SIMON THOMPSON

MAKING IT LAZY: NEVER EVALUATE ANYTHING MORE THAN ONCE

fun

https://github.com/simonjohnthompson/streams

FUNCTIONS AS DATA

“Functions are first-class citizens”

A function actively represents behaviour of some sort, and we deal with it just like any other kind of data.

Original image: https://www.thishopeanchors.com/single-post/2017/04/06/Rock-Paper-Scissors

What is a strategy?

Random

Echo

No repeats

Statistical

...

What is a strategy?

We choose what to play, depending on your last move, or the history of all your moves.

What is a strategy?

```
-type play() :: rock | paper | scissors.

-type strategy() :: fun(([play()]) -> play()).
```

We choose what to play, depending on your last move, or the history of all your moves.

Random

Echo

No repeats

Statistical

...

```
echo([]) ->
     random_play();
echo([X|_Xs]) ->
    X.

beat([]) ->
    random_play();
beat([X|_]) ->
    case X of
    rock -> scissors;
    paper -> rock;
    scissors -> paper
    end.
```

```erlang
% The second argument here is the accumulated input from the player
% Note that this function doesn't cheat: the Response is chosen
% before the Play from the player.

-spec interact(strategy(),[play()]) -> ok.

interact(Strategy,Xs) ->
    Response = Strategy(Xs),
    {ok,[Play|_]} = io:fread('play rock, paper, scissors, stop: ',"~a"),
    case Play of
      stop -> ok;
      _ ->
        Result = result({Play,Response}),
        io:format("Machine plays ~p, result is ~p~n",[Response,Result]),
        interact(Strategy,[Play|Xs])
    end.
```

What is a strategy combinator?

Choose randomly between these strategies.

Apply them all and choose most popular result.

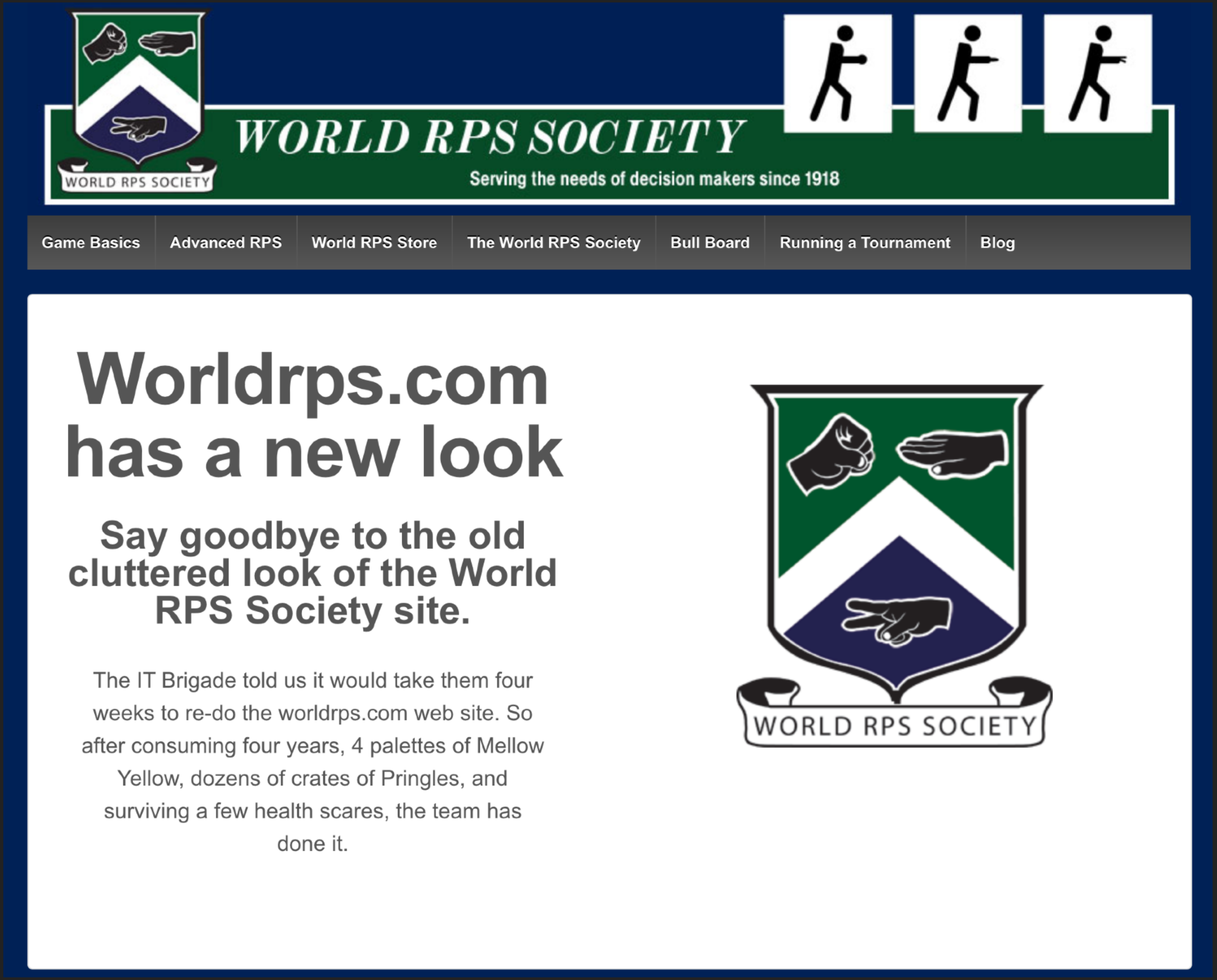
Replay each of these strategies on the history so far and apply the one that's been best so far.

Take home

Toy example

Generality: not just a finite set . . .

Up a level: combining strategies



http://worldrps.com

EVALUATION ON DEMAND

function evaluation in Erlang

function evaluation in Erlang

evaluate the arguments before the body

```
switch(N,Pos,Neg) ->
    case N>0 of
        true -> Pos;
        _    -> Neg
    end.
```

function evaluation in Erlang

evaluate the arguments before the body

```
switch(N,Pos,Neg) ->
    case N>0 of
        true -> Pos;
        _    -> Neg
    end.
```

fully evaluate the argument

```
sum_first_two([A,B|_Rest])
    -> A+B.
```

but if an argument is a
function then it's
passed unevaluated.

but if an argument is a **function** then it's passed unevaluated.

```
fun () -> Stuff end
```

but if an argument is a **function** then it's passed unevaluated.

```
fun () -> Stuff end
```

```
fun () -> Stuff end ()
```

DELAY!

a lazy switch

```
-spec lswitch(number(),fun(() ->T),fun(() ->T)) -> T.


lswitch(N,Pos,Neg) ->
    case N>0 of
        true -> Pos();
        _    -> Neg()
    end.
```

a lazy switch

```
-spec lswitch(number(),fun(() ->T),fun(() ->T)) -> T.

lswitch(N,Pos,Neg) ->
    case N>0 of
        true -> Pos();
        _    -> Neg()
    end.
```

```
lex1() -> lswitch(1,fun() -> 3+4 end,fun () -> 1/0 end).
```

```erlang
-spec lswitch(number(),fun(() ->T),fun(() ->T)) -> T.


lswitch(N,Pos,Neg) ->
    case N>0 of
        true -> Pos();
        _    -> Neg()
    end.
```

```erlang
-define(switch(N,Pos,Neg),
        lswitch(N,fun() -> Pos end,fun() -> Neg end)).
```

```erlang
-spec lswitch(number(),fun(() ->T),fun(() ->T)) -> T.

lswitch(N,Pos,Neg) ->
    case N>0 of
        true -> Pos();
        _    -> Neg()
    end.
```

```erlang
-define(switch(N,Pos,Neg),
        lswitch(N,fun() -> Pos end,fun() -> Neg end)).
```

```erlang
lex2() -> ?switch(1,3+4,1/0).
```

STREAMS

Original image: http://www.metso.com/services/spare-wear-parts-conveyors/conveyor-belts/

streams

build

```
cons(X,Xs) ->
  fun() -> {X,Xs} end.
```

streams

build

```
cons(X,Xs) ->
  fun() -> {X,Xs} end.
```

deconstruct

```
head(L) ->
  case (L()) of
    {H,_} -> H
  end.

tail(L) ->
  case (L()) of
    {_,T} -> T
  end.
```

streams

build

```
-define(cons(X,Xs),
        fun() -> {X,Xs} end).
```

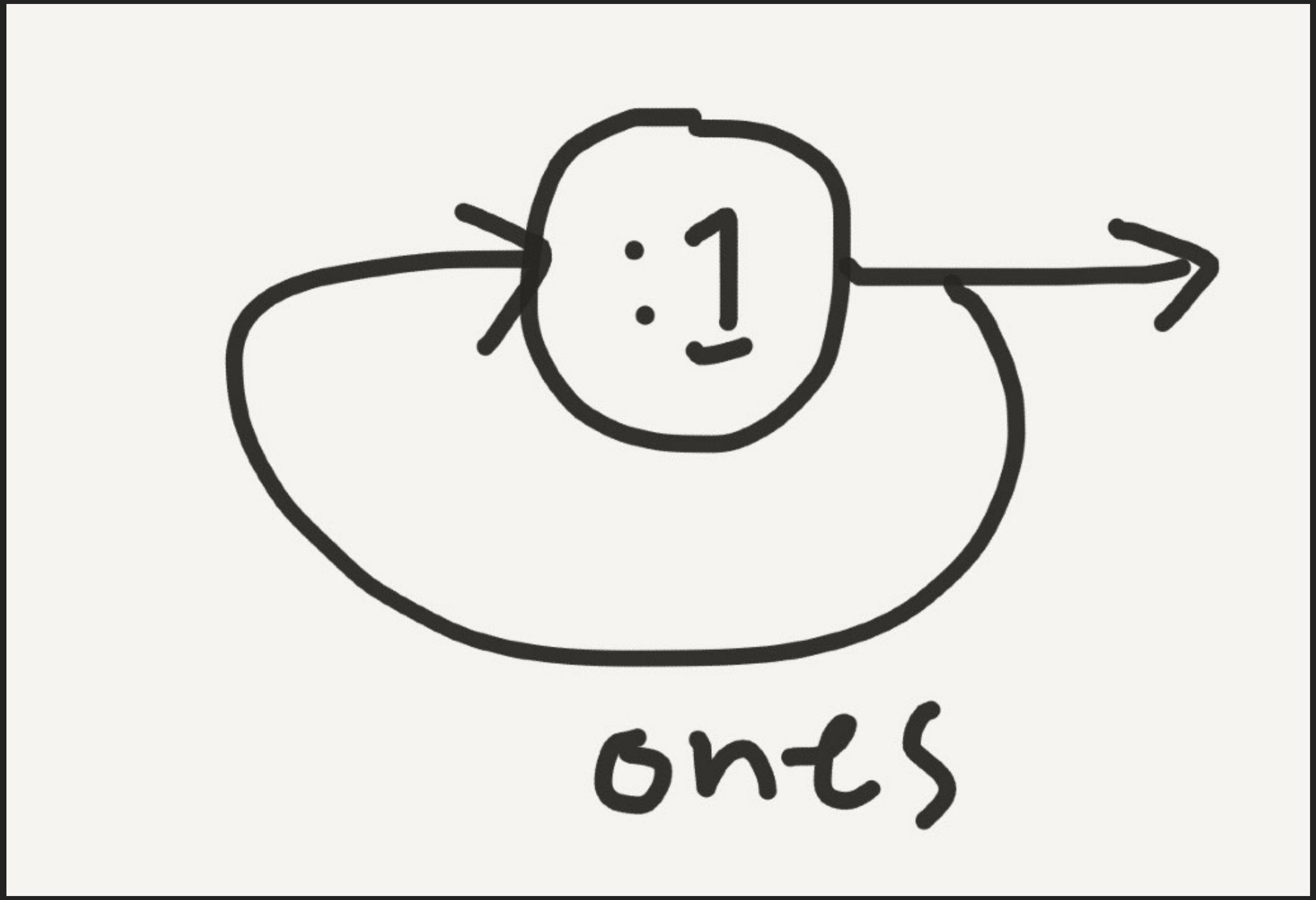
deconstruct

```
head(L) ->
  case (L()) of
    {H,_} -> H
  end.

tail(L) ->
  case (L()) of
    {_,T} -> T
  end.
```

```erlang
ones() ->
  ?cons(1,ones()).
```

1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...

```
ones() ->
  ?cons(1,ones()).
```



```
ns(N) ->
  ?cons(N,ns(N+1)).
```

42, 43, 44, 45, 46, 47, 48, 49, 50, . . .

2, 3, 5, 7, 11,

13, 17, 19,

23, 29, 31,

37, 41, 43,

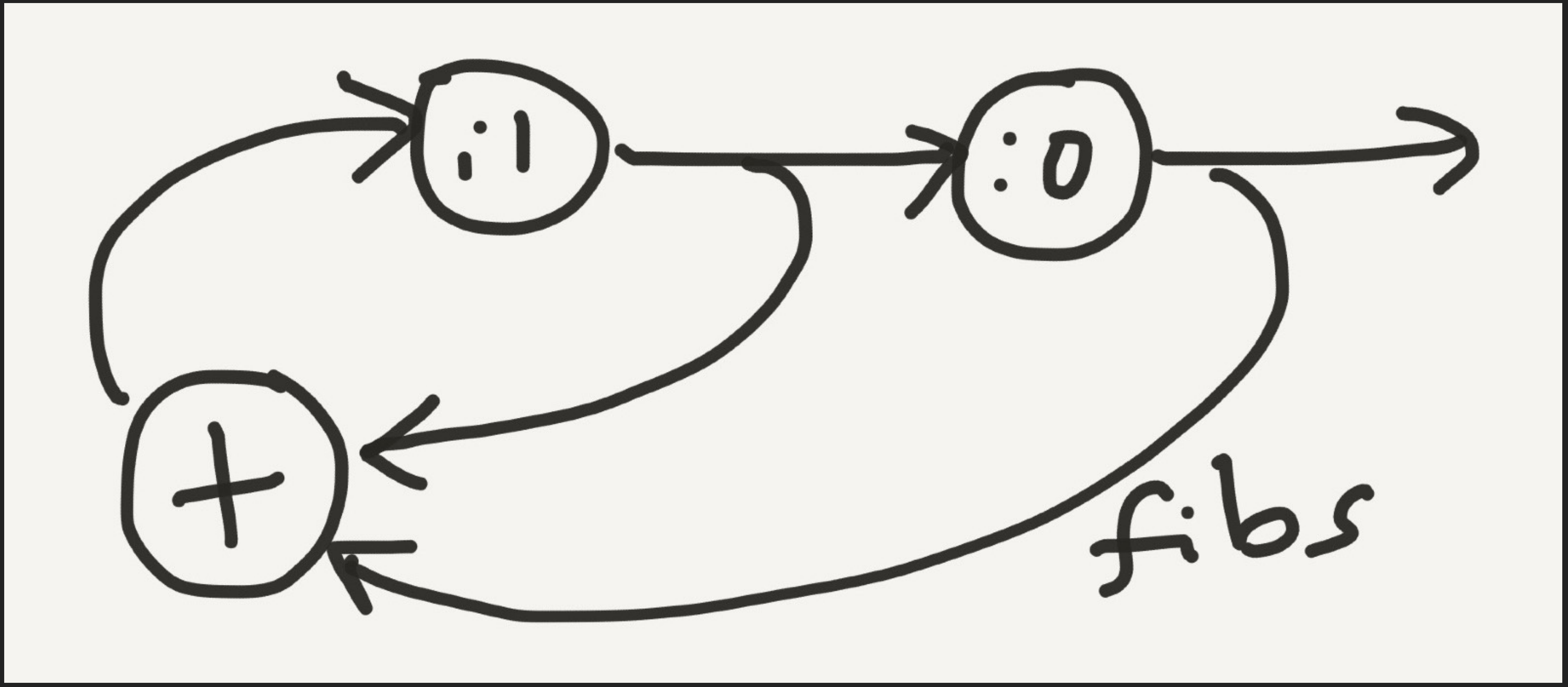
47, . . .

```
primes() -> sieve(ns(2)).

sieve(Ns) ->
  H = head(Ns),
  ?cons(H,sieve(cut(H,tail(Ns)))).

cut(N,Ns) ->
  H = head(Ns),
  case H rem N of
    0 ->          cut(N,tail(Ns));
    _ -> ?cons(H,cut(N,tail(Ns)))
  end.
```

```
fibs() ->
  ?cons(0,
    ?cons(1,
      addZip(fibs(),tail(fibs())))).

addZip(Xs,Ys) ->
   ?cons(head(Xs)+head(Ys), addZip(tail(Xs),tail(Ys))).
```

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, . . .

```
fibs() ->
  ?cons(0,
    ?cons(1,
      addZip(fibs(),tail(fibs())))).

addZip(Xs,Ys) ->
   ?cons(head(Xs)+head(Ys), addZip(tail(Xs),tail(Ys))).
```



demo

Take home

“infinite” streams

apparently circular

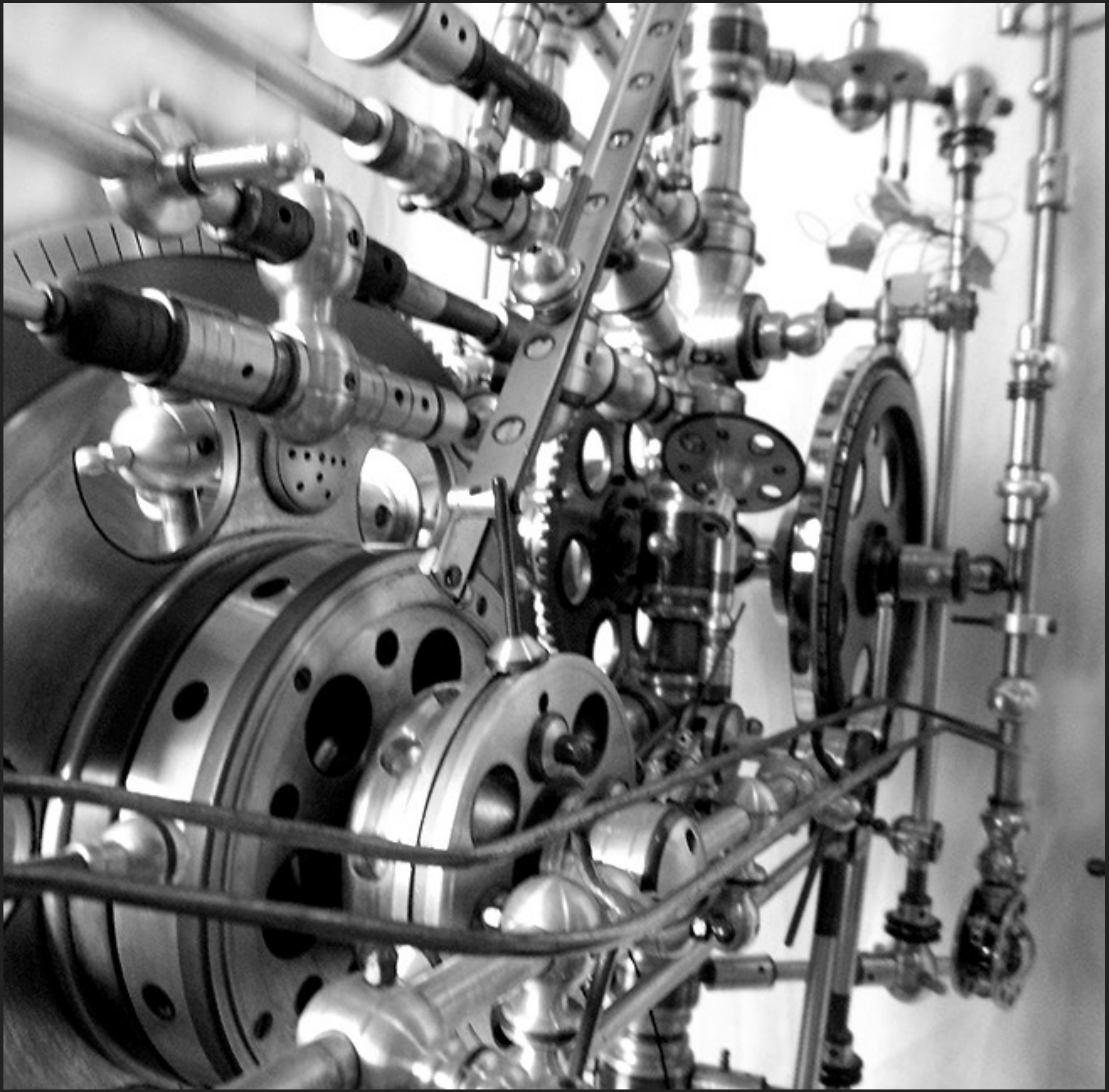
repeated re-computation

LAZY EVALUATION

ensure that each argument is evaluated at most once ensure that each argument is evaluated at most once

we must ensure that results are memoised in some way

but isn't
that a job
for the
compiler?



key idea

we explicitly manage how
results are stored once evaluated

use an ETS table to keep track
of evaluated results, or . . .

. . . model the store functionally,
thread it through the calculations

MEMOISATION

use ETS for general memoisation

use ETS for general memoisation

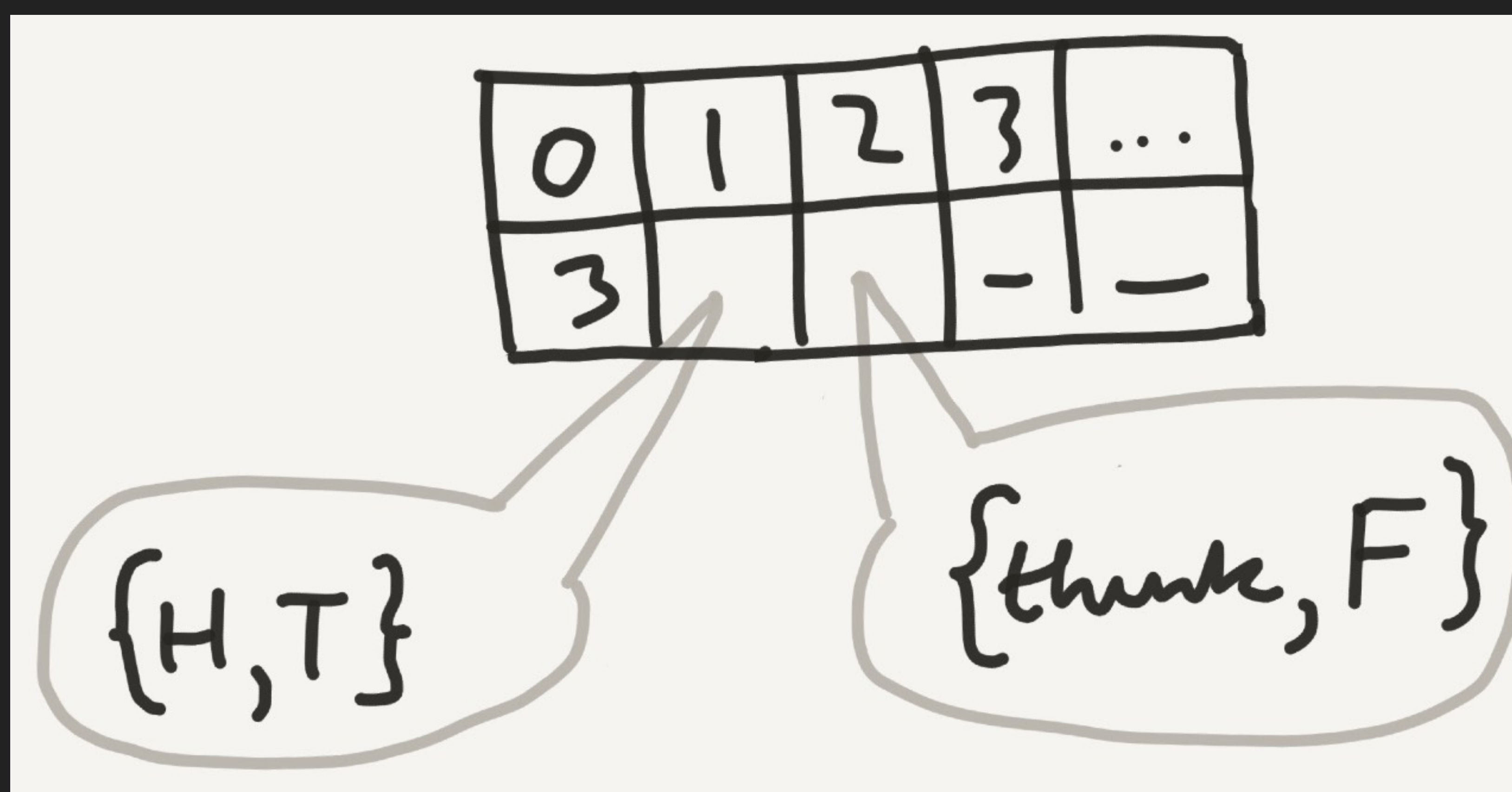
```
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

```
fib(0) -> 0;
fib(1) -> 1;
fib(N) -> fib(N-1) + fib(N-2).
```

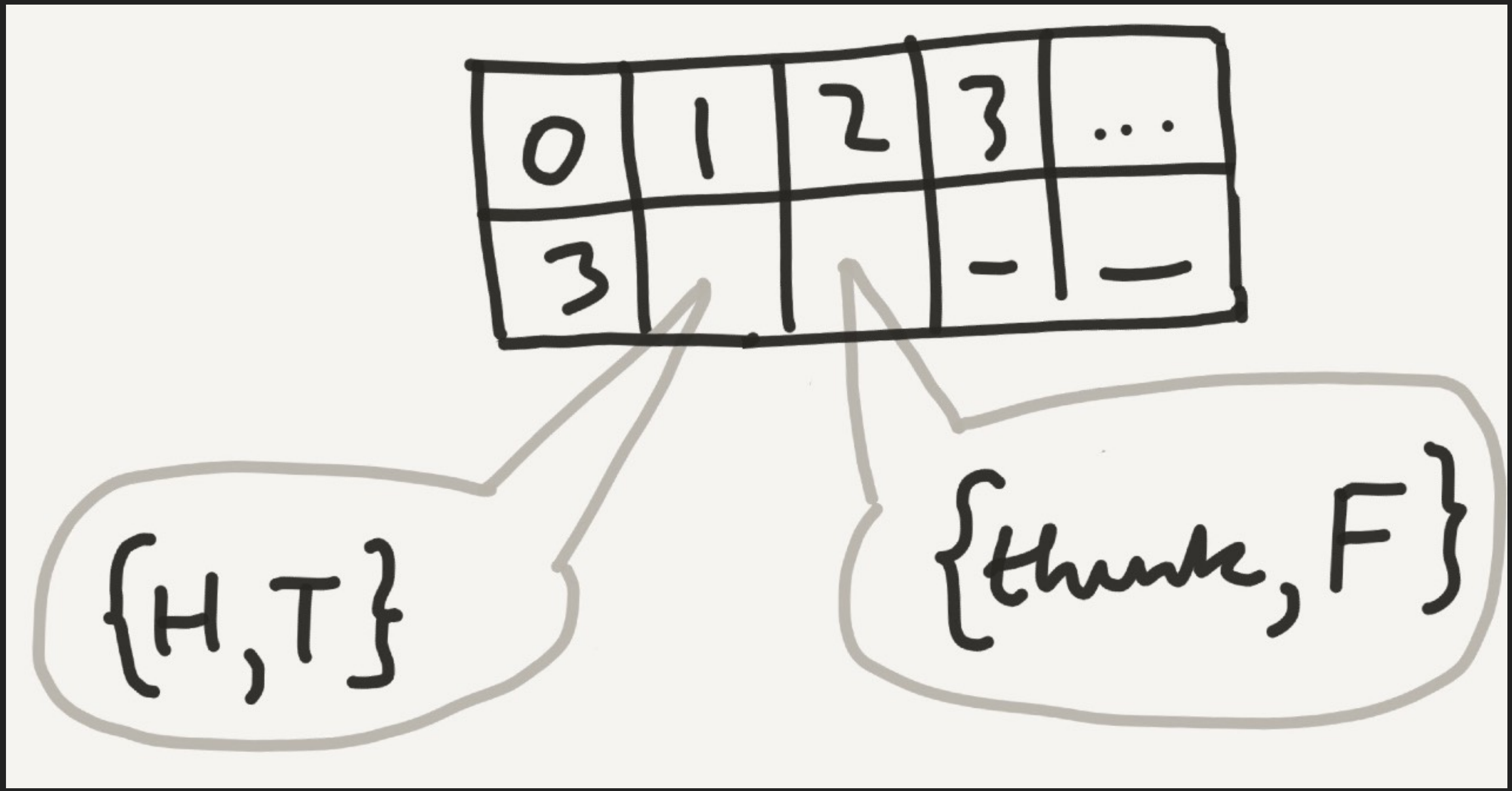
```
fibM(0) -> 0;
fibM(1) -> 1;
fibM(N) ->
    case ets:lookup(tab,N) of
        [] -> V = fibM(N-1) + fibM(N-2),
              ets:insert(tab,{N,V}),
              V;
        [{N,V}] -> V
    end.
```

USING ETS TABLES

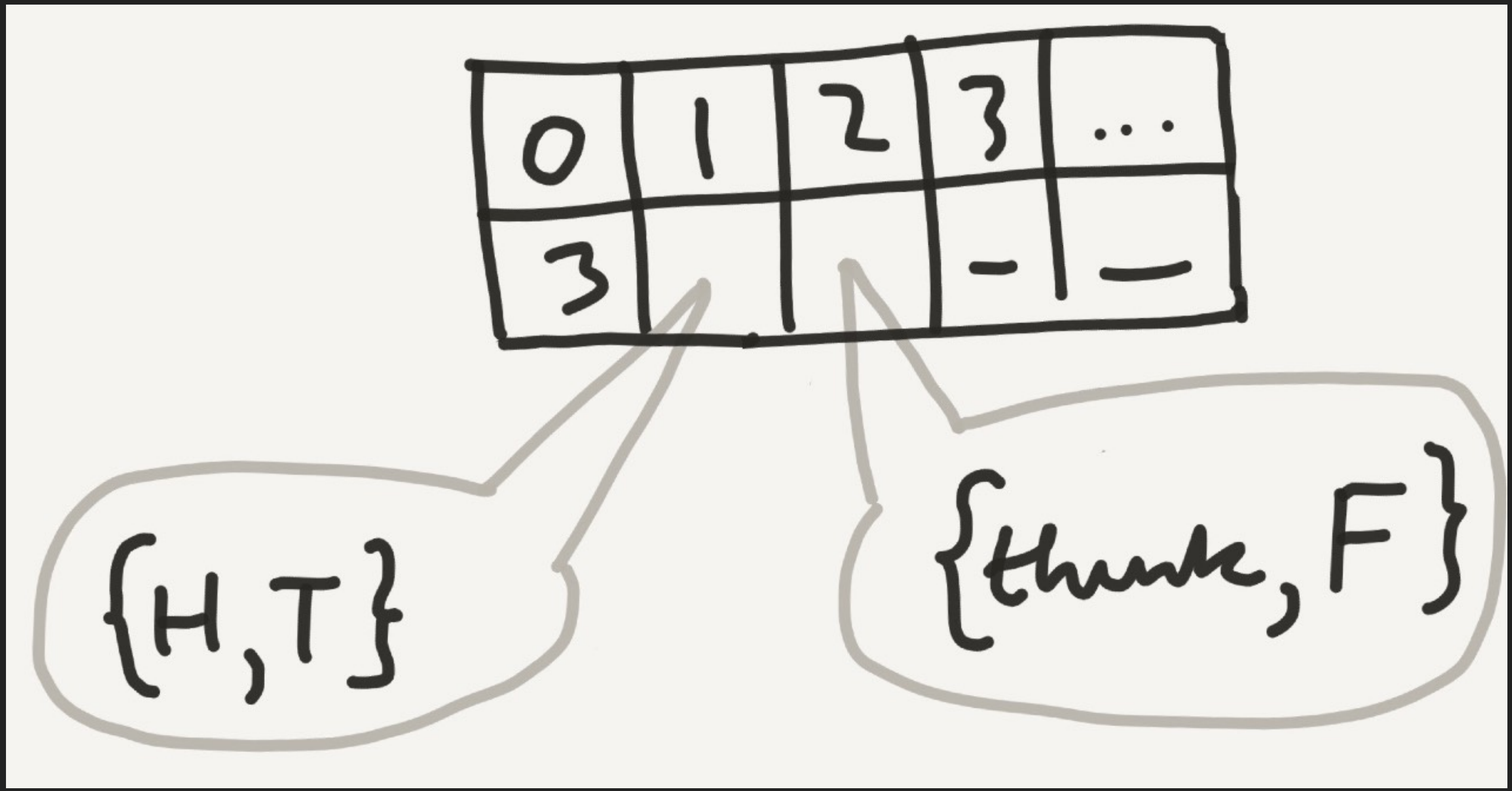
store either the head and tail, or a "thunk" to be evaluated



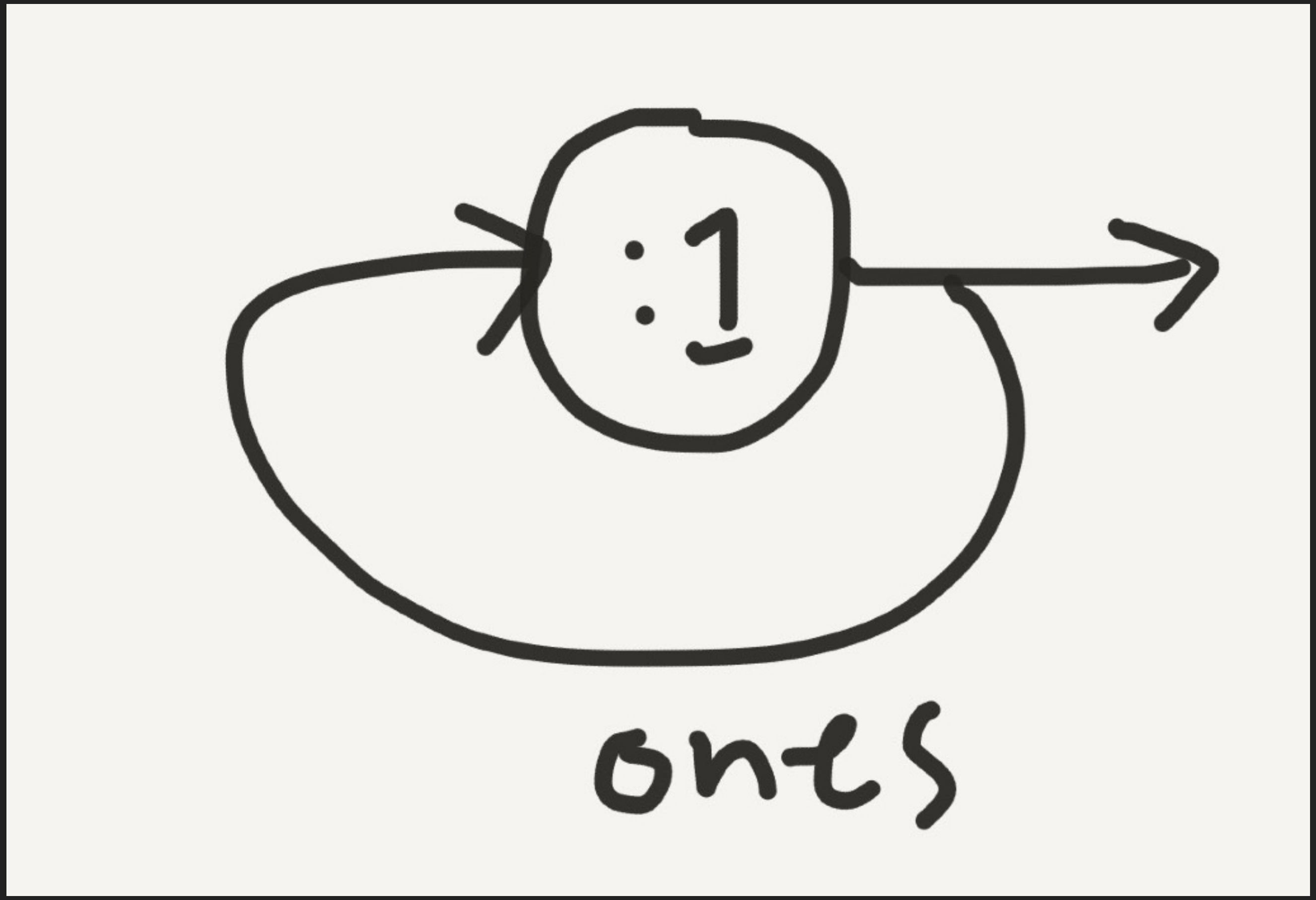
```
-define(cons(X,Xs), begin ets:insert(tab,{0,next_ref()+1}) ,
                          ets:insert(tab, {next_ref(),
                                           {thunk, fun () -> {X,Xs} end}}),
                          % io:format("done cons insert~n"),
                          {ref,next_ref()} end).
```



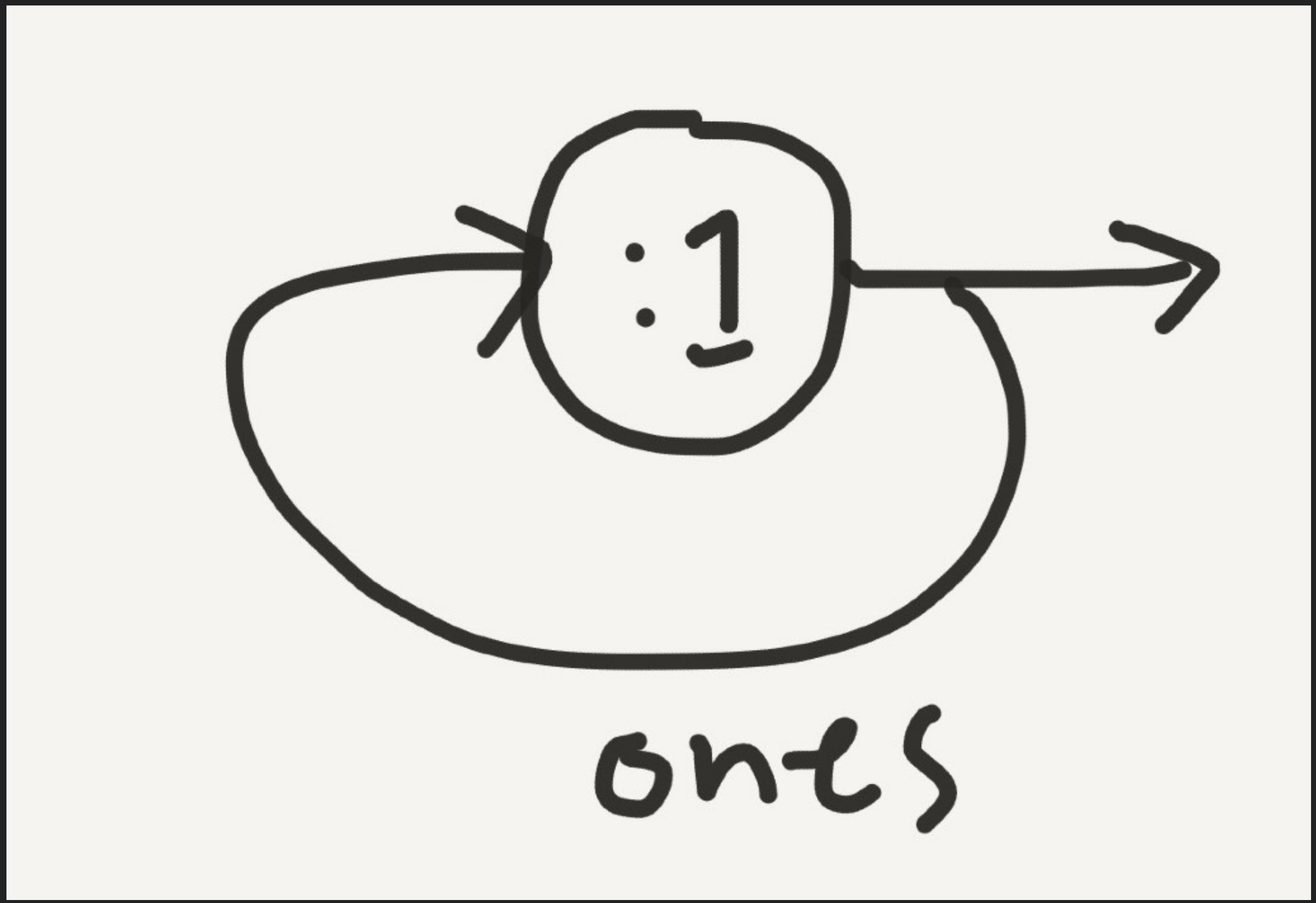
```
head({ref,Ref}) ->
 case ets:lookup(tab,Ref) of
    [{Ref,{thunk,F}}] -> Val = F(),
                         ets:insert(tab,{Ref,Val}),
                         {H,_} = Val,
                          H;
    [{Ref,{H,_}}]     ->  H
  end.
```



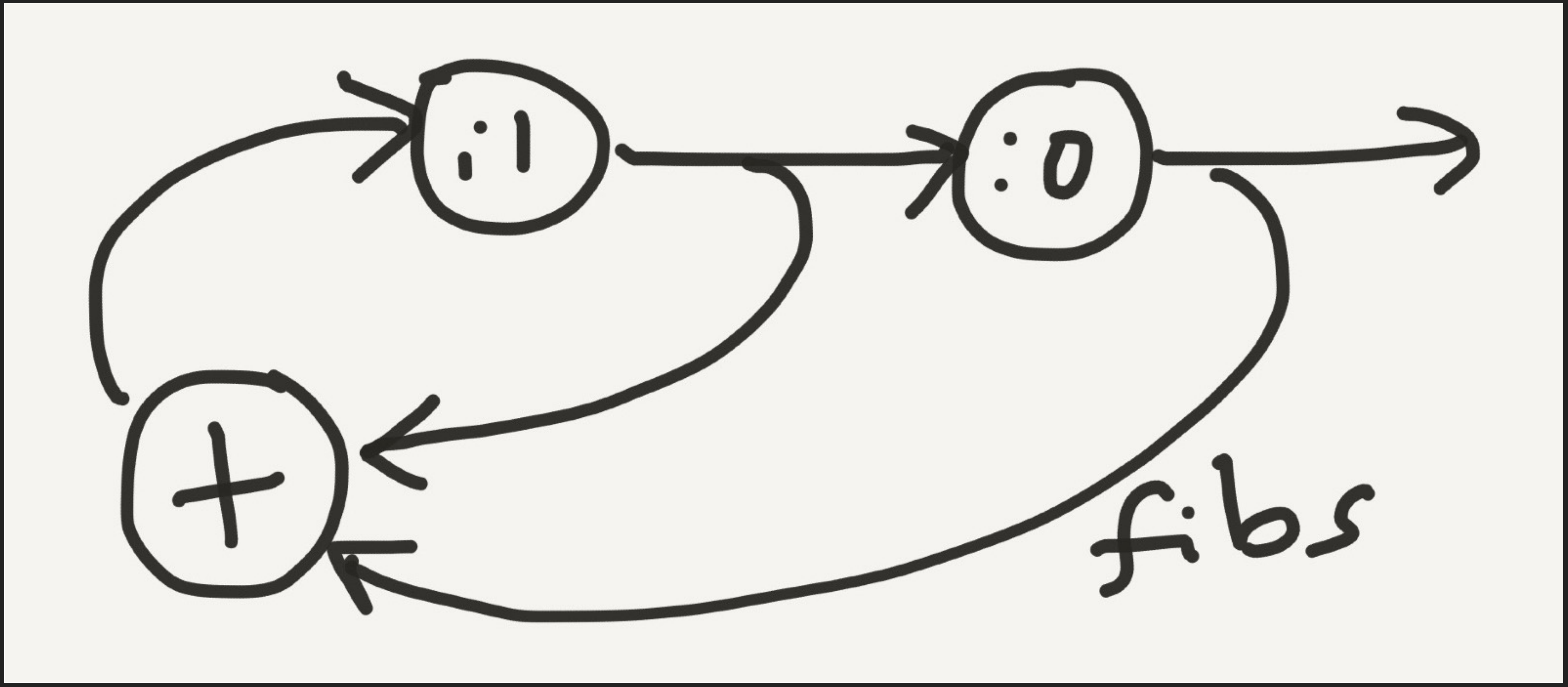
```
ones() ->
  ?cons(1,ones()).
```



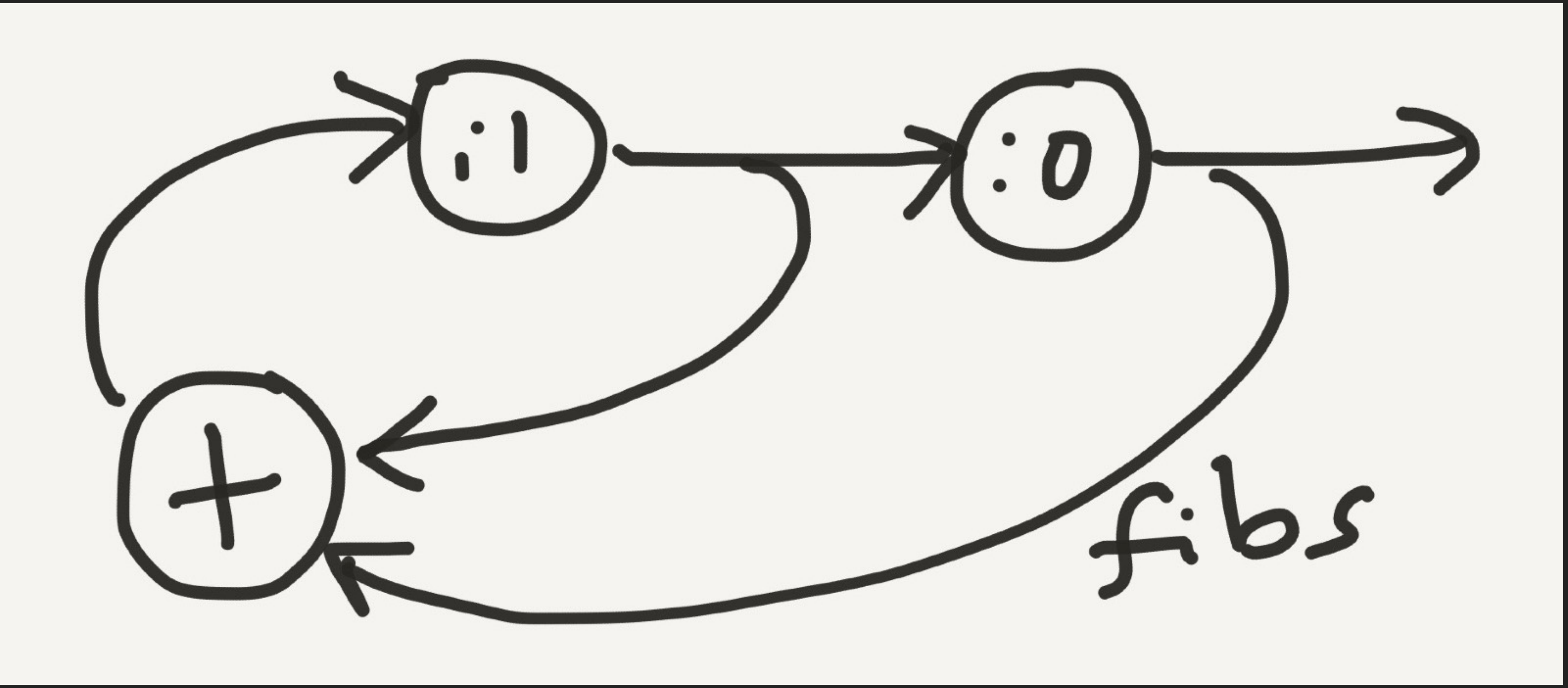
```
onesC() ->
  This = next_ref()+1,
  ?cons(1,{ref,This}).
```



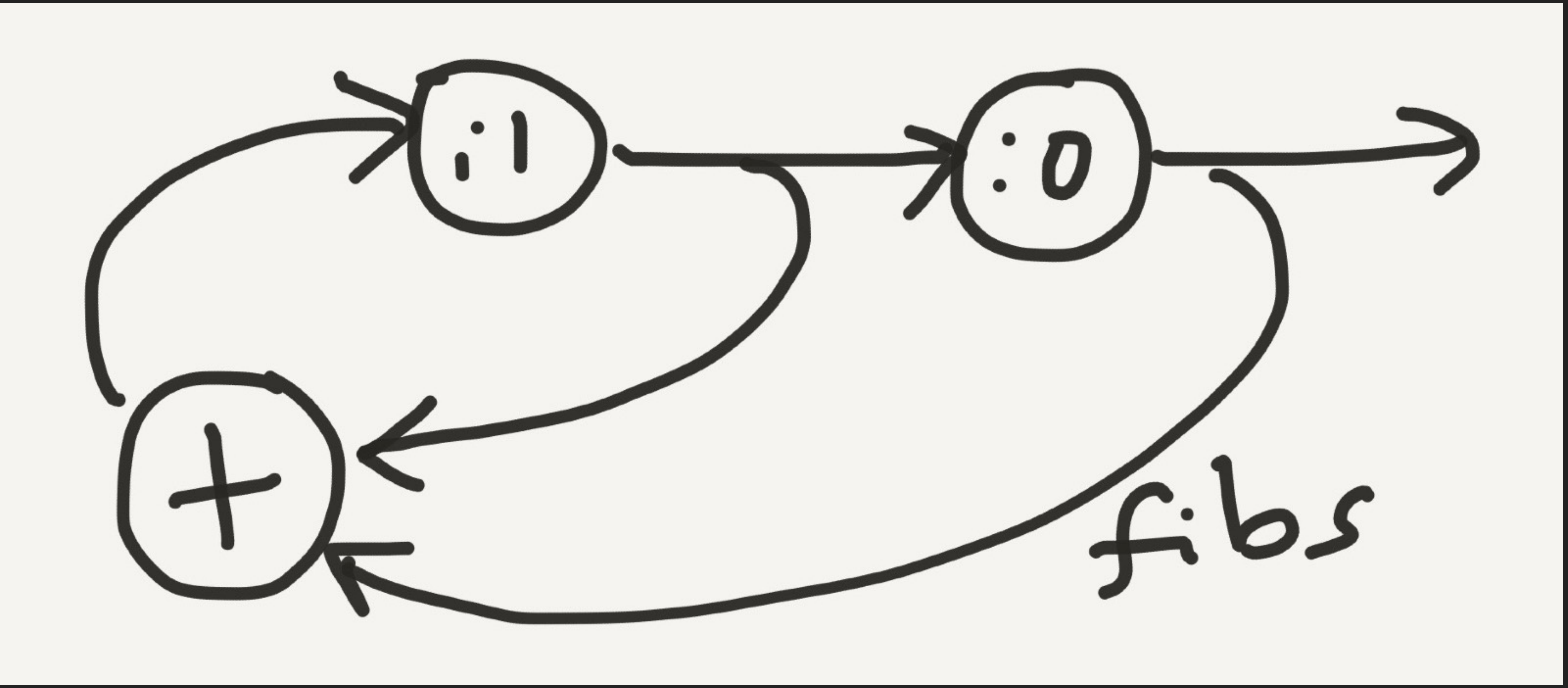
```
fibs() ->
  ?cons(0,
    ?cons(1,
      addZip(fibs(),tail(fibs())))).

addZip(Xs,Ys) ->
   ?cons(head(Xs)+head(Ys), addZip(tail(Xs),tail(Ys))).
```



```
fibs() ->
  ?cons(0,
    ?cons(1,
      addZip(fibs(),tail(fibs()))))
```



```
fibsCVar() ->
  This = next_ref()+1,
  ?cons(0,
    ?cons(1,
      addZip({ref,This},tail({ref,This})))).
```



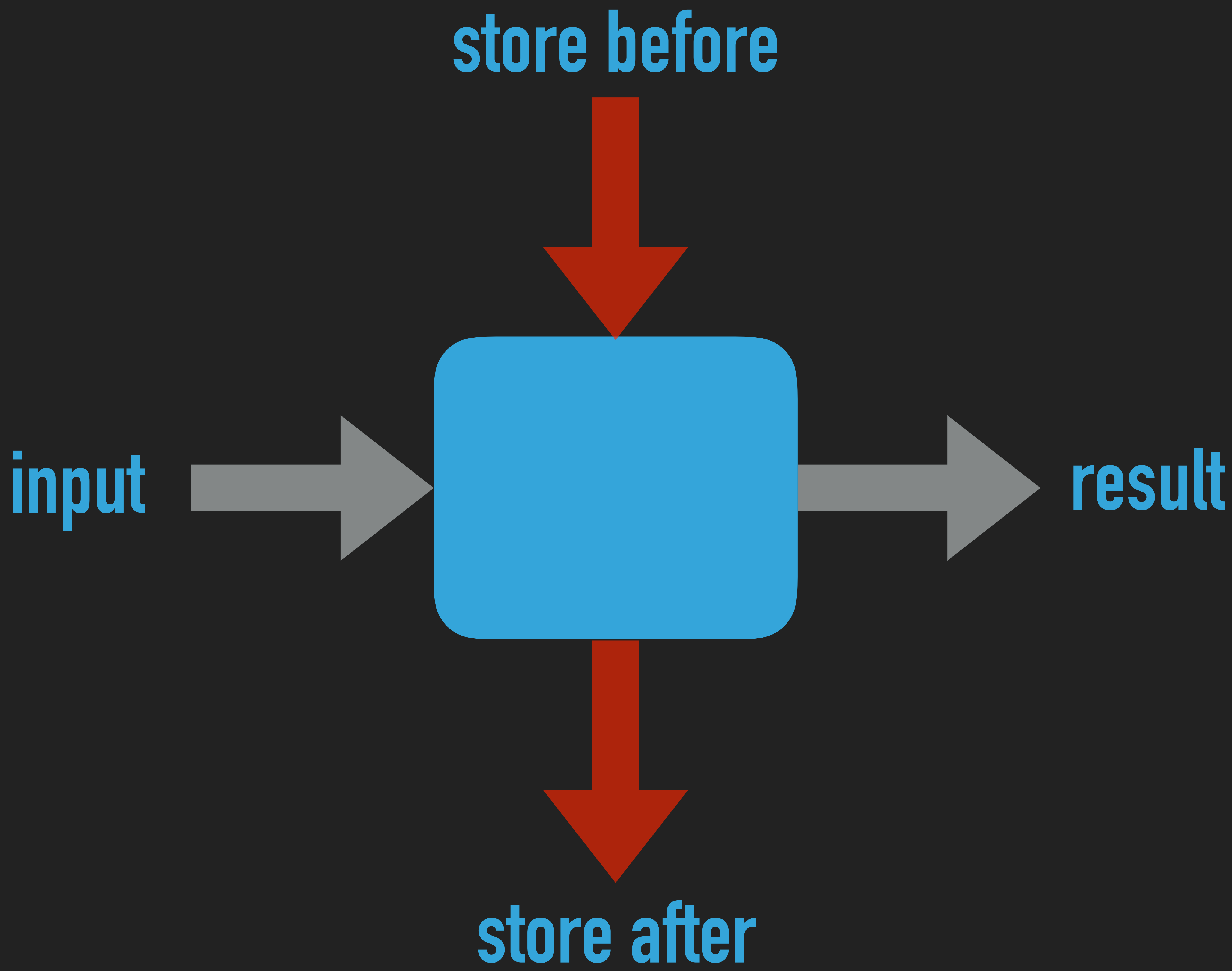
Explicitly managed refs

Simulates full lazy implementation

Uses impure features . . .

. . . but a smooth transition

AN EXPLICIT STORE


store before
input
result
store after

Printing out the first N values

```
ss(_Xs,0,_T) ->
  io:format("~n");

ss(Xs,N,Sto) ->
  io:format("~w, ",[head(Xs,Sto)]),
  {T,Sto1} = tail(Xs,Sto),
  ss(T,N-1,Sto1).
```

Node to {Head, {thunk, Tail}}

Thunk takes state as argument . . .

. . . so that the suspended computation can be evaluated in the context of the current state.

Construct a list

```
-define(cons(X,F,Sto),
        begin {{ref,next_ref(Sto)},
                Sto#{next_ref(Sto) => {X, {thunk, F}},
                     0 => next_ref(Sto)+1}}
        end).
```

```
tail({ref,Ref},Sto) ->
  case maps:get(Ref,Sto) of
    {ref,R}    ->
                 Hd = head({ref,R}, Sto),
                 {Tl,Sto1} = tail({ref,R},Sto),
                 Sto2 = Sto1#{Ref => {Hd,Tl}},
                 {Tl,Sto2};
    {Hd,{thunk,F}} ->
                 {Tl,StoC} = F(Sto),
                 Sto1 = StoC#{Ref => {Hd,Tl}},
                 {Tl,Sto1};
    {_,T} ->
                 {T,Sto}
  end.
```

Fibonacci numbers

```
fibsC(Sto) ->
  This = next_ref(Sto),
  ?cons(0, fun(T) ->
             ?cons(1, fun(S) ->
                            begin
                            {Tl,S1} = tail({ref,This},S),
                            addZip({ref,This},Tl,S1)
                            end
                      end,
                  T)
           end,
        Sto).
```

TO CONCLUDE

functions are flexible and powerful modelling tool

strategies

simulations

suspensions

pure modelling of effects
is not straightforward

monads, monad transformers,
effects, ... provide some
useful patterns

reify?

can model DSLs of strategies, parsers, and write interpreters for these DSLs into the functions we’ve seen here

data and types

all the data we used here was
well understood 30 years ago

it is just that the types have changed

https://github.com/simonjohnthompson/streams fun