Lazy interactions – back to the future

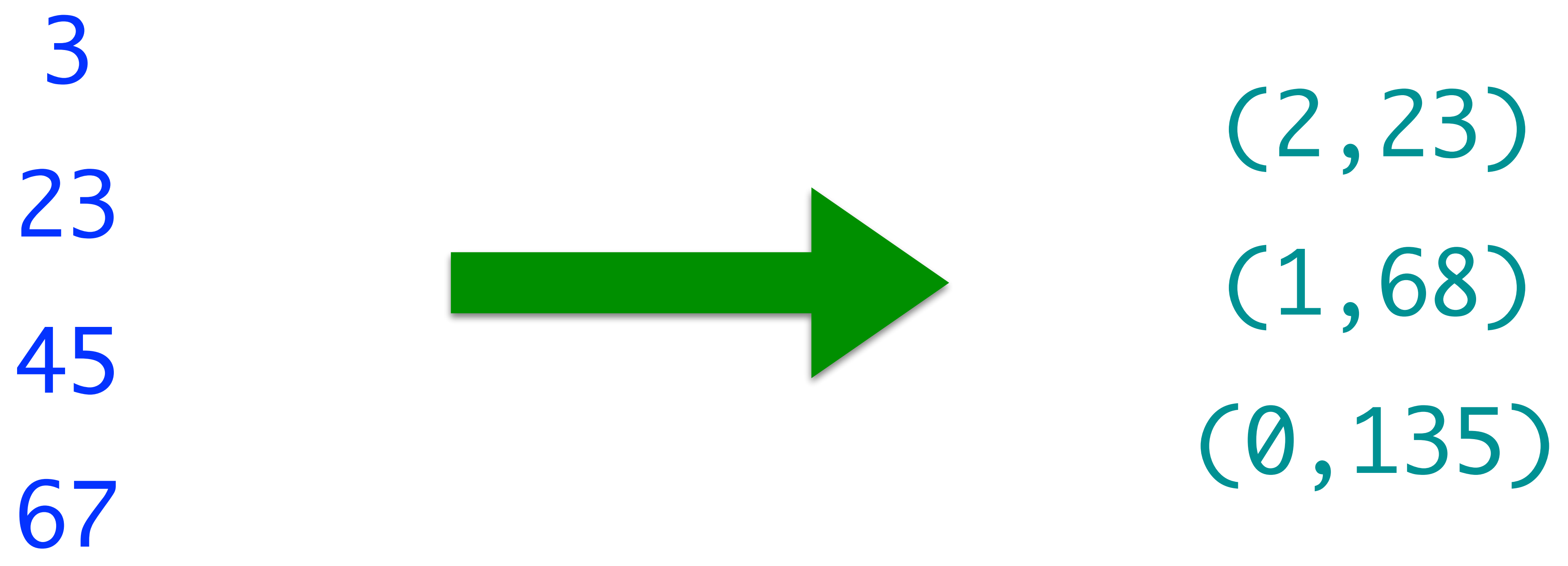
Simon Thompson, University of Kent

```
System.IO.interact :: (String -> String) -> IO ()
System.IO.interact :: (String -> String) -> IO ()


interact f = do s <- getContents
                putStr (f s)
```

```
f :: (Input -> Output)
```

The output of the program is a function of its input.

```
f :: (Input -> Output)
```



3

23

45

67

(2,23)

(1,68)

(0,135)

```haskell
f :: (Input -> Output)
```

```
    3
   23
(2,23)
   45
(1,68)
   67
(0,135)
```

```
f :: (Input -> Output)
```

```
    3
   23
 (2,23)
   45
 (1,68)
   67
(0,135)
```

Interaction
=
input / output
interleaving

```
f :: (Input -> Output)
```

```
3
23
(2,23)
45
(1,68)
67
(0,135)
```

Interaction
=
input / output
interleaving

Interleaving
determined
by lazy evaluation

The essence of laziness

```
f ⊥
  = "type now" ++ ⊥

f ("echo" ++ ⊥)
  = … ++ "ohce" ++ ⊥
```

The essence of laziness

```
f ⊥
  = "type now" ++ ⊥

f ("echo" ++ ⊥)
  = … ++ "ohce" ++ ⊥
```

Lazy interactions are determined by the behaviour of the function on **partial data**.

Demo

“Seat of the pants?”

```
necho ys
  = "Prompt: " ++ [head ys] ++ "\n" ++ necho (tail ys)
```

vs

```
necho (x:xs)
  = "Prompt: " ++ [x] ++ "\n" ++ necho xs
```

“Seat of the pants?”

```
necho ~(x:xs)
  = "Prompt: " ++ [x] ++ "\n" ++ necho xs
```

vs

```
necho (x:xs)
  = "Prompt: " ++ [x] ++ "\n" ++ necho xs
```

“Seat of the pants?”

```
necho ~(x:xs)
  = "Prompt: "
```

vs

```
necho (x:xs)
  = "Prompt: "
```

Let’s build a model of interactions and how to combine them together …

Back to the future?

YEAR OF PROGRAMMING

The 1987 University of Texas Year of Programming was established early in 1986, in response to a proposal by Profs. ~~J. C.~~ Browne and J. Misra, with the following goals:

- 1) to advance the art and science of programming by bringing leading scientists together for discussions and collaboration;

- 2) to disseminate among leading practitioners the best of what has been learned about the theory and practice of programming; and

YEAR OF PROGRAMMING

The 1987 University of Texas Year of Programming was established early in 1986, in response to a proposal by Profs. ~~J. C.~~ Browne and J. Misra, with the following goals:

- 1) to advance the art and science of programming by bringing leading scientists together for discussions and collaboration;
- 2) to disseminate among leading practitioners the best of what has been learned about the theory and practice of programming; and

The tutorial, which provided an introduction to lazy functional programming, consisted of lectures interspersed with programming sessions (conducted with pencil and paper) attended by the lecturers and several teaching assistants. Major topics included data types, polymorphism, recursion and induction, lists, domain theory, program synthesis, and several case studies.

YEAR OF PROGRAMMING

The 1987 University of Texas Year of Programming was established early in 1986, in response to a proposal by Profs. ~~J. C.~~ Browne and J. Misra, with the following goals:

• 1) to advance the art and science of programming by bringing leading scientists together for discussions and collaboration;

• 2) to disseminate among leading practitioners the best of what has been learned about the theory and practice of programming; and

The tutorial, which provided an introduction to lazy functional programming, consisted of lectures interspersed with programming sessions (conducted with pencil and paper) attended by the lecturers and several teaching assistants. Major topics included data types, polymorphism, recursion and induction, lists, domain theory, program synthesis, and several case studies.

This institute elicited particular enthusiasm among a group of UT graduate students, who circulated among themselves, and subsequently presented to the UT Department of Computer Sciences, a petition calling on the department "to make Functional Programming a more visible priority in the department... [through] recruitment of faculty engaged in research in the field [and] more formal contacts with private research and other departments...".

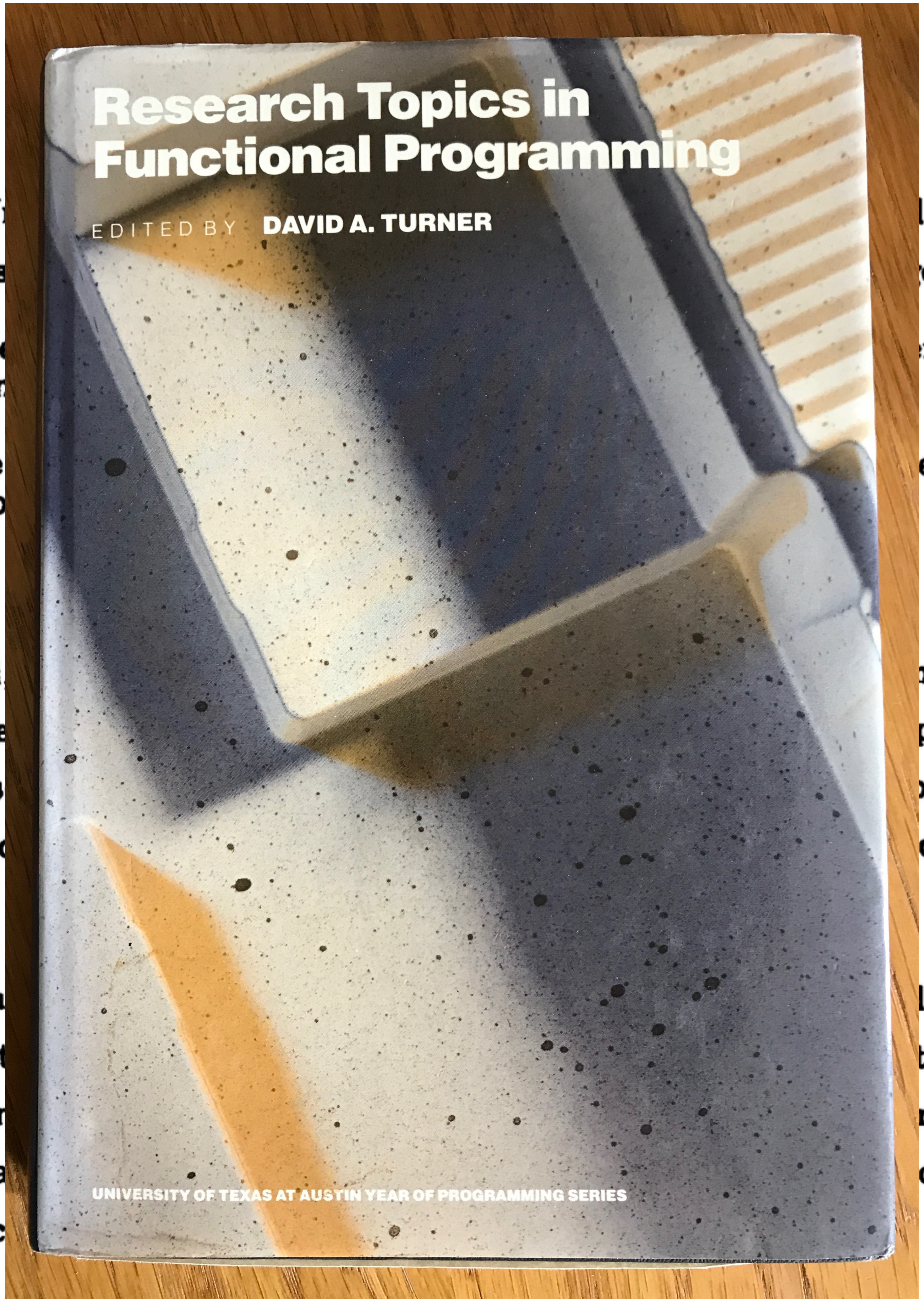
The 1987 U
early in 1986, in
response to a propos
oals:

1) to advance the
discussions an
ntists together for

2) to disseminate
and practice o
ed about the theory

The tutorial
mming, consisted of
lectures interspers
paper) attended by
the lecturers and s
pes, polymorphism,
recursion and induc
case studies.

This institu
luate students, who
circulated among t
tment of Computer
Sciences, a petition
ning a more visible
priority in the depa
ch in the field [and]
more formal contac


Research Topics in Functional Programming
EDITED BY DAVID A. TURNER
UNIVERSITY OF TEXAS AT AUSTIN YEAR OF PROGRAMMING SERIES

Back to the future?

Does it still make sense now?

Back to the future?

Does it still make sense now?

The power of retrospection …

Back to the future?

Does it still make sense now?

The power of retrospection …

… how we can bring it up to date?

Back to the future?

Does it still make sense now?

The power of retrospection …

… how we can bring it up to date?

… and any missed opportunities?

Back to the future?

Does it still make sense now?

The power of retrospection …

… how we can bring it up to date?

… and any missed opportunities?

Transliterating from Miranda to Haskell

Back to the future?

Does it still make sense now?

The power of retrospection …

… how we can bring it up to date?

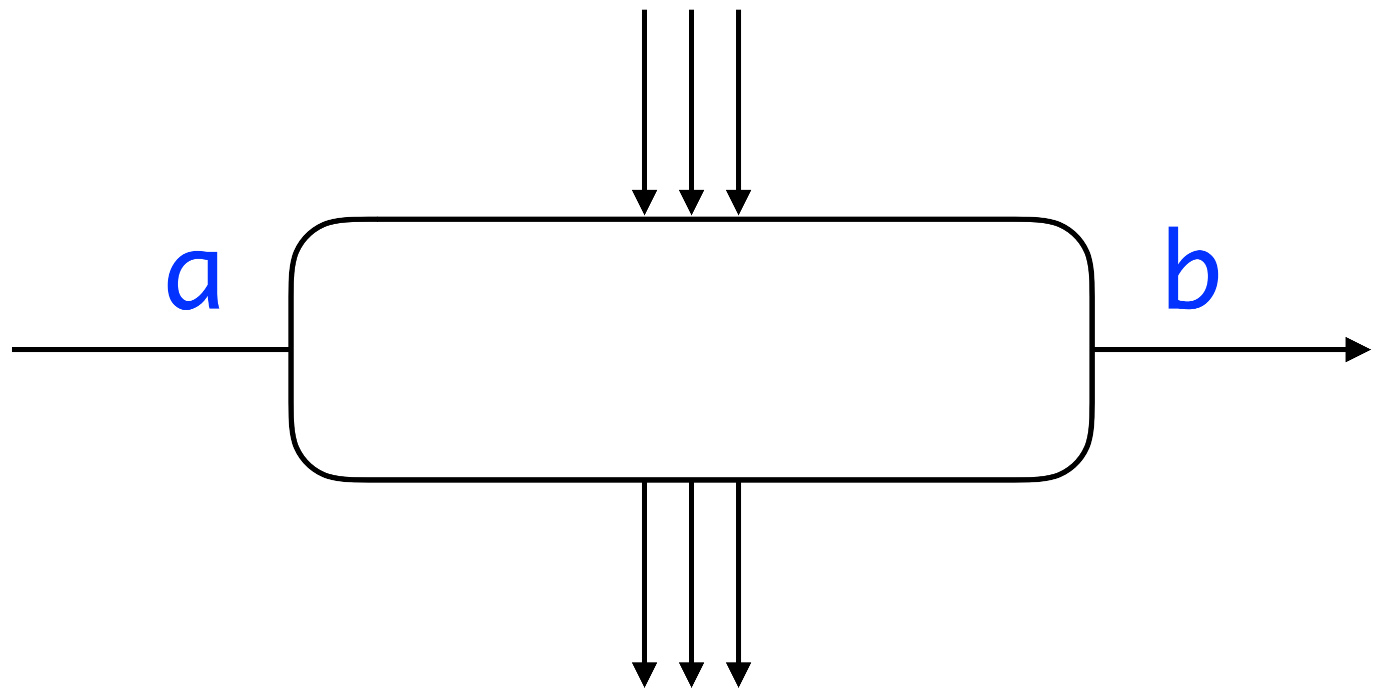
… and any missed opportunities?

Transliterating from Miranda to Haskell

Building a formal model of interactions, with some proofs …

```
(Input,a) -> (Input,b,Output)
```

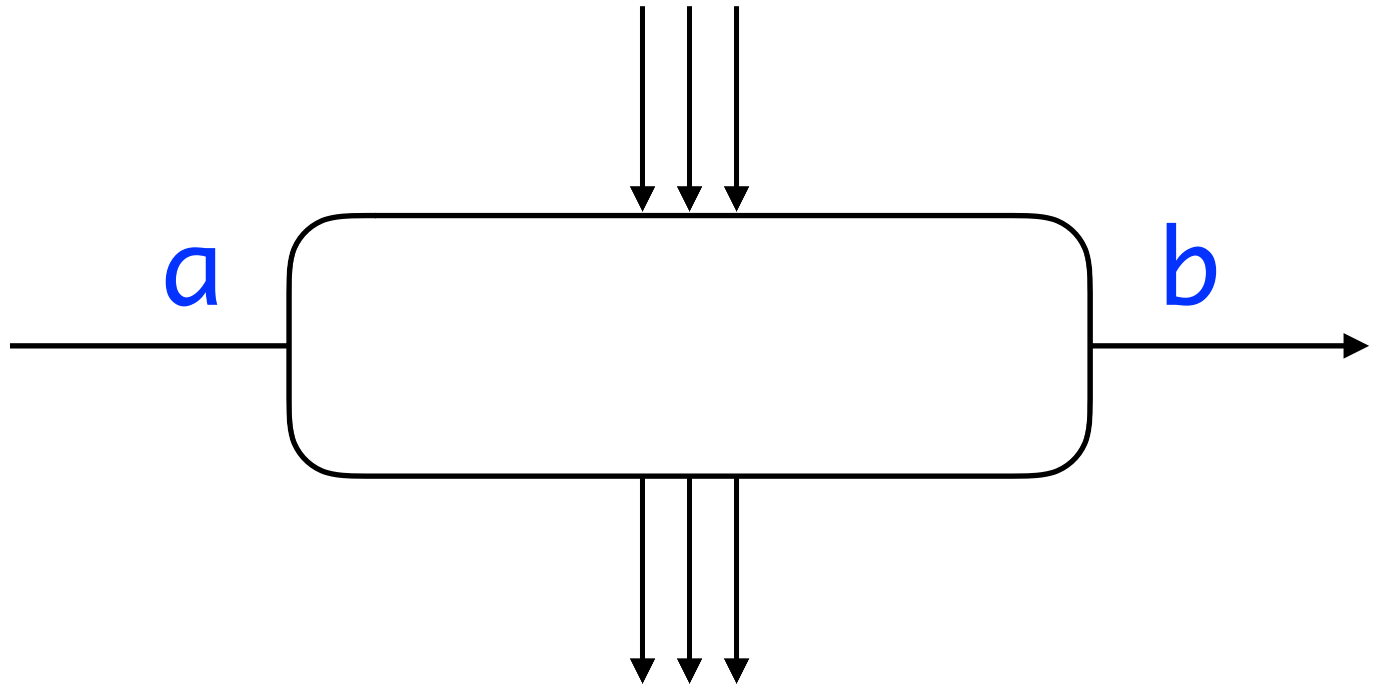
(Input,a) -> (Input,b,Output)



```
(Input,a) -> (Input,b,Output)
```

Functions with IO side effects

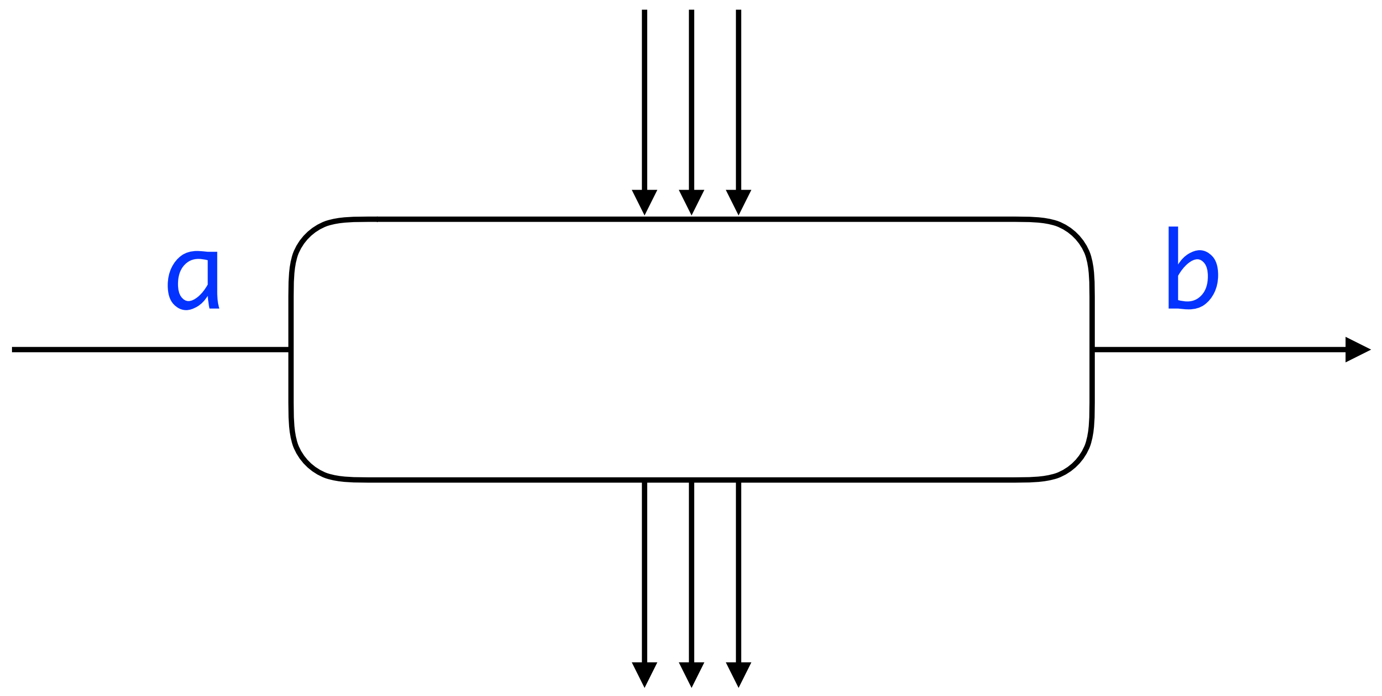
Build by composition



`(Input,a) -> (Input,b,Output)`

Functions with IO side effects

Build by composition



Interactions with states

State changes type between steps …

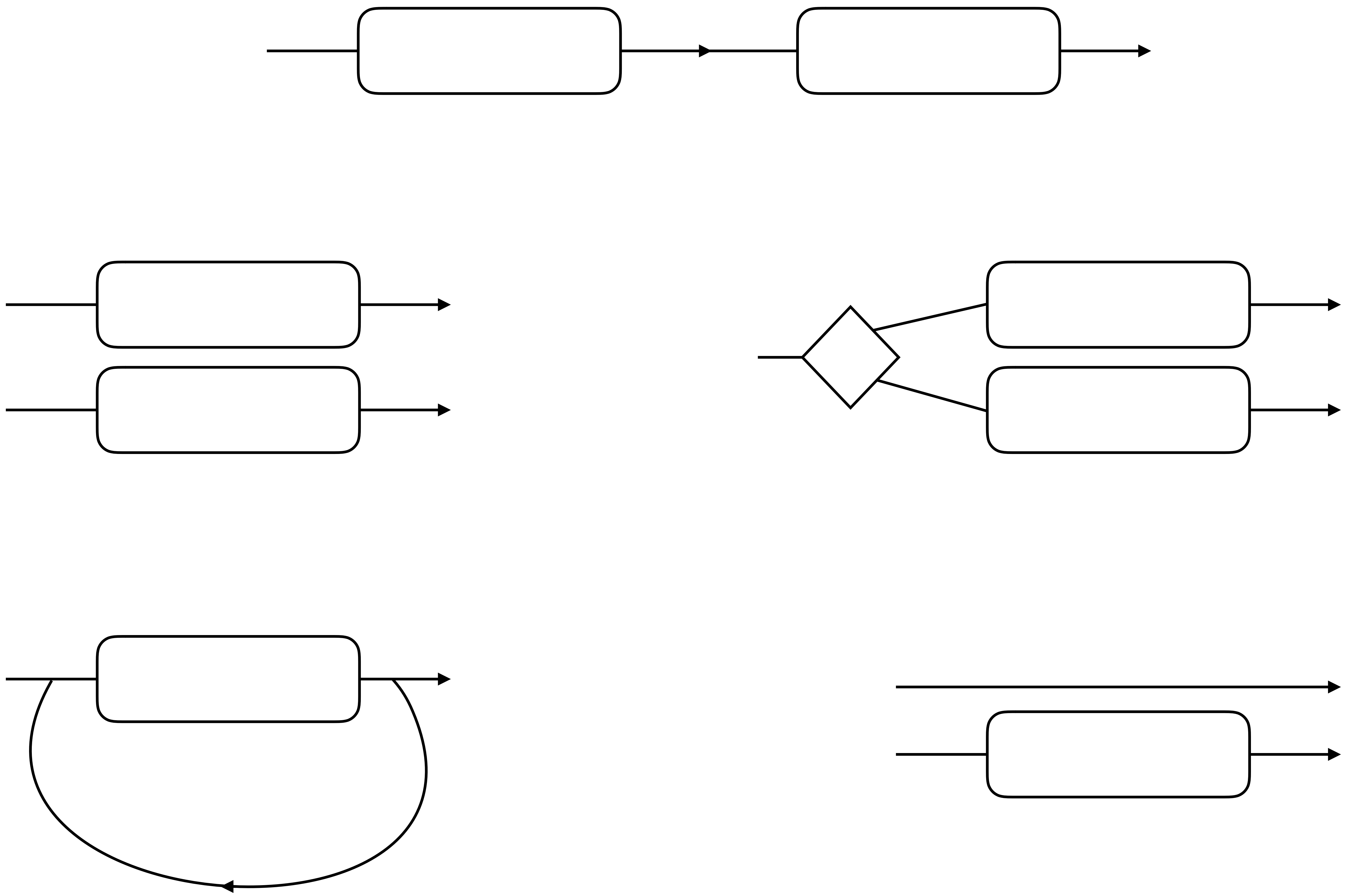
… can add, remove, and modify what’s there.

Basic types …

```
type Interact a b
  = (Input,a) -> (Input,b,Output)

type Condition a
  = (Input,a) -> Bool

type Input  = [String]
type Output = [String]
```

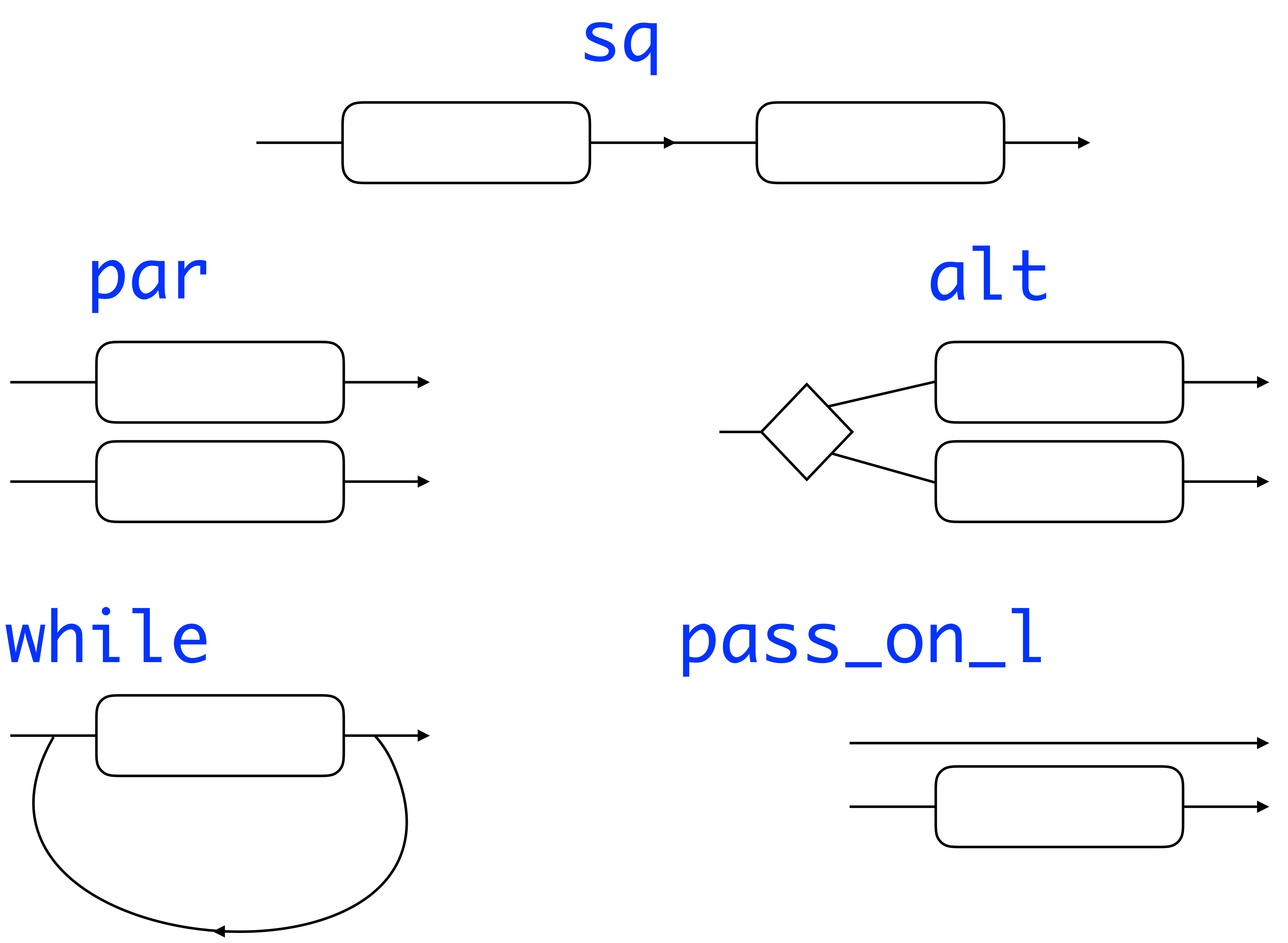
How do we put these together?

We assume that all diagrams are well-typed



How do we put these together?

We assume that all diagrams are well-typed



sq

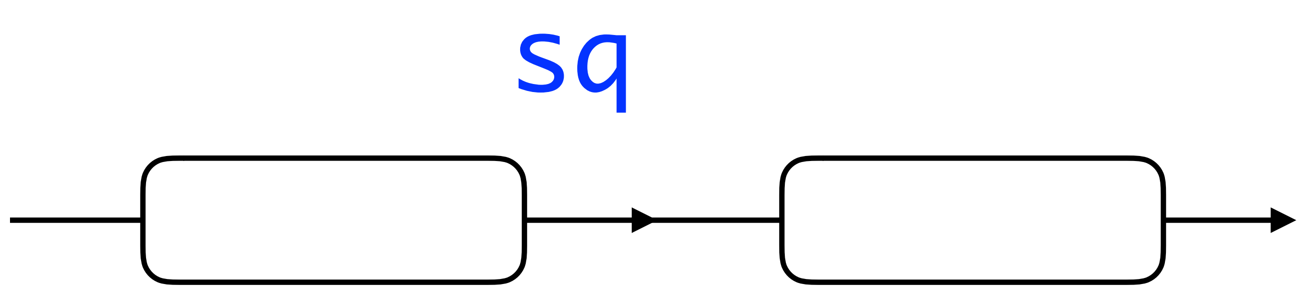
par

alt

while

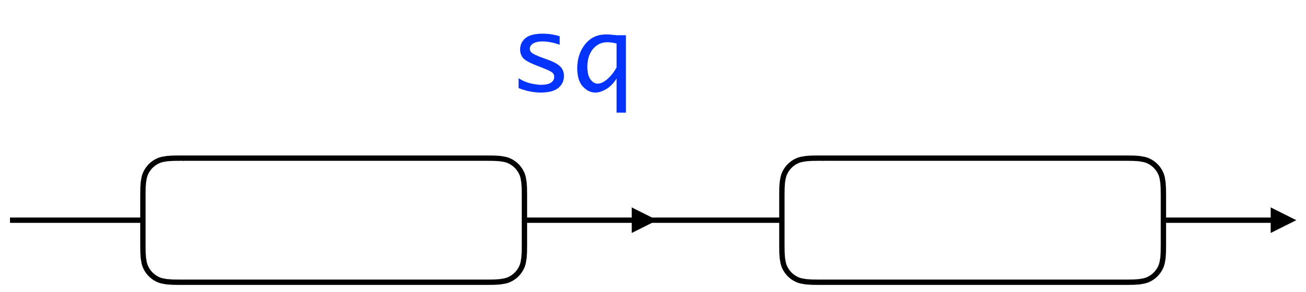
pass_on_l

Sequencing … key combinator



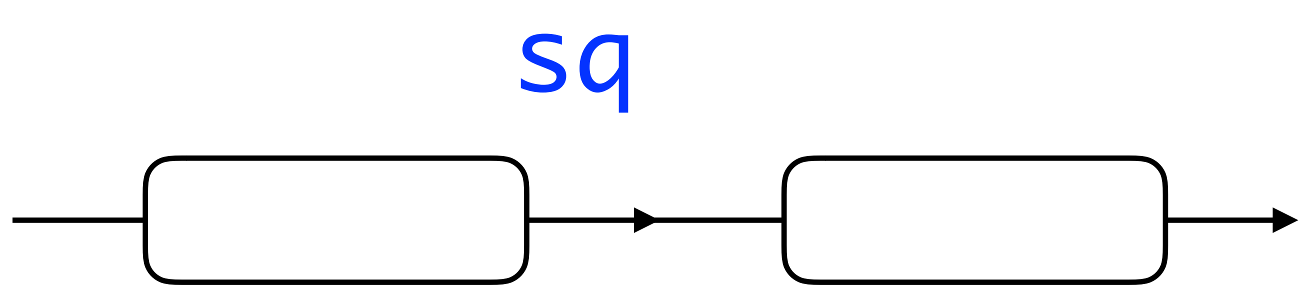
```
sq :: Interact a b -> Interact b c -> Interact a c

sq inter1 inter2 x
  = make_Output out1 (inter2 (rest,st))
    where (rest,st,out1) = inter1 x
```

Sequencing … key combinator



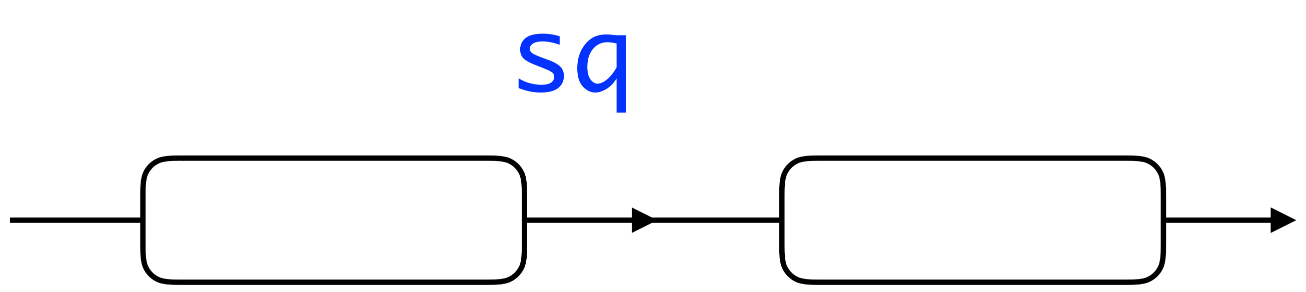
```
sq :: Interact a b -> Interact b c -> Interact a c

sq inter1 inter2 x
  = make_Output out1 (inter2 (rest,st))
    where (rest,st,out1) = inter1 x


make_Output :: Output -> (Input,a,Output) -> (Input,a,Output)

make_Output piece (input,st,out) = (input,st,piece++out)
```

Sequencing … key combinator



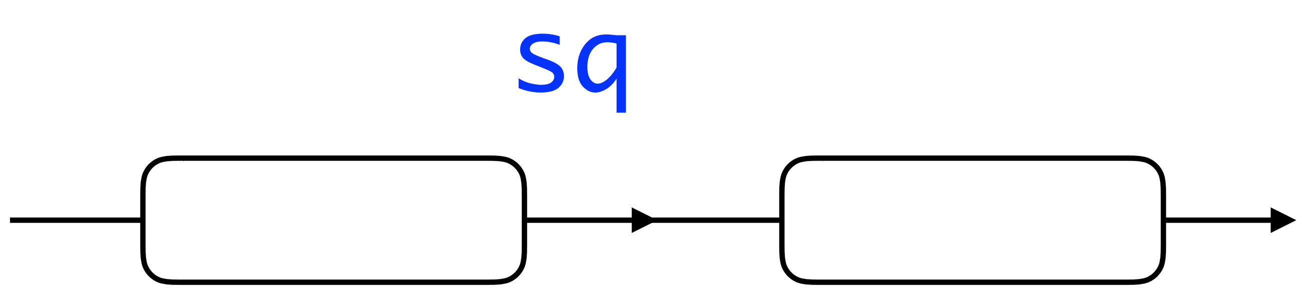
```
sq :: Interact a b -> Interact b c -> Interact a c

sq inter1 inter2 x
  = make_Output out1 (inter2 (rest,st))
    where ~(rest,st,out1) = inter1 x


make_Output :: Output -> (Input,a,Output) -> (Input,a,Output)

make_Output piece (input,st,out) = (input,st,piece++out)
```

Sequencing … key combinator



```
sq :: Interact a b -> Interact b c -> Interact a c

sq inter1 inter2 x
  = make_Output out1 (inter2 (rest,st))
    where ~(rest,st,out1) = inter1 x


make_Output :: Output -> (Input,a,Output) -> (Input,a,Output)

make_Output piece ~(input,st,out) = (input,st,piece++out)
```

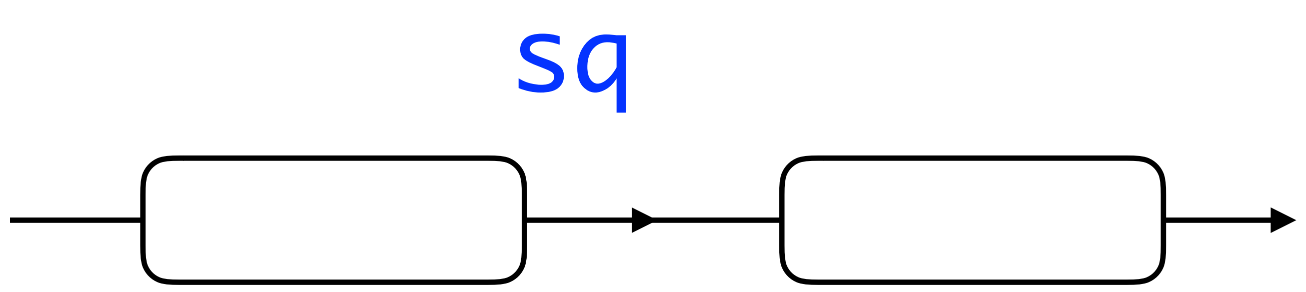
Sequencing … key combinator



```
sq :: Interact a b -> Interact b c -> Interact a c

sq inter1 inter2 x
  = make_Output out1 (inter2 (rest,st))
    where ~(rest,st,out1) = inter1 x


make_Output :: Output -> (Input,a,Output) -> (Input,a,Output)

make_Output piece ~(input,st,out) = (input,st,piece++out)
```

Need an irrefutable pattern in a function definition …

Sequencing … key combinator



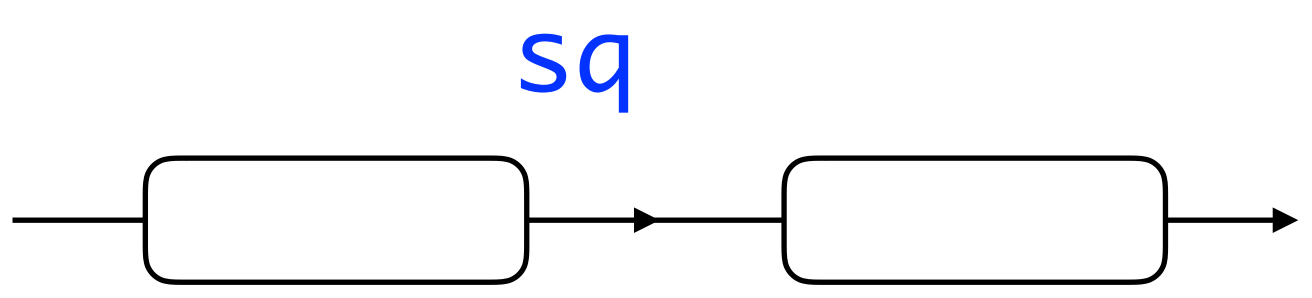
```
sq :: Interact a b -> Interact b c -> Interact a c

sq inter1 inter2 x
  = make_Output out1 (inter2 (rest,st))
    where ~(rest,st,out1) = inter1 x
```

```
make_Output :: Output -> (Input,a,Output) -> (Input,a,Output)

make_Output piece ~(input,st,out) = (input,st,piece++out)
```

… but patterns in where clauses are irrefutable by default.

Need an irrefutable pattern in a function definition …

Sequencing … key combinator



```
sq :: Interact a b -> Interact b c -> Interact a c

sq inter1 inter2 x
  = make_Output out1 (inter2 (rest,st))
    where (rest,st,out1) = inter1 x
```

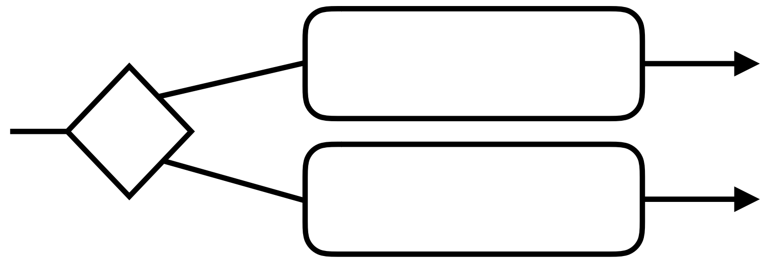
… but patterns in where clauses are irrefutable by default.

```
make_Output :: Output -> (Input,a,Output) -> (Input,a,Output)

make_Output piece ~(input,st,out) = (input,st,piece++out)
```

Need an irrefutable pattern in a function definition …

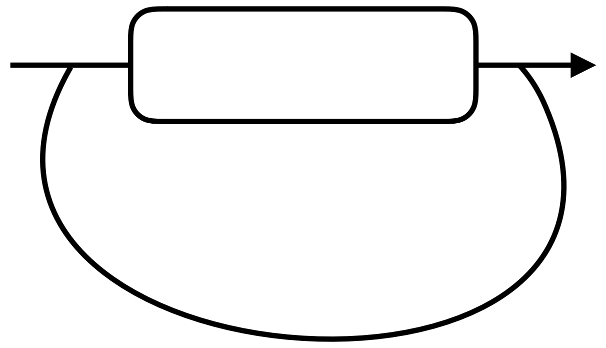
Alternation and repetition

alt



```
alt :: Cond a -> Interact a b -> Interact a b -> Interact a b

alt cond inter1 inter2 x
    | cond x      = inter1 x
    | otherwise   = inter2 x
```

while



```
while ::  Cond a -> Interact a a -> Interact a a

while cond inter
  = whi
    where
    whi = alt cond (inter `sq` whi) null
```

“Passing parameters”

```
pass_param :: Interact a b ->
              (b -> Interact () d) ->
              Interact a d
```

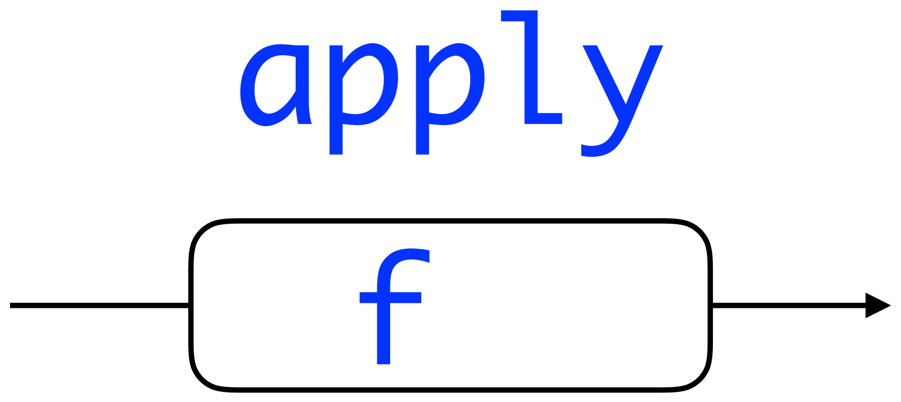
```
pass_param int f (input,st)
  = (rest,final,out1++out)
    where
    (inter1,st1,out1) = int (input,st)
    (rest,final,out)  = (f st1) (inter1,())
```

“Passing parameters”

```
pass_param :: Interact a b ->
              (b -> Interact b d) ->
              Interact a d
```

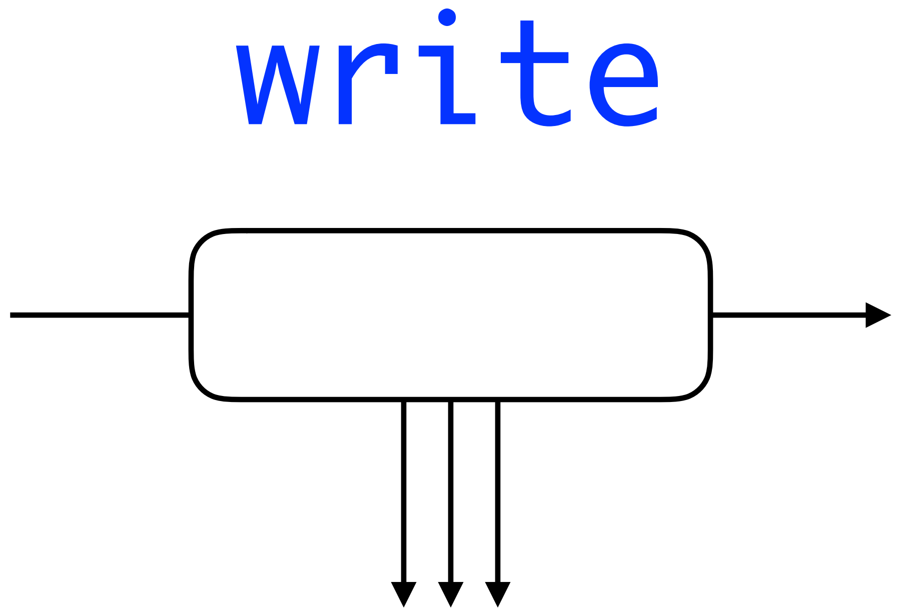
```
pass_param int f (input,st)
  = (rest,final,out1++out)
    where
    (inter1,st1,out1) = int (input,st)
    (rest,final,out)  = (f st1) (inter1,st1)
```

And some primitives …

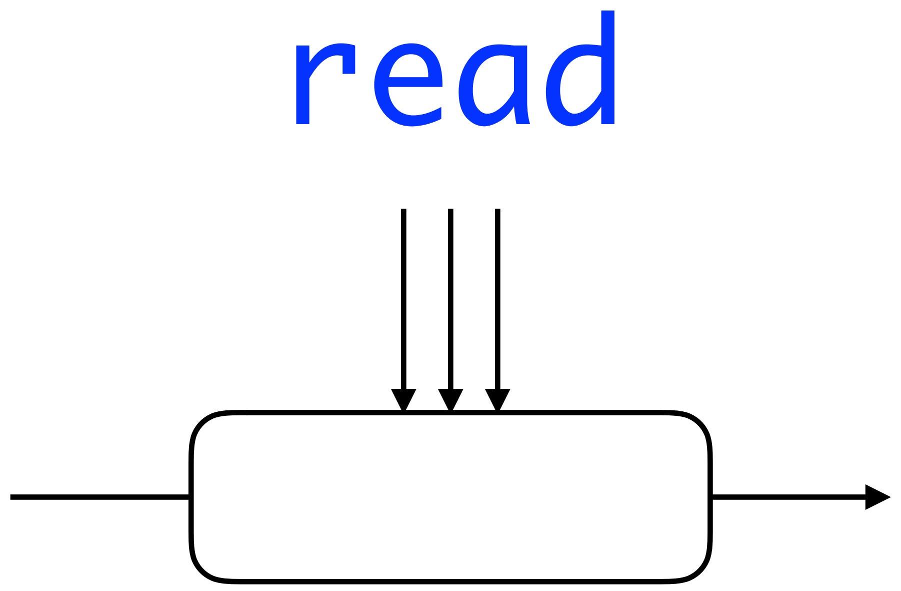


apply

f



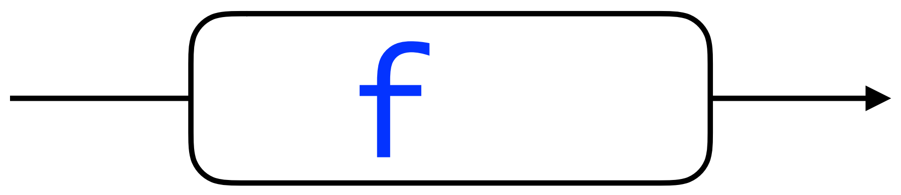
write



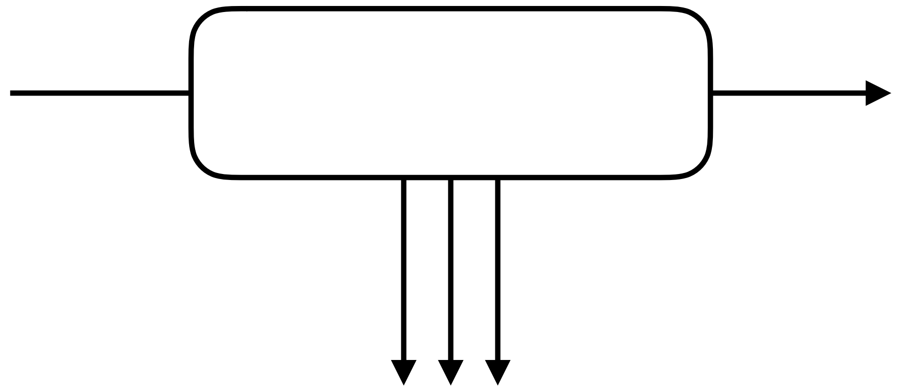
read

And some primitives …

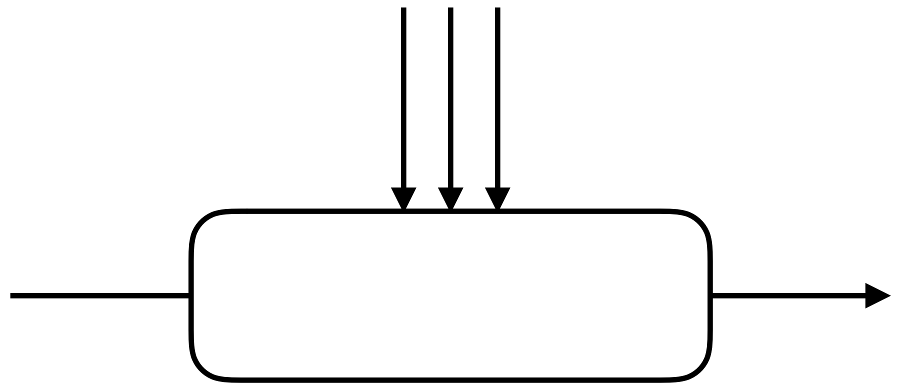
apply



write



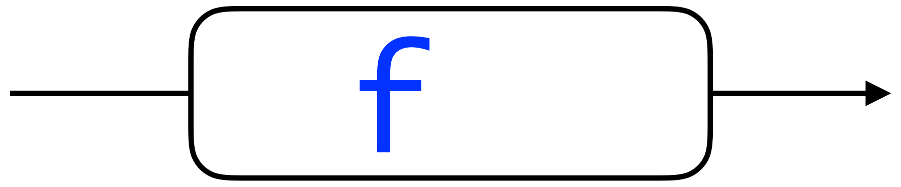
read



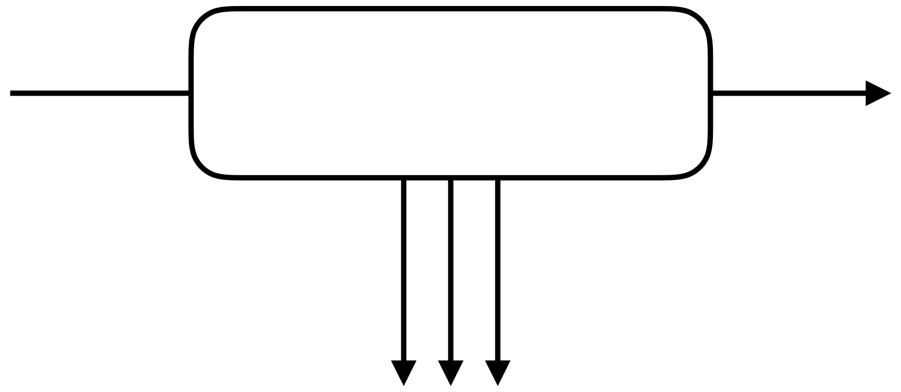
forget   start   change   wait

And some primitives …

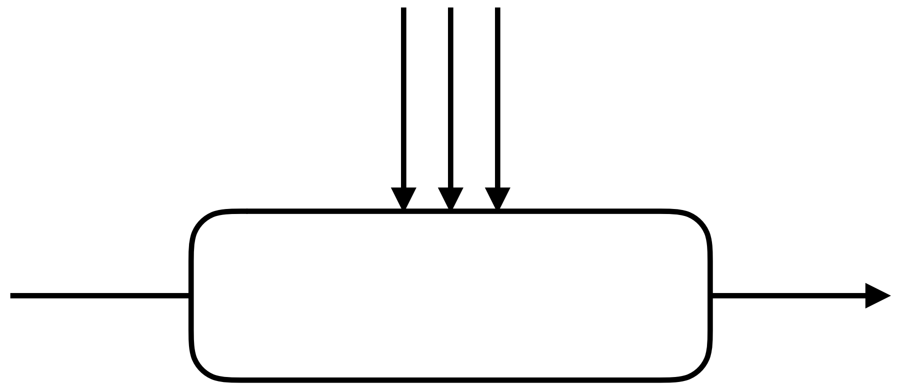
apply



write



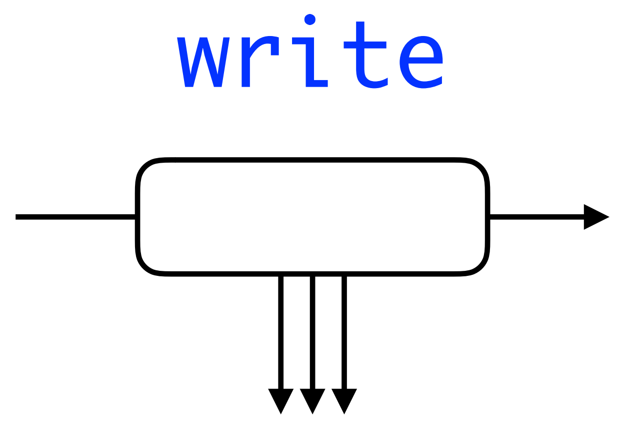
read



`forget   start   change   wait`

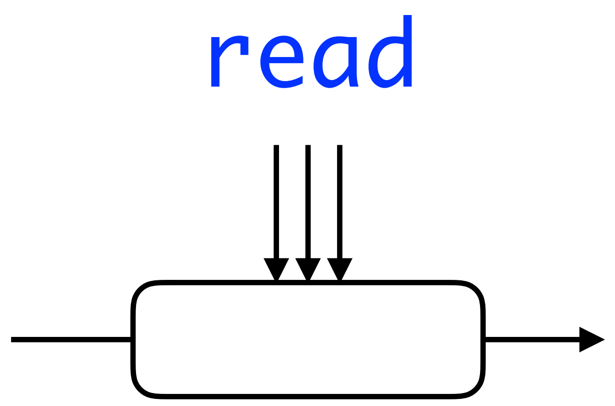
```
run :: Interact a b -> a -> IO ()
```

And some primitives …

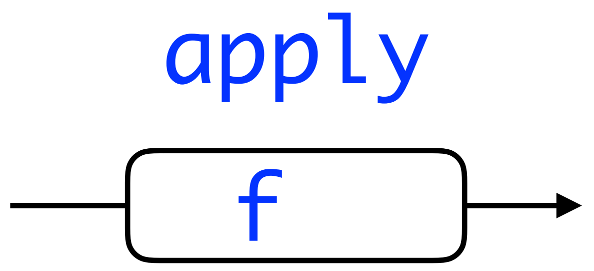


```
write :: String -> Interact a a

write outstring (input,st)
  = (input,st,[outstring])
```

```
run :: Interact a b -> a -> IO ()

run inter st
  = interact (\chs ->
       case inter (split chs,st) of
         (_,_,out) -> join out ++ "\n")
```



```
readin :: Interact () String

readin (input,())
  = (tail input, head input,[])
```



```
apply :: (a -> b) -> Interact a b

apply f (input,st)
  = (input, f st , [])
```

Demo

Copy input

```
copy :: Interact () ()

copy = while (\_ -> True) (readin `sq` writeout id)
```

Copy input

```
copy :: Interact () ()

copy = while (\_ -> True) (readin `sq` writeout id)
```

Imperative style

Copy input

```
copy :: Interact () ()

copy = while (\_ -> True) (readin `sq` writeout id)
```

Imperative style

```
copy :: Interact () ()

copy = readin `sq` writeout id `sq` copy
```

Copy input

```
copy :: Interact () ()

copy = while (\_ -> True) (readin `sq` writeout id)
```

Imperative style

```
copy :: Interact () ()

copy = readin `sq` writeout id `sq` copy
```

A little
meta-circularity

Input N then sum N numbers

```
collector :: Interact () (Int,Int)

collector
  =  getInt `sq`
     add_val_right 0 `sq`
     while ((>(0::Int)).fst.snd)
         (add_val_left () `sq`
          pass_on getInt `sq`
          apply (\(p,(m,s))->(m-1,s+p)) `sq`
          wait `sq`
          showkeep)
```

Input N then sum N numbers

```
collector :: Interact () (Int,Int)

collector
  =  getInt `sq`                                           counter
     add_val_right 0 `sq`
     while ((>(0::Int)).fst.snd)
         (add_val_left () `sq`
          pass_on getInt `sq`
          apply (\(p,(m,s))->(m-1,s+p)) `sq`
          wait `sq`
          showkeep)
```

Input N then sum N numbers

```
collector :: Interact () (Int,Int)

collector
  =  getInt `sq`                                            counter
     add_val_right 0 `sq`                                  (counter,sum)
     while ((>(0::Int)).fst.snd)
         (add_val_left () `sq`
          pass_on getInt `sq`
          apply (\(p,(m,s))->(m-1,s+p)) `sq`
          wait `sq`
          showkeep)
```

Input N then sum N numbers

```
collector :: Interact () (Int,Int)

collector
  =  getInt `sq`                                           counter
     add_val_right 0 `sq`                                 (counter,sum)
     while ((>(0::Int)).fst.snd)                          (counter,sum)
         (add_val_left () `sq`
          pass_on getInt `sq`
          apply (\(p,(m,s))->(m-1,s+p)) `sq`
          wait `sq`
          showkeep)
```

Input N then sum N numbers

```
collector :: Interact () (Int,Int)

collector
  =  getInt `sq`                                           counter
     add_val_right 0 `sq`                                 (counter,sum)
     while ((>(0::Int)).fst.snd)                          (counter,sum)
         (add_val_left () `sq`                       ((), (counter,sum))
          pass_on getInt `sq`
          apply (\(p,(m,s))->(m-1,s+p)) `sq`
          wait `sq`
          showkeep)
```

Input N then sum N numbers

```
collector :: Interact () (Int,Int)

collector
  =  getInt `sq`                                           counter
     add_val_right 0 `sq`                                 (counter,sum)
     while ((>(0::Int)).fst.snd)                          (counter,sum)
         (add_val_left () `sq`                       ((), (counter,sum))
          pass_on getInt `sq`                        (Int,(counter,sum))
          apply (\(p,(m,s))->(m-1,s+p)) `sq`
          wait `sq`
          showkeep)
```

Input N then sum N numbers

```
collector :: Interact () (Int,Int)

collector
  =  getInt `sq`                                           counter
     add_val_right 0 `sq`                                 (counter,sum)
     while ((>(0::Int)).fst.snd)                          (counter,sum)
         (add_val_left () `sq`                       ((), (counter,sum))
          pass_on getInt `sq`                        (Int,(counter,sum))
          apply (\(p,(m,s))->(m-1,s+p)) `sq`              (counter,sum)
          wait `sq`
          showkeep)
```

Input N then sum N numbers

```
collector :: Interact () (Int,Int)

collector
  =  getInt `sq`                                           counter
     add_val_right 0 `sq`                                 (counter,sum)
     while ((>(0::Int)).fst.snd)                          (counter,sum)
         (add_val_left () `sq`                       ((), (counter,sum))
          pass_on getInt `sq`                        (Int,(counter,sum))
          apply (\(p,(m,s))->(m-1,s+p)) `sq`              (counter,sum)
          wait `sq`                                        :-)
          showkeep)
```

Input N then sum N numbers

Make the state abstract, with accessors, mutators etc.

```
collector :: Interact () (Int,Int)

collector
  =  getInt `sq`                                          counter
     add_val_right 0 `sq`                                (counter,sum)
     while ((>(0::Int)).fst.snd)                         (counter,sum)
         (add_val_left () `sq`                      ((), (counter,sum))
          pass_on getInt `sq`                       (Int,(counter,sum))
          apply (\(p,(m,s))->(m-1,s+p)) `sq`             (counter,sum)
          wait `sq`                                       :-)
          showkeep)
```

Input N then sum N numbers

```
collectNums :: Interact Int Int

collectNums
  = addNum
      `pass_param`
           (\n -> start 0 `sq`
                  seqlist (replicate n addNum) `sq`
                  write "finished")
```

Input N then sum N numbers

Leave the internal state and synthesise a program.

```
collectNums :: Interact Int Int

collectNums
  = addNum
      `pass_param`
          (\n -> start 0 `sq`
                 seqlist (replicate n addNum) `sq`
                 write "finished")
```

Looking back

All the ingredients were there …

Higher-order functions

Lazy evaluation

Pattern matching

Algebraic data types

… well, almost all

Miranda had no `lambda`, or `let`.

- A variant of "point-free" style … need to name abstractions.

Equality overloaded, similarly printing values, but no `class` / `instance` …

Few established “design patterns”

The model mixes aspects of

- `Monad`
- `Arrow`
- `Applicative`

The linguistic turn …

Can see this as a *shallow embedding* of an interaction language.

What would happen if we made that deep?

The linguistic turn …

Can see this as a *shallow embedding* of an interaction language.

What would happen if we made that deep?

```
data Inter =
   While Cond Inter |
   Alt Cond Inter Inter |
   Seq Inter Inter |
   …

interpret ::
  Inter -> Interact Int Int
```

The linguistic turn …

Can see this as a *shallow embedding* of an interaction language.

What would happen if we made that deep?

Questions of reflection, dependent types etc.

```
data Inter =
   While Cond Inter |
   Alt Cond Inter Inter |
   Seq Inter Inter |
   …


interpret ::
  Inter -> Interact Int Int
```

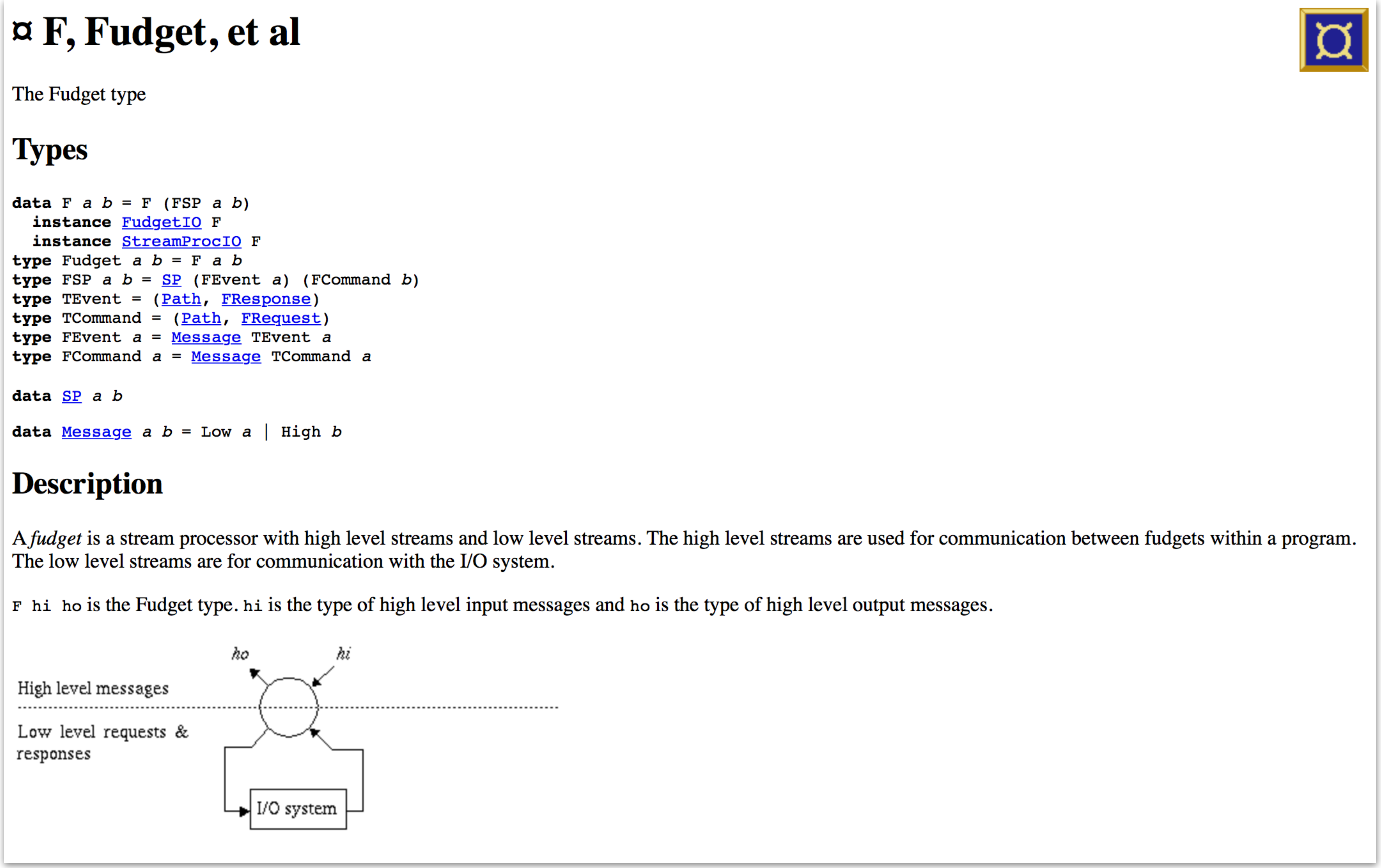
Types

The fundamental scope of values hasn’t changed …

… but their classifications have.

Roles for e.g. GADTs, dependency here, especially with DSLs?

http://www.altocumulus.org/Fudgets/



¤ F, Fudget, et al

The Fudget type

Types

```
data F a b = F (FSP a b)
  instance FudgetIO F
  instance StreamProcIO F
type Fudget a b = F a b
type FSP a b = SP (FEvent a) (FCommand b)
type TEvent = (Path, FResponse)
type TCommand = (Path, FRequest)
type FEvent a = Message TEvent a
type FCommand a = Message TCommand a

data SP a b

data Message a b = Low a | High b
```

Description

A *fudget* is a stream processor with high level streams and low level streams. The high level streams are used for communication between fudgets within a program. The low level streams are for communication with the I/O system.

`F hi ho` is the Fudget type. `hi` is the type of high level input messages and `ho` is the type of high level output messages.

Compilation

Libraries

Interop e.g FFI

APIs

Tools

???

Concurrency

Community

And what hasn’t happened?

Routine verification … semantics.

Compilers derived from semantics.

The end of the program as text.

Special purpose parallel hardware.

https://github.com/simonjohnthompson/Interaction

Code and slides are available now at

https://github.com/simonjohnthompson/Interaction

This presentation will soon be available at

https://skillsmatter.com/conferences/8522-haskell-exchange-2017#skillscasts