terminates the spawned process (or the process of spawning a process) after Integer milliseconds.
- src_list(+ListOfClauses) ISOBASE
  Injects a list of Web Prolog clauses into the process to be created. The node at URI must have ISOTOPE capabilities.
- src_text(+Atom) ISOBASE
  Injects the clauses specified by a source text into the process. The node at URI must have ISOTOPE capabilities.
- src_uri(+URI) ISOBASE
  Injects the clauses specified in the source code located at URI into the process to be created. The node at URI must have ISOTOPE capabilities.
- src_predicates(+List) ISOBASE
  Injects the local predicates denoted by List into the process to be created. List is a list of predicate indicators. The node at URI must have ISOTOPE capabilities.

- `type(+Atom)` ISOBASE
  Indicates the type of the source to be injected into the process. Default is `web-prolog`. Note that some `src_*` options may not be compatible with other values of this option.
- `monitor(+Boolean)` ACTOR
  Default is `false`, i.e. to not monitor. The node at `URI` must be another ACTOR node.
- `transport(+Atom)` ACTOR
  If `Atom` is `http` (default), the HTTP protocol will be used as transport, and if `Atom` is `ws`, a WebSocket connection will be used.
- `pid(-Pid)` ACTOR
  The `pid` option is passed with a free variable `Pid` which will be bound to the pid of the remote pengine when the call returns. Using the `pid` option breaks the abstraction for remote procedure calling, so it should be used with care. Note that if `transport` is `http`, `Pid` will be bound to `anonymous`. The node at `URI` must be another ACTOR node.

Other options are related to HTTP and are only relevant if the value of the `transport` option is `http`:[2]

- `connection(+Connection)` RELATION
  Specify the Connection header. Default is `close`. The alternative is `Keep-alive`.

Predicate: `promise/3-4` ACTOR

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

Makes an asynchronous RPC call to node `URI` with `Query`. This is a type of RPC which 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` ACTOR

```
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.

[2] Here we give only one example. Inspiration for other options may be had at `https://www.swi-prolog.org/pldoc/doc_for?object=http_open/3`.

1.4 Some Built-in ISO Prolog Predicates

Predicate: `use_module/1-2` ISOTOPE

```
use_module(+Resources) is det.
use_module(+Resources, +Imports) is det.
```

As in `https://www.swi-prolog.org/pldoc/man?section=import`. `Resources` is a list of terms of the form `uri(+URI)` and `text(+Atom)`.

Predicate: `op/3` ISOTOPE

```
op(+Precedence, +Type, :Name) is det.
```

As in ISO Prolog.

Predicate: `asserta/1` ACTOR

```
asserta(+Clause) is det.
```

As in ISO Prolog.

Predicate: `assert/1` ACTOR

```
assert(+Clause) is det.
```

As in ISO Prolog.

Predicate: `retract/1` ACTOR

```
retract(+Clause) is det.
```

As in ISO Prolog.

Predicate: `write/1` ISOBASE

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

This predicate is defined as a *noop* in the ISOBASE and ISOTOPE profiles, i.e. calling it does nothing. In the ACTOR profile, it behaves as in ISO Prolog.

Other predicates, such as `member/2`, `append/3`, `length/2`, `between/3`, `select/3`, `nth/3` and `maplist/2-8`, may well be treated as built-ins too.

2 Web Prolog Web APIs

2.1 The HTTP API

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 HTTP API that offers access to an arbitrary node on the Prolog Web is based on the realisation that URIs can be used to denote any resource, including a slice of solutions to a query.

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 a generic HTTP API to Prolog should take the form of such URIs as its point of departure.

In a request of the form `<BaseURI/ask?<Parameters>`, only the `query` parameter is mandatory and its value must be parsable into something that is callable.

Optional request parameters:

- `template=<String>` RELATION
  By default, if `format` is `prolog`, the value of `template` is identical to the query. If `format` is `json*`, the template is a structure mapping variable names to variables contained in the query.
- `offset=<Integer>` RELATION
  Specifies a slice of solutions starting at this offset. Default is `0`.
- `limit=<Integer>` RELATION
  By default, *all* solutions to the query is computed and returned as a list of solutions embedded in an answer term of type `success`. By using the limit parameter, the length of this list can be restricted to `Integer`.
- `timeout=<Integer>` RELATION
  Terminates the spawned process (or the process of spawning a process) after `Integer` milliseconds. Default is `infinite`.
- `format=<String>` RELATION
  Determines the format of answers. The value is either `prolog` or `json`. Default is `json`.
- `src_text=<String>` ISOTOPE
  Injects the clauses specified by a source text into the process.
- `options=<String>` RELATION
  ...

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"}],
  "more":true
}
```

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.

The value `true` of the `more` property indicates 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
}
```

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. 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 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
}
```

2.2 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.

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.[3] 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:[4]

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>
}));
```

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

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

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>`}.

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 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 1.2, we do not repeat the details here. A brief reminder should be sufficient. In a 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`.

3 Example Programs and Interactions

In this section, we show a number of examples of programs written in Web Prolog, and how a client may interact with processes running them.

3.1 An Implementation of `flush/0` and the Use of `self/1` and `!/2`

We start by showing an implementation of the built-in `flush/0` predicate, programmed in terms of `receive/2`:

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

In this program, 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. The value `0` of the `timeout` option passed to `flush/0` ensures that the loop terminates immediately if no messages remain in the mailbox. This prevents the call from blocking.

Here is an interaction using `self/1` and `!/2`, and `flush/0` for inspecting the mailbox of the process to which the shell is attached:

```
?- self(Self).
Self = 12732393.
?- $Self ! foo.
true.
?- $Self ! bar.
true.
?- flush.
Shell got foo
Shell got bar
true.
?- $Self ! baz.
true.
?- flush.
Shell got baz
true.
?-
```

Note also the use of another shell utility feature, borrowed from SWI-Prolog, which allows bindings resulting from the successful execution of a toplevel goal to be reused in future toplevel goals as `$Var`. Together with `flush/0`, this facility comes in handy during interactive programming in a Web Prolog shell.

3.2 Two Simple Ping Servers

A typical use of `spawn/2-3` is to call a locally or remotely defined Web Prolog procedure that specifies the 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. 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 response to ping messages can be written as below, using recursion as in the program to the left, or using a repeat-fail loop as in the program to the right:

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

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

The two servers are spawned in exactly the same way, and behave identically, like so:

```
?- spawn(ping_server, Pid).
Pid = 12763451.
?- self(Self).
Self = 98732093@'http://example.org'.
?- $Pid ! ping($Self).
```

```
true.
?- receive({Answer -> true}).
Answer = pong.
?- $Pid ! ping($Self).
true.
?- receive({Answer -> true}).
Answer = pong.
...
```

3.3 Two Simple Count Servers

Let us now turn to *stateful* servers. 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:

```
count_server(Count0) :-
    Count is Count0 + 1,
    receive({
        next(Pid) ->
            Pid ! Count,
            count_server(Count)
    }).
```

```
count_server :-
    between(1, infinite, Count),
    receive({
        next(Pid) ->
            Pid ! Count,
            fail
    }).
```

Below, we spawn the server to the right and take it for a trial run:

```
?- spawn(count_server, Pid).
Pid = 69774322.
?- self(Self).
Self = 98732093@'http://example.org'.
?- $Pid ! next($Self).
true.
?- receive({Answer -> true}).
Answer = 1.
?- $Pid ! next($Self).
true.
?- receive({Answer -> true}).
Answer = 2.
...
```

3.4 A Store Server

The following code shows how a simple stateful "memory cell" can be implemented:

```
store_server(Value0) :-
    receive({
        set(Value) ->
            store_server(Value);
        get(Pid) ->
            Pid !  Value0,
```

```
            store_server(Value0)
    }).
```

Below, the server is spawned:

```
?- spawn(store_server(undefined), Pid).
Pid = 56333258.
?- self(Self).
Self = 98732093@'http://example.org'.
?- $Pid ! set(5).
true.
?- $Pid ! get($Self).
true.
?- receive({Answer -> true}).
Answer = 5.
...
```

Here, we *cannot* use a failure driven loop.

3.5 A Query Server

```
query_server(Template^Query) :-
    call(Query),
    receive({
        next(Pid) ->
            Pid ! Template,
            fail
    }).
```

Below, the server is spawned with the query `?-between(1,infinite,N)`:

```
?- spawn(query_server(N^between(1,infinite,N)), Pid).
Pid = 44333558.
?- self(Self).
Self = 98732093@'http://example.org'.
?- $Pid ! next($Self).
true.
?- receive({Answer -> true}).
Answer = 1.
?- $Pid ! next($Self).
true.
?- receive({Answer -> true}).
Answer = 2.
...
```

Here, there's no escape. When lazily looping through the solutions to a query, we *must* use a failure driven loop.

3.6 A Priority Queue

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 (Hebert 2013). 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 toplevel process, and then calling `important/1`:

```
?- self(S),
   S ! 15-high, S ! 7-low, S ! 1-low, S ! 17-high.
S = 98732093@'http://example.org'.
?- important(Messages).
Messages = [high,high,low,low].
?-
```

3.7 Concurrent and Distributed Programming

In the following example, inspired by a user's guide to Erlang,[5] 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,
    format('Ping finished',[]).
ping(N, Pong_Pid) :-
    self(Self),
    Pong_Pid ! ping(Self),
    receive({
        pong ->
            format('Ping received pong',[])
    }),
    N1 is N - 1,
    ping(N1, Pong_Pid).
```

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

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

```
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.

3.8 A Generic Encapsulated Search Procedure

Below, we show how we can build a generic search predicate 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 `setup_call_cleanup/2` is here used not only to perform the query, but also to check if any choice points remain after the goal has been called or backtracked into.[6]

```
search(Query, Pid, Options) :-
    self(Self),
    spawn(query(Query, Pid, Self), Pid, [
        monitor(true),
        src_predicates([query/2])
        | Options
    ]).

query(Query, Self, Parent) :-
    setup_call_cleanup(true, Query, Det=true),
    (   var(Det)
    ->  Parent ! success(Self, Query, true),
        receive({
```

[6] See e.g. `http://www.swi-prolog.org/pldoc/man?predicate=setup_call_cleanup/2`

```
            next -> fail;
            stop -> Parent ! stopped(Self)
        })
    ;   Parent ! success(Self, Query, false)
    ).
```

By passing the src_predicate option, we send the predicate query/3 along for injection into the workspace of the process. Here is an example of how search/3 can be used to query a remote node:

```
?- search(human(Who), Pid, [
       node('http://n2.org')
   ]).
Pid = 34760012@'http://n2.org'.
?- flush.
Shell got success(34760012@'http://n2.org',human(plato),true)
true.
?- $Pid ! next.
true.
?- flush.
Shell got success(34760012@'http://n2.org',human(aristotle),false)
Shell got down(34760012@'http://n2.org',true)
true.
```

Note that the code for search/3 does not say anything about what should happen if an exception is thrown, which would for example be the case if the predicate called by the goal is not defined. However, 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.

3.9 The Birth, Life and Death of a Pengine

Below, we show how to create and interact with a pengine process that runs as a child of the current toplevel process:

```
?- pengine_spawn(Pid, [
       node('http://n2.org'),
       src_text("p(a). p(b). p(c)."),
       monitor(true),
       exit(false)
   ]),
   pengine_ask(Pid, p(X), [
       template(X),
       limit(1)
   ]).
Pid = 439752@'http://ex.org'.
?- flush.
Shell got success(439752@'http://n2.org',[a],true,[])
true.
?- pengine_next($Pid, [
       limit(2)
   ]),
```

```
   receive({Answer -> true}).
Answer = success(439752@'http://n2.org',[b,c],false,[]).
?-
```

There is quite a lot going on here. The `node` option passed to `pengine_spawn/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.

Given the pid returned by the `pengine_spawn/2` call, 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 toplevel). 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.

Since we passed the option `exit(false)` to `pengine_spawn/2` 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(439752@'http://n2.org',hello).
?-
```

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`:

```
?- pengine_ask($Pid, pengine_input('|:', Answer)),
   receive({Msg -> true}).
Msg = prompt(439752@'http://n2.org','|:').
?- pengine_respond($Pid, hi),
   receive({Msg -> true}).
Msg = success(439752@'http://n2.org',[pengine_input('|:',hi)],false,[]).
```

The pengine is still not dead so let us see what happens when a 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(439752@'http://n2.org').
?-
```

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

```
?- pengine_exit($Pid, goodbye),
   receive({Answer -> true}).
Answer = down(439752@'http://n2.org',exit(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.

3.10 Reducing the number of network roundtrips

Often, we want to perform as few network roundtrips as possible, and then it makes sense to go with the default limit of `infinite`:

```
?- pengine_spawn(Pid, [
        node('http://n2.org'),
        src_text("p(a). p(b). p(c).")
   ]),
   pengine_ask(Pid, p(X), [
        template(X)
   ]).
Pid = 985752@'http://n2.org'.
?- flush.
Shell got success(985752@'http://n2.org',[a,b,c],false)
true.
```

However, there are times when this will not work since there is an infinite number of solutions that would take an infinite time to compute:

```
?- pengine_spawn(Pid, [
        node('http://n2.org')
   ]),
   pengine_ask(Pid, between(1,infinite,N), [
        template(N)
   ]).
Pid = 597562@'http://n2.org'.
?-
```

In this case, a stack limit will likely be exceeded, and an error message to this effect will appear in the mailbox of the caller. Still, this can easily be handled if the `limit` option is set to a suitable value (e.g. 100 or 1000 in this case).

3.11 Pengines and the Message Deferring Mechanism

One consequence of the message deferring mechanism (which relies on the selective receive) 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 = 37123398.
```

```
?- pengine_next($Pid).
?- flush.
true.
?- pengine_ask($Pid, human(Who), [limit(1)]).
true.
?- flush.
Shell got success(37123398,[human(socrates)],true,[])
Shell got success(37123398,[human(plato)],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”.[7] 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.

3.12 An Implementation of rpc/2-3 over the Stateful WebSocket API

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) :-
    rpc(URI, Query, []).

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


wait_answer(Query, Pid) :-
    receive({
        failure(Pid) -> fail;
        exception(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)
    }).
```

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

3.13 Some Example Uses of the HTTP API

Here is how the process running the shell can ask `http://n1.org` for the first solution to the query `?-mortal(Who)`:

```
GET http://n1.org/ask?query=mortal(Who)&format=json
```

Since the value of the `format` parameter is `json`, the answer response is returned in the form of a JSON structure:

```
{ "type":"success",
  "pid":"anonymous",
  "data":[{"Who":"socrates"}],
  "more":true
}
```

The value of the `type` property shows that the query succeeded. The value of the `pid` property is where we would normally find the process identifier of the process which computed the solution(s). However, answers returned as responses to HTTP requests made to a node on the Prolog Web are special in that they hide the identity of this process. Therefore, in this case, the value of the `pid` property is not a real pid, but the string `anonymous`. The solution(s) to the query appear as the value of the `data` property and the value `true` of the `more` property indicates there may be more solutions to the query.

To ask for more solutions, we can make a new GET request, setting `offset` to `1` this time. (It was the default `0` in the previous request.) As we want to retrieve more than one solution, we also set the value of `limit` to `10`:

```
GET http://n1.org/ask?query=mortal(Who)&offset=1&limit=10&format=json
```

This request results in:

```
{ "type":"success",
  "pid":"anonymous",
  "data":[{"Who":"plato"},{"Who":"aristotle"}],
  "more":false
}
```

The node responded with two more solutions, and the value `false` of the `more` property shows that there are no more solutions to be found. If we insist anyway, by setting `offset` to `3` (or `4` or `5` or ...), we would receive a response with a content of the form {`"type":"failure","pid":"anonymous"`}.

Note that the `limit` option may save us some network roundtrips when we are dealing with queries with more than one solution. Also, for searches when we want to paginate our results, this is nice to have.

When making API requests from Prolog (rather than from a non-prolog programming language such as JavaScript or Python), we choose to ask for answers encoded in Prolog rather than JSON. In response to a request such as

```
GET http://n2.org/ask?query=human(Who)&offset=1&limit=10&format=prolog
```

we get the following term:

```
success(anonymous,[human(plato),human(aristotle)],false)
```

3.14 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:

```
 1  var ws = new WebSocket(’ws://example.org/ws’,’PCP-1.0’);
 2  ws.onopen = function (message) {
 3    ws.send(JSON.stringify({
 4      command: ’pengine_spawn’,
 5      options: ’[format(json)]’
 6    }));
 7  };
 8  ws.onmessage = function (message) {
 9    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.[8] 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 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, 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:

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

```
[{"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://example.org/ws','PCP-0.2');
var program =
   `echo :-
        pengine_input('Input a term!', Something),
        (   Something == null
        ->  true
        ;   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({
      command: 'pengine_respond',
      pid: event.pid,
      term: response
    }));
  } else if (event.type == 'output') {
    document.getElementById("output").innerHTML +=
        JSON.stringify(event.data) + "<br/>";
  }
};
```

4 Open issues

4.1 Additional options

- `once=<Boolean>` RELATION
  Tells the node that you are only going to ask the number of solutions specified by the value of the limit parameter once.

Make `Goal` succeed at most once. This is not the same as running a goal `once(Goal)`. Passing the option will (if it succeeds without errors) 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. Depending on the implementation, this option may help to manage the node's resources.

- `once(+Boolean)`
  Make `Query` succeed at most once. This is not the same as running a query `once(Goal)`. Passing the option will (if it succeeds without errors) produce an answer containing the maximum number of solutions suggested by the limit option, whereas the query `once(Goal)` will only provide one solution. The purpose, in both cases, is to avoid leaving unwanted choice points around.
- `once(+Boolean)` ISOBASE
  Passing this option will make `rpc/2-3` produce an answer to `Query` with at most the number of solutions suggested by the `limit` option.
- `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`.

4.2 Modules

References

AGHA, G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

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.* 235–245.

KIFER, M., DE BRUIJN, J., BOLEY, H., AND FENSEL, D. 2005. *A Realistic Architecture for the Semantic Web.* Springer Berlin Heidelberg, Berlin, Heidelberg, 17–29.

LOMBARDI, A. 2015. *WebSocket: Lightweight Client-server Communications.* O'Reilly Media, Inc.