Part 1: Web Prolog

Torbjörn Lager

Department of Philosophy, Linguistics and
Theory of Science, University of Gothenburg

Outline

- Why am I here?
- SWISH and library Pengines
- The Web
- Web Prolog
- Erlang comparison
- Actors, pengines and nondet RPC
- Promises

Why am I here?

- Markus Triska (University of Vienna), who read my manuscript and seems to like it, wrote:

> “One researcher that I think you should study closely and eventually definitely also contact is Kazunori Ueda from Waseda University. Please read his "Logic/Constraint Programming and Concurrency: The Hard-Won Lessons of the Fifth Generation Computer Project" from FLOPS 2016. He has tremendous experience with parallelism, and one can tell from his writing that he is ready for some sort of culmination of his work. I think Web Prolog may well be what sets off this process, and it also can only help to have strong ties with Japanese researchers in this area.”

Some 5 years ago…

- In 2013, I wrote the first version of **SWISH** – an web-based programming environment for Prolog. I presented it as a homage to SWI-Prolog – which was then my favorite programming platform – on its 25th birthday.

- I also wrote a library that made it easy to develop web applications using Prolog as a back-end. That became `library(pengines)`.

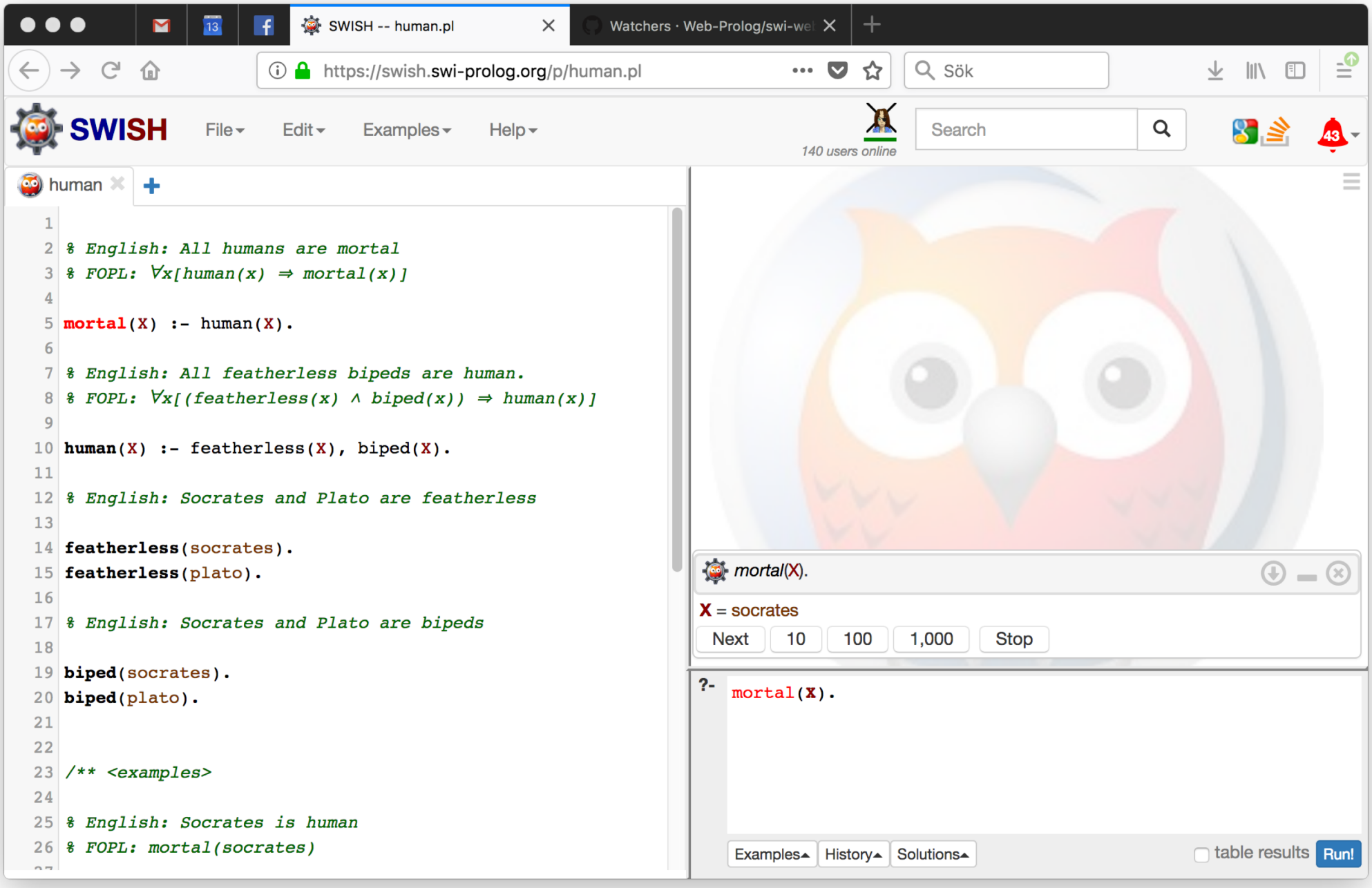
- Jan Wielemaker, the creator of SWI-Prolog, liked what he saw, so I went down to Amsterdam and started a collaboration with him.

- Two publications:

  - Torbjörn Lager and Jan Wielemaker (2014), **Pengines: Web Logic Programming Made Easy**, In *Theory and Practice of Logic Programming*, 14:4-5.

  - 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**. In *Theory and Practice of Logic Programming,* 19:2

SWISH


SWISH -- human.pl
Watchers · Web-Prolog/swi-wel
https://swish.swi-prolog.org/p/human.pl
Sök
SWISH
File
Edit
Examples
Help
140 users online
Search
43
human
1 % English: All humans are mortal
2 % FOPL: ∀x[human(x) ⇒ mortal(x)]
3
4 mortal(X) :- human(X).
5
6 % English: All featherless bipeds are human.
7 % FOPL: ∀x[(featherless(x) ∧ biped(x)) ⇒ human(x)]
8
9 human(X) :- featherless(X), biped(X).
10
11 % English: Socrates and Plato are featherless
12
13 featherless(socrates).
14 featherless(plato).
15
16 % English: Socrates and Plato are bipeds
17
18 biped(socrates).
19 biped(plato).
20
21
22 /** <examples>
23
24 % English: Socrates is human
25 % FOPL: mortal(socrates)
mortal(X).
X = socrates
Next
10
100
1,000
Stop
?-
mortal(X).
Examples
History
Solutions
table results
Run!

Why am I not 100% happy?

- Being able to talk to Prolog from a browser is nice, but in my opinion it's not *only* what Prolog on the Web should be about.
- The library behind SWISH doesn't really fulfil my initial expectations. The pengines behind SWISH are not the pengines I wanted to have there. I made design mistakes. I want to correct them.
- Two years ago, inspired by the Erlang programming language, I found a *much* better approach, and decided to design a new Prolog *language* rather than just a library.
- In addition, I see this as my attempt to contribute to the solving of the crisis of Prolog.


Web Prolog
and the programmable
Prolog Web

The Web

- The biggest distributed programming system ever constructed
- Billions of daily users and millions of contributors
- Decentralised and open (by default)
- As a consequence, JavaScript has become the most commonly used programming language on Earth. Even back-end developers are more likely to use it than any other language. (StackOverflow, 2016)
- JavaScript is undoubtedly *the* web programming language of our times.
- JavaScript uses ECMAScript standard to ensure interoperability.

From a CfP for the 2nd International Workshop on Logic Programming Tools for Internet Applications
(in conjunction with ICLP'97, Leuven, Belgium, 1997*)*

This workshop is the second in a series intended to explore the elective affinities between Logic Programming and Internet technologies with emphasis on **enhancing the World Wide Web with knowledge, deductive abilities and superior forms of interactive behavior**. With the paradigm shift to highly inter-connected computers and programming tools, **logic programming languages have a unique opportunity to contribute to practical Internet application development. Simplicity, remote executability, robustness, automatic memory management**, are among the features some LP languages share with emerging tools like Java. **Superior meta-programming and high-level distributed programming facilities, built-in grammars and dynamic databases, declarative semantics are among their competitive advantages**.


Web Prolog
and the programmable
Prolog Web

Web Prolog – the elevator pitch

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.

The Erlang programming language

- Invented by Joe Armstrong, Robert Virding and Mike Williams in the second half of the eighties.
- Started out as a proprietary language within Ericsson, but was released as open source in 1998.
- Can be characterised as an *actor programming language* – famous for concurrency and distribution.
- Has its roots in Prolog, but is a functional language.
- Currently *much* more popular than Prolog.

The syntax of Prolog vs Erlang

Prolog

```
% length

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

% naive reverse

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

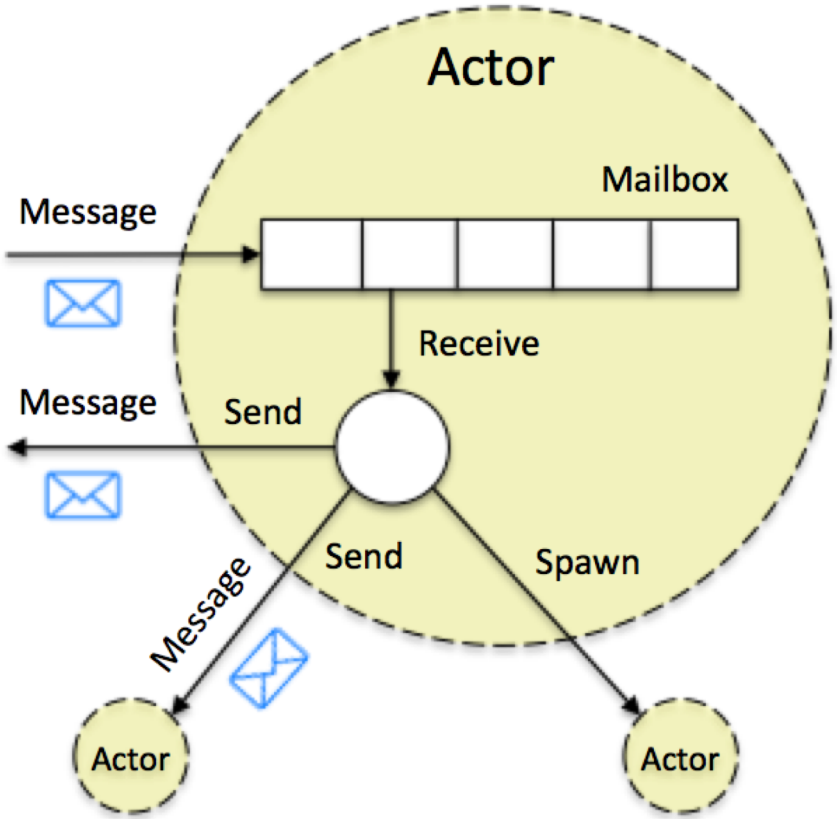
Erlang

```
% length

length([]) -> 0;
length([_|T]) ->
    1 + length(T).


% naive reverse

reverse([]) -> [];
reverse([H|T]) ->
    reverse(T) ++ [H].
```

The road from Prolog to Erlang

- Features that were *removed* from Prolog in order to arrive at Erlang:
  - Built-in backtracking search
  - Unification
  - Logic-based knowledge representation
  - Reasoning
  - Meta-programming
  - User defined operators
  - The term expansion mechanism
  - Definite Clause Grammar (DCG)
- Armstrong: “Erlang is a concurrent programming language with a functional core.”
- Me: “Web Prolog is a concurrent language with a relational/logic programming core.” (With all the above features intact!)

An actor



- The fundamental unit of computation in Erlang and Web Prolog
- Sequential processing internally
- A unique name (pid)
- Isolated memory – no sharing
- Communicate by sending each other asynchronous messages
- They all have a mailbox
- An actor can create new actors
- Location transparency

A nice metaphor

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

Fred Hebert

Spawning and messaging in Web Prolog

Node-resident code

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

Shell (attached to a pengine)

```
?- spawn(server, Pid).
Pid = '32057121'.

?- self(Self).
Self = '10785476'@'http://local.org'.

?- $Pid ! ping($Self).
true.

?- flush.
Shell got pong
true.

?-
```

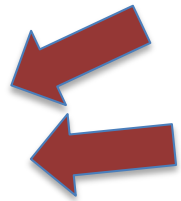
Spawning and messaging in Web Prolog

Injected code

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

Shell (attached to a pengine)

```
?- spawn(server, Pid, [
        node('http://remote.org'),
        src_predicates([server/0])
    ]).
Pid = '18566831'@'http://remote.org'.

?- self(Self).
Self = '31224736'@'http://local.org'.

?- $Pid ! ping($Self).
true.

?- flush.
Shell got pong
true.

?-
```



Does it look and behave like Erlang?

"Reading the code in the book was fun – I had to do a double take – was I reading Erlang or Prolog – they often look pretty much the same".

Joe Armstrong

Priority queue

Web Prolog

```
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 = [])
    ]).
```

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

?- important(Messages).
Messages = [high,high,low,low].
?-
```

Erlang

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

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

Erlang example from "*Learn You Some Erlang for great good!*" *by Fred Hébert*

Fridge simulation

Web Prolog

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

Erlang

```
fridge(FoodList) ->
    receive
        {From, {store, Food}} ->
            From ! {self(), ok},
            fridge([Food|FoodList]);
        {From, {take, Food}} ->
            case lists:member(Food, FoodList) of
                true ->
                    From ! {self(), {ok, Food}},
                    fridge(lists:delete(Food, FoodList));
                false ->
                    From ! {self(), not_found},
                    fridge(FoodList)
            end;
        terminate ->
            ok
    end.
```

Erlang example from "*Learn You Some Erlang for great good!*" *by Fred Hébert*

Ping-pong example

Web Prolog

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

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

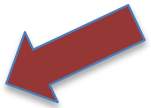
Erlang

```
-module(tut15).
-export([start/0, ping/2, pong/0]).


ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished");
ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong")
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong finished");
        {ping, Ping_PID} ->
            io:format("Pong received ping"),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

Ping-pong example

Web Prolog

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

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



Erlang

```
-module(tut15).
-export([start/0, ping/2, pong/0]).


ping(0, Pong_PID) ->
    Pong_PID ! finished,
    io:format("ping finished");
ping(N, Pong_PID) ->
    Pong_PID ! {ping, self()},
    receive
        pong ->
            io:format("Ping received pong")
    end,
    ping(N - 1, Pong_PID).

pong() ->
    receive
        finished ->
            io:format("Pong finished");
        {ping, Ping_PID} ->
            io:format("Pong received ping"),
            Ping_PID ! pong,
            pong()
    end.

start() ->
    Pong_PID = spawn(tut15, pong, []),
    spawn(tut15, ping, [3, Pong_PID]).
```

Programming abstractions

- Actors – the loci of computation in Web Prolog
- Pengines – actors adhering to the PCP protocol
- Non-deterministic remote procedure calls (NDRPC)

They are all related!

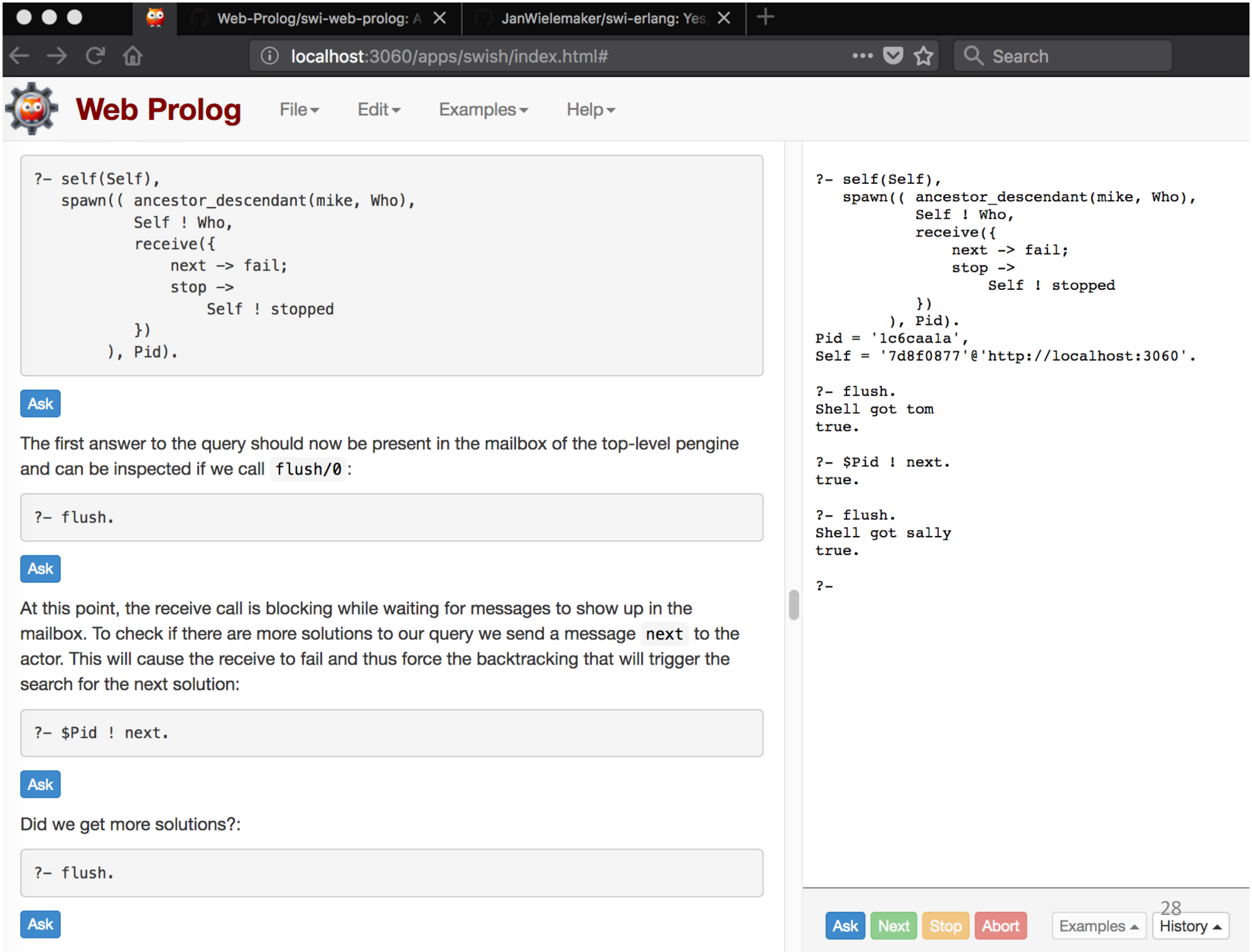
Handling non-determinism

```
?- self(Self),
   spawn(( p(X),
           Self ! X,
           receive({
               next -> fail;
               stop -> Self ! stopped
           })
         ), Pid).
Pid = '10351611',
Self = '31224736'@'http://local.org'.

?- flush.
Shell got a
true.

?- $Pid ! next.
true.

?- flush.
Shell got b
true.

?- $Pid ! stop,
   receive({Message -> true}).
Message = stopped.
```

Handling non-determinism

```
?- self(Self),
   spawn(( p(X),
           Self ! X,
           receive({
               next -> fail;
               stop -> Self ! stopped
           })
         ), Pid).
Pid = '10351611',
Self = '31224736'@'http://local.org'.

?- flush.
Shell got a
true.

?- $Pid ! next.
true.

?- flush.
Shell got b
true.

?- $Pid ! stop,
   receive({Message -> true}).
Message = stopped.
```

`receive/1-2` is semideterministic, and calling `fail/0` here causes backtracking


Web-Prolog/swi-web-prolog: A
JanWielemaker/swi-erlang: Yes
localhost:3060/apps/swish/index.html#
Search
Web Prolog
File
Edit
Examples
Help
?- self(Self),
spawn((ancestor_descendant(mike, Who),
Self ! Who,
receive({
next -> fail;
stop ->
Self ! stopped
})
), Pid).
Ask
The first answer to the query should now be present in the mailbox of the top-level pengine and can be inspected if we call flush/0 :
?- flush.
Ask
At this point, the receive call is blocking while waiting for messages to show up in the mailbox. To check if there are more solutions to our query we send a message next to the actor. This will cause the receive to fail and thus force the backtracking that will trigger the search for the next solution:
?- $Pid ! next.
Ask
Did we get more solutions?:
?- flush.
Ask
?- self(Self),
spawn((ancestor_descendant(mike, Who),
Self ! Who,
receive({
next -> fail;
stop ->
Self ! stopped
})
), Pid).
Pid = '1c6caala',
Self = '7d8f0877'@'http://localhost:3060'.
?- flush.
Shell got tom
true.
?- $Pid ! next.
true.
?- flush.
Shell got sally
true.
?-
Ask
Next
Stop
Abort
Examples
History
28

Generic encapsulated search

```
% search(+Query, -Pid, +Options)

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

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

```
?- search(p(X), Pid, [
        node('http://remote.org')
   ]).
Pid = 'b0ffdb86'@'http://remote.org').

?- flush.
Shell got success('b0ffdb…',p(a),true)
true.

?- $Pid ! next.
true.

?- flush.
Shell got success('b0ffdb…',p(b),true)
true.

?- $Pid ! stop.
true.

?- flush.
Shell got stopped('b0ffdb86…')
Shell got down('b0ffdb86…', true)
true.

?-
```

Generic encapsulated search

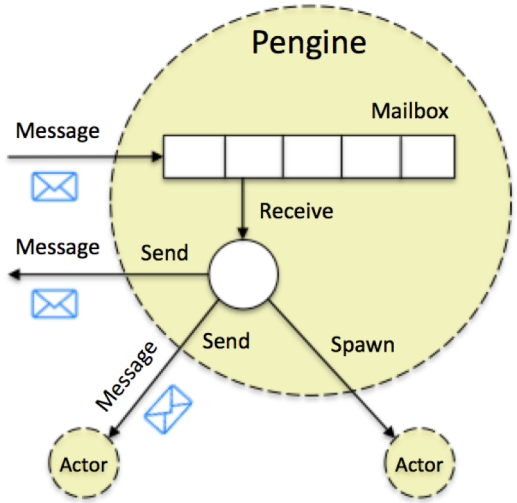
```
% search(+Query, -Pid, +Options)

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

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

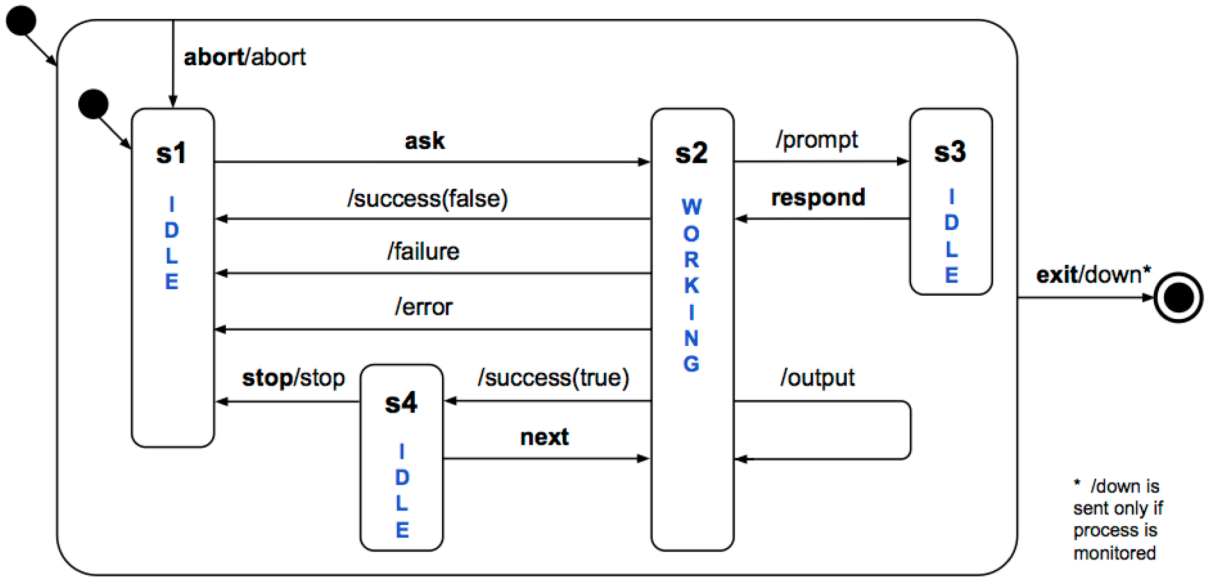
But there is no need for this!

```
?- search(p(X), Pid, [
        node('http://remote.org')
   ]).
Pid = 'b0ffdb86'.

?- flush.
Shell got success('b0ffdb86',p(a),true)
true.

?- $Pid ! next.
true.

?- flush.
Shell got success('b0ffdb86',p(b),true)
true.

?- $Pid ! stop.
true.

?- flush.
Shell got stopped('b0ffdb86')
Shell got down('b0ffdb86', true)
true.

?-
```

Enter pengines

- A pengine is special *kind* of actor characterised by the Prolog Communication Protocol (PCP)





This is what makes a pengine into a generic encapsulated *Prolog session* – a first-class Prolog top-level

Working with pengines

```
?- pengine_spawn(Pid, [
       node('http://remote.org')
   ]),
   pengine_ask(Pid, p(X), [
       template(X)
   ]).
Pid = '7528c178'@'http://remote.org'.

?- flush.
Shell got success('7528c178'@'http://remote.org',[a], true)
true.

?- pengine_next($Pid, [
       limit(2)
   ]).
true.

?- flush.
Shell got success('7528c178'@'http://remote.org',[b,c], false)
true.
```

An effect of deferring messages

```
?- pengine_next($Pid, [
       limit(2)
   ]).

?- flush.
true.

?- pengine_ask($Pid, p(X), [
       template(X)
   ]).

?- flush.
Shell got success('7528c178'@'http://remote.org',[a], true)
Shell got success('7528c178'@'http://remote.org',[b,c], false)
true.
```

Non-deterministic remote procedure calls (NDRPC)

```
?- rpc('http://remote.org', p(X)).
X = a ;
X = b ;
X = c.
```

Has to make three network round trips

```
?- rpc('http://remote.org', p(X), [
       limit(10)
   ]).
X = a ;
X = b ;
X = c.
```

Needs only one network round trip

```
?- rpc('http://remote.org', (p(X),q(X)), [
       src_text("q(a). q(b).")
   ]).
X = a ;
X = b.
```

Inject code

```
?- rpc(localnode, p(X), [
       src_uri('http://remote.org/src')
   ]).
X = a ;
X = b ;
X = c.
```

Runs locally, with code fetched from a URI. Needs only one network round trip

How is NDRPC implemented?

```
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)
    }).
```

Promises (Futures)

```
?- promise('http://remote.org', p(X), Reference, [
       offset(1)
   ]).
Reference = 'f7780f96'@'http://remote.org'.

?- yield($Reference, Answer).
Answer = success(anonymous,[p(b)],true).

?- promise('http://remote.org', p(X), Reference, [
       offset(2)
   ]).
Reference = 'f7780f96'@'http://remote.org'.

?- yield($Reference, Answer).
Answer = success(anonymous,[p(c)],false).

?-
```

Programming abstractions

- Actors – the loci of computation in Web Prolog
- Pengines – actors adhering to the PCP protocol
- Non-deterministic remote procedure calls (NDRPC)

They are all related!

Outline for Part 2

- The Prolog Web
- Two kinds of Web APIs
- The Prolog Web and the Semantic Web
- Voice Uis for talking to the Prolog Web
- Statecharts, Web Prolog and the Prolog Web
- The crisis of Prolog
- Rebranding Prolog
- Standardising Web Prolog
- Celebrating Prolog

Thank you!

(And don’t forget, I’ll be happy to demonstrate the proof-of-concept implementation while I’m here!)