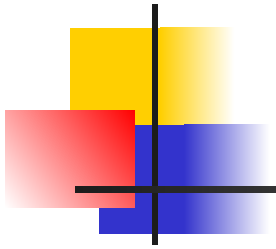
Lazy functional programming for real

Tackling the Awkward Squad

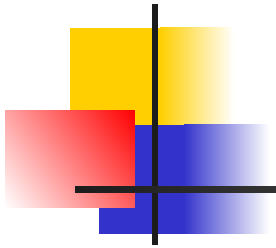
Simon Peyton Jones

Microsoft Research



Beauty and the Beast

- Functional programming is beautiful, and many books tell us why
- But to write real applications, we must deal with un-beautiful issues "around the edges":
  - Input/output
  - Concurrency
  - Error recovery
  - Foreign-language interfaces



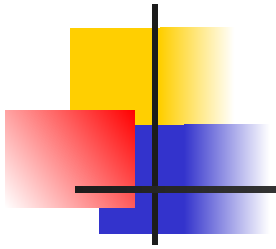
The direct approach

Do everything in "the usual way"

- **I/O** via "functions" with side effects

```
putchar 'x' + putchar 'y'
```

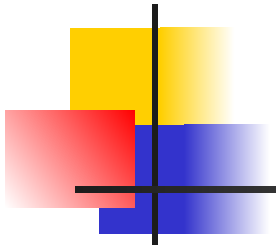
- **Concurrency** via operating system threads; OS calls mapped to "functions"
- **Error recovery** via exceptions
- **Foreign language procedures** mapped to "functions"



The lazy hair shirt

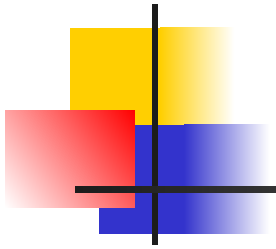
In a lazy functional language, like Haskell, order of evaluation is deliberately undefined.

- `putchar 'x' + putchar 'y'`
  Output depends on evaluation order of (+)
- `[putchar 'x', putchar 'y]`
  Output (if any) depends on how the consumer evaluates the list



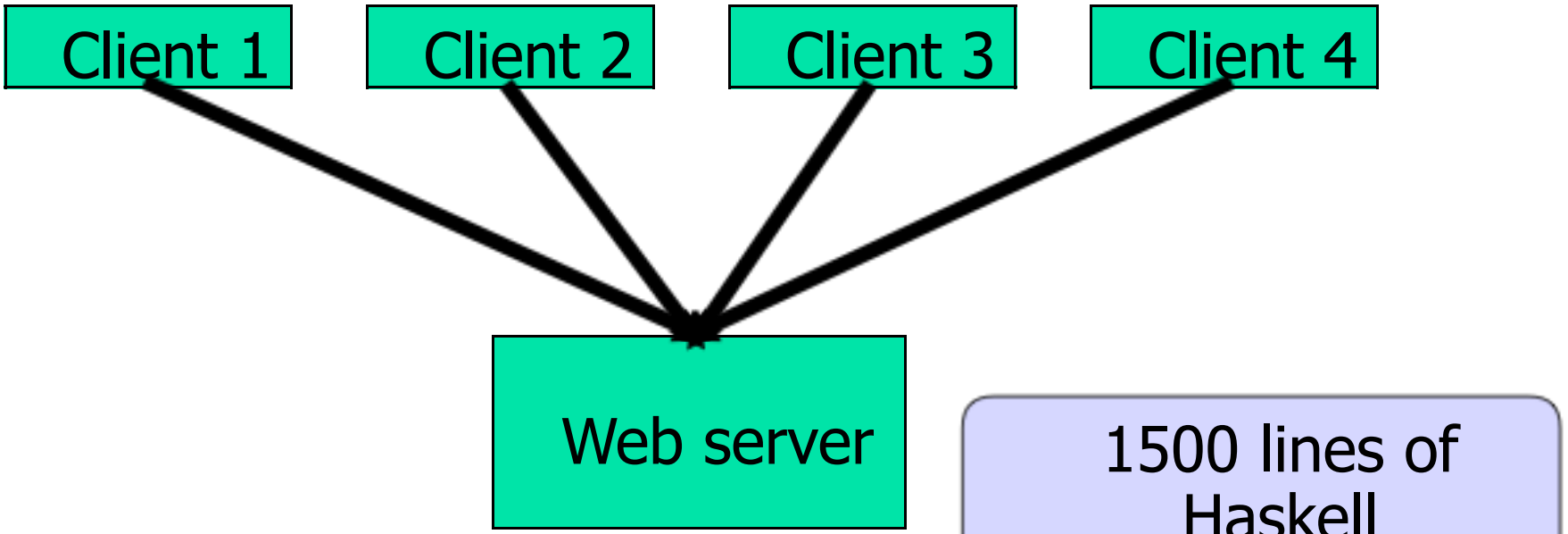
Tackling the awkward squad

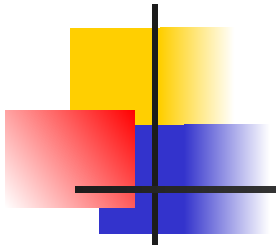
- So lazy languages force us to take a different, more principled, approach to the Awkward Squad.
- I'm going to describe that approach for Haskell
- But I think there are lessons for strict, or even imperative, languages too.



A web server

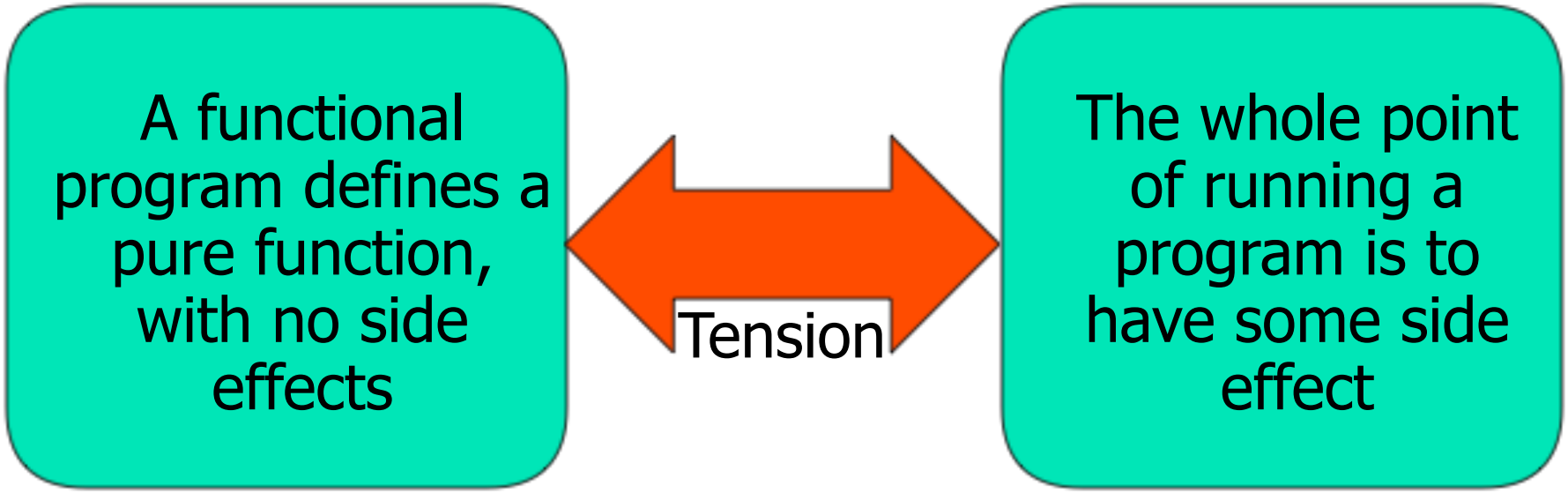
- I'll use a web server as my motivating example
- Lots of I/O, lots of concurrency, need for error recovery, need to call external libraries





Monadic input and output

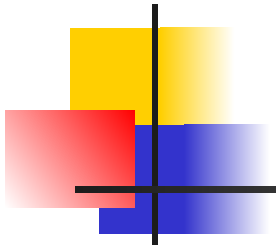
The problem


A functional program defines a pure function, with no side effects
Tension
The whole point of running a program is to have some side effect

Functional I/O

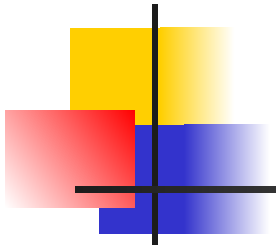
```
main :: [Response] -> [Request]

data Request    = ReadFile Filename
        | WriteFile FileName String
        | …

data Response    = RequestFailed
        | ReadOK String
        | WriteOk
        | …
```

- "Wrapper program" interprets requests, and adds responses to input



Functional I/O is awkward

- Hard to extend (new I/O operations new constructors)
- No connection between **`Request`** and corresponding **`Response`**
- Easy to get "out of step" (deadlock)



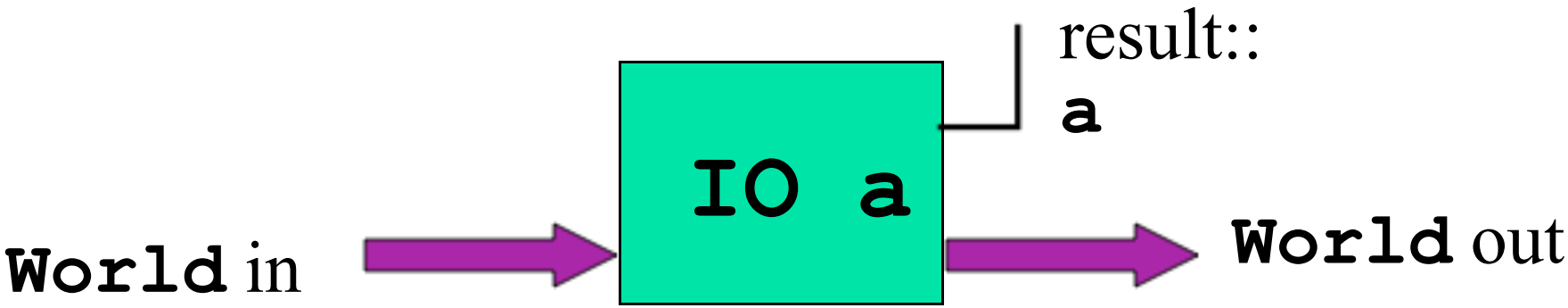
Monadic I/O: the key idea

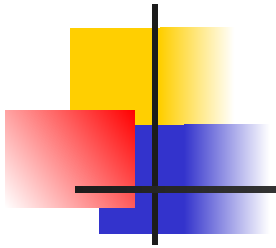
A value of type (`IO t`) is an "action" that, when performed, may do some input/output before delivering a result of type t.

A helpful picture

A value of type (`IO t`) is an "action" that, when performed, may do some input/output before delivering a result of type t.

```
type IO a = World -> (a, World)
```





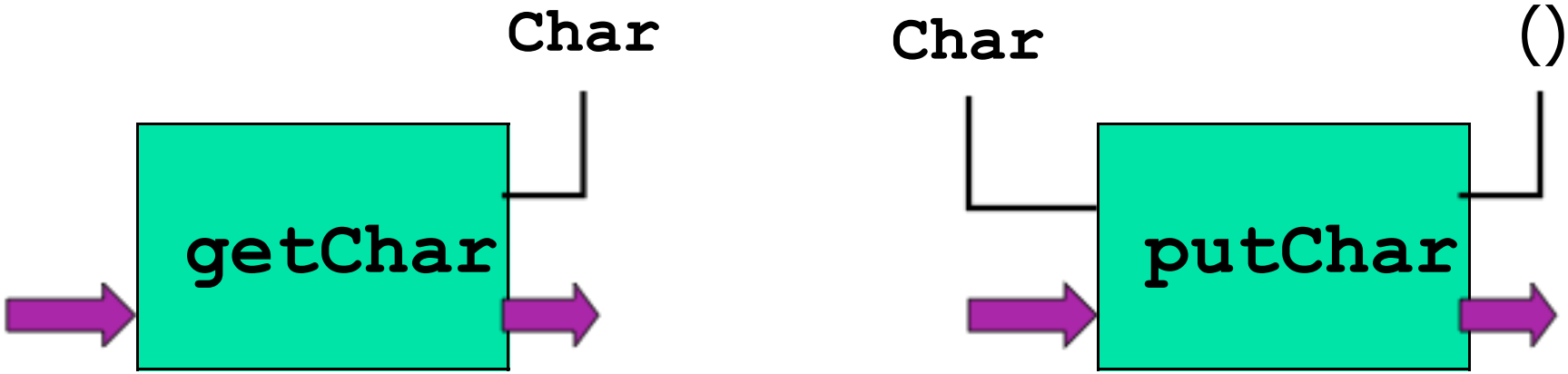
Actions are first class

A value of type (`IO t`) is an "action" that, when performed, may do some input/output before delivering a result of type t.

```
type IO a = World -> (a, World)
```

- "Actions" sometimes called "computations"
- An action is a first class value
- Evaluating an action has no effect; performing the action has an effect

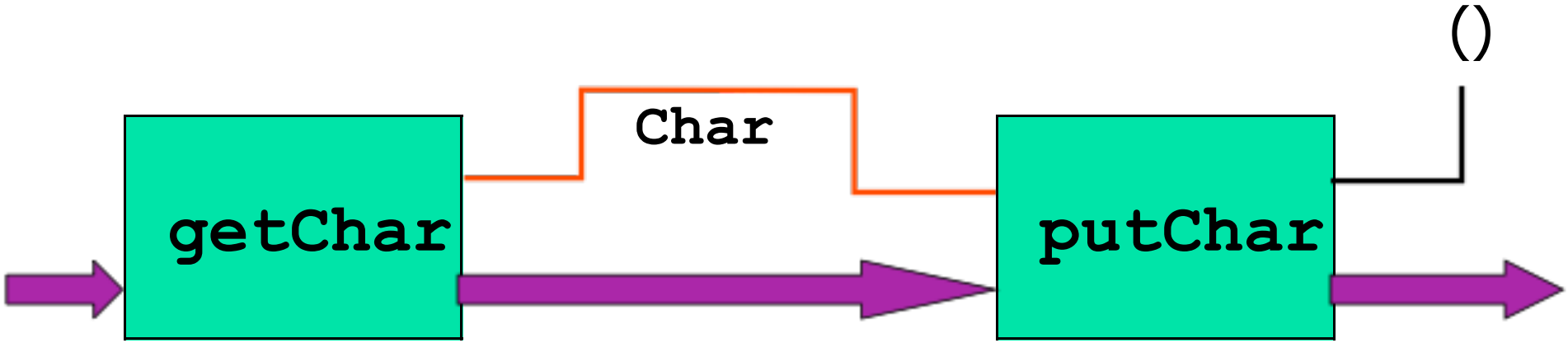
Simple I/O




```
getChar :: IO Char
putChar :: Char -> IO ()


main :: IO ()
main = putChar 'x'
```

Main program is an action of type IO ()

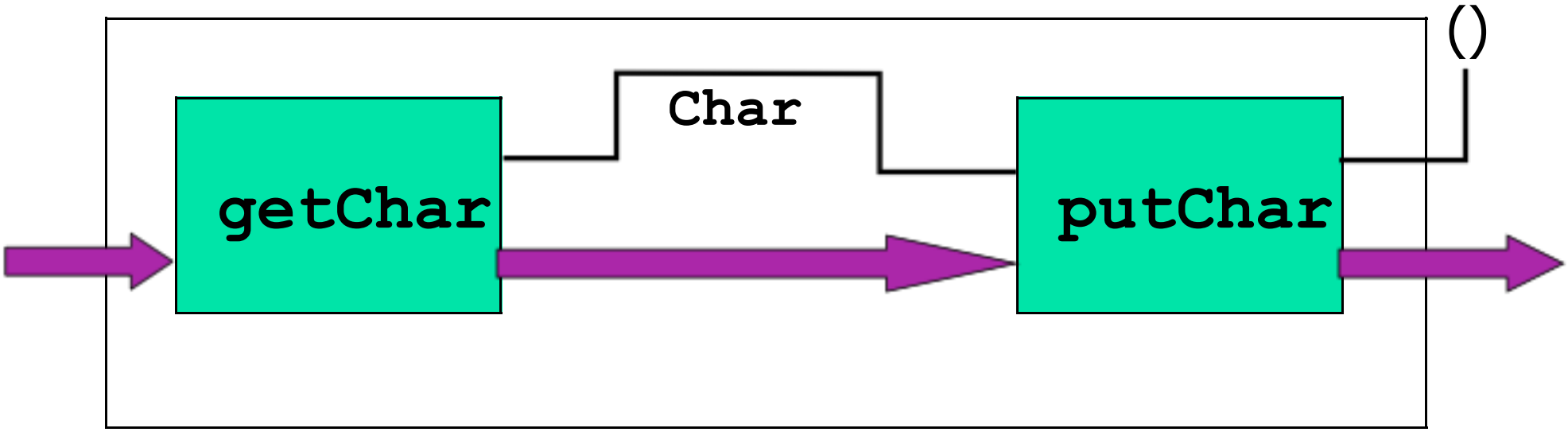
Connecting actions up



Goal: read a character and then write it back out

The (>>=) combinator

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```



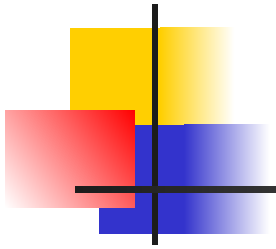
- We have connected two actions to make a new, bigger action.

```
echo :: IO ()
echo = getChar >>= putChar
```

Printing a character twice

```
echoDup :: IO ()
echoDup = getChar     >>= (\c ->
            putChar c     >>= (\()
  ->
            putChar c))
```

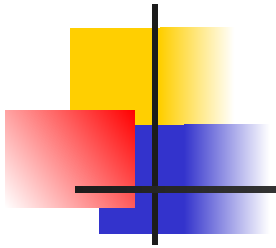
- The parentheses are optional



The (>>) combinator

```
echoDup :: IO ()
echoDup = getChar     >>= \c ->
          putChar c     >>
          putChar c
```

```
(>>) :: IO a -> IO b -> IO b
m >> n  =  m >>= (\x -> n)
```



Getting two characters

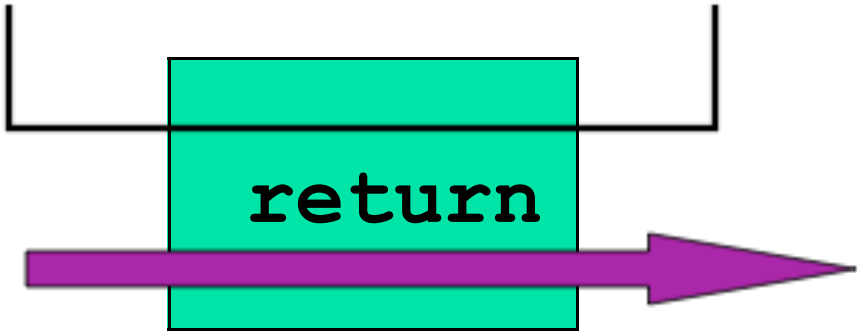
```
getTwoChars :: IO (Char,Char)
getTwoChars =    getChar    >>= \c1
  ->
        getChar    >>= \c2 ->
        ????
```

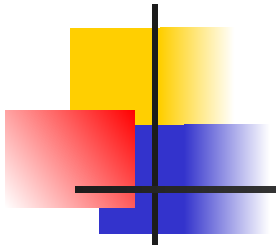
We want to just return (c1,c2)

The **return** combinator

```
getTwoChars :: IO (Char,Char)
getTwoChars =    getChar    >>= \c1
  ->
        getChar    >>= \c2 ->
        return (c1,c2)
```

return :: a -> IO a



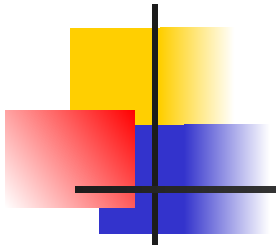


Notational convenience

```
getTwoChars :: IO (Char,Char)
getTwoChars =    getChar    >>= \c1 ->
        getChar    >>= \c2 ->
        return (c1,c2)
```

- By design, the layout looks imperative

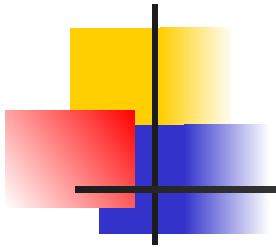
```
c1 = getchar();
  c2 = getchar();
  return (c1,c2);
```



Notational convenience

```
getTwoChars :: IO (Char,Char)
getTwoChars =     do {     c1 <- getChar
  ;                 c2 <- getChar ;
     return (c1,c2) }
```

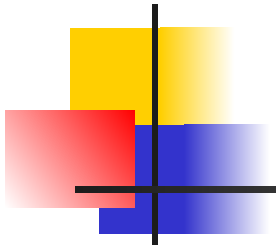
"do" notation adds only syntactic sugar



Desugaring do notation

"do" notation adds only syntactic sugar

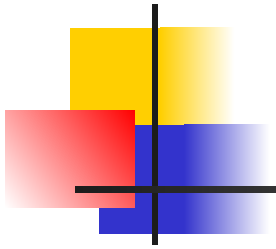
```
do { x<-e; s }   =   e >>= \x -> do { s }
do { e; s }   =   e >> do { s }
do { e }   =   e
```



Getting a line

```
getLine :: IO [Char]
getLine = do { c <- getChar;
               if c == '\n' then
                   return []
               else do {     cs <- getLine;
                             return (c:cs) }
          }
```

Note the nested "do" expression



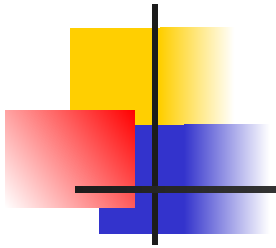
Control structures

Values of type (IO t) are first class

So we can define our own "control structures"

```
forever :: IO () -> IO ()
forever a = a >> forever a

repeatN :: Int -> IO () -> IO ()
repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a
```

e.g. `repeatN 10 (putChar 'x')`



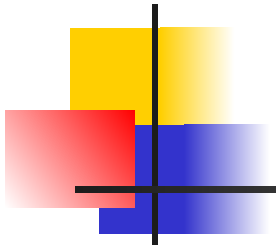
Loops

Values of type (IO t) are first class

So we can define our own "control structures"

```
for :: [a] -> (a -> IO b) -> IO ()
for []     fa = return ()
for (x:xs) fa = fa x >> for xs fa
```

e.g. `for [1..10] (\x -> putStr (show x))`

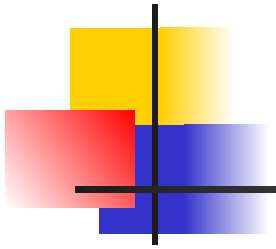


Loops

A list of IO actions

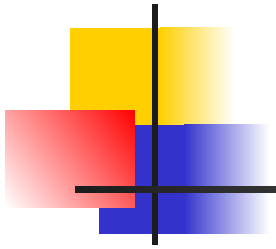
An IO action returning a list

```
sequence :: [IO a] -> IO [a]
sequence []     = return []
sequence (a:as) = do { r <- a;
                       rs <- sequence as;
                       return (r:rs) }

for :: [a] -> (a -> IO b) -> IO ()
for xs fa = sequence (map fa xs) >>
            return ()
```



First class actions

Slogan: first class actions let us write application-specific control structures

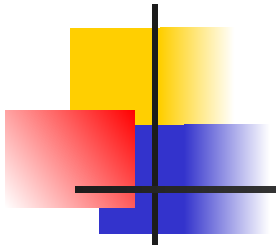


Monads

A monad consists of:

- A type constructor M
- bind :: M a -> (a -> M b) -> M b
- unit :: a -> M a

PLUS some laws

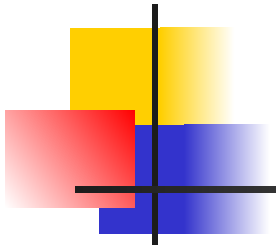


Laws for monads

$$\text{unit } x \;>\!\!>\!\!=\; f \;=\; f\ x$$

$$m \;>\!\!>\!\!=\; \text{unit} \;=\; m$$

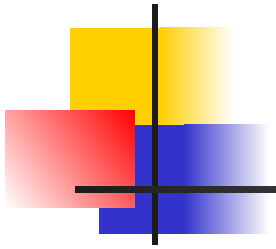
$$m1 \;>\!\!>\!\!=\; (\lambda x.m2 \;>\!\!>\!\!=\; (\lambda\ y.m3)) \;=\; (m1 \;>\!\!>\!\!=\; (\lambda\ x.m2)) \;>\!\!>\!\!=\; (\lambda\ y.m3)$$



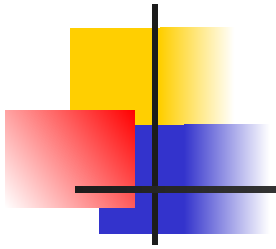
Third law in do-notation

```
do {   x <- do {   y <- m2 ;     =    do {   y <- m2;
         m1 } ;                           x <- m1;
       m3 }                             m3 }
```

$$m1 >>= (\lambda x.m2 >>= (\lambda y.m3)) = (m1 >>= (\lambda x.m2)) >>= (\lambda y.m3)$$



What does it all mean?



Functional semantics

```
type IO a = World -> (a, World)
```

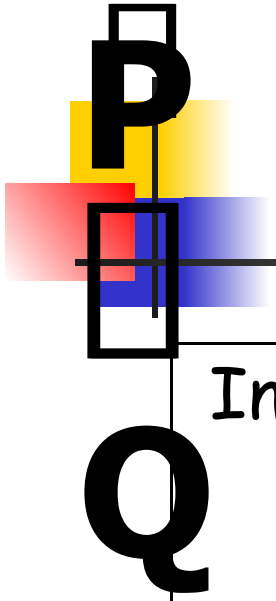
- A program that loops forever has meaning ⊥
  A program that prints 'x' forever has meaning ⊥
- What is the meaning of two Haskell programs running in parallel?
- Does not scale well to concurrency, non-determinism

Trace semantics

```
type IO a   = Set Trace
type Trace = [Event]
type Event = ReadChar Char
           | WriteChar Char
           …
```

echo = { [ReadChar ‘a’; WriteChar ‘a’],
[ReadChar ‘b’; WriteChar ‘b’],
[ReadChar ‘c’; WriteChar ‘c’],
…}

- Can deal with non-determinism, and concurrency

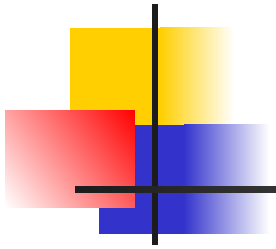


Operational semantics

Instead of saying what the meaning of a program **is**, say how the program **behaves**

Equivalance of programs becomes **similarity of behaviour** instead of **identity of meaning**

Program state P can move to state Q, exchanging event with the environment



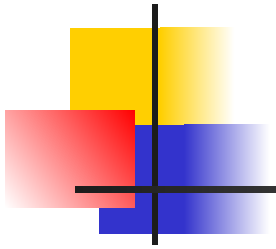
Program states

A program state represents the current internal state of the program.

Initially, it is just a term, {M}

Curly braces say "here is a program state"

e.g. `{putChar 'x' >> putChar 'y'}`

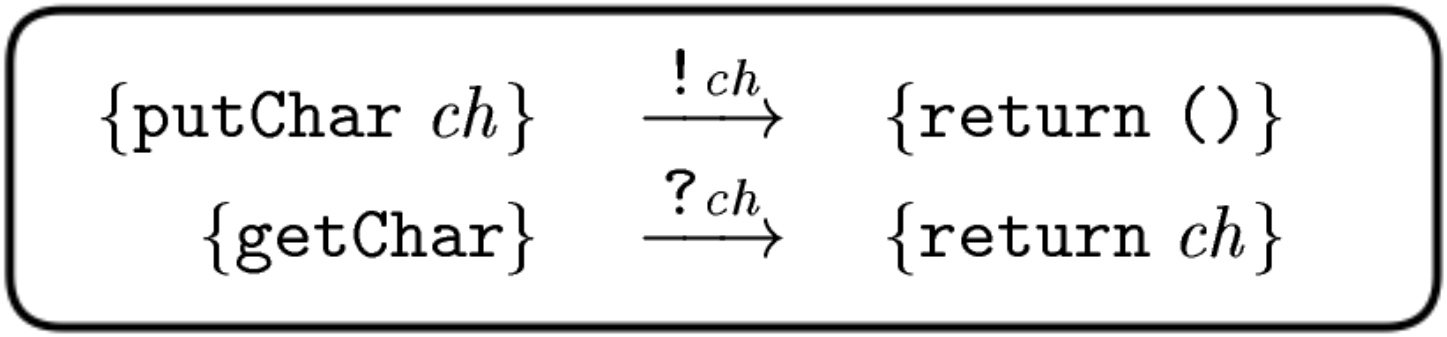


Events

Events describe how the program interacts with the external world: i.e. what I/O it performs

- $P \xrightarrow{!c} Q$ P can move to Q, writing c to stdout
- $P \xrightarrow{?c} Q$ P can move to Q, reading c to stdin

Our first two rules



But: what happens to this?

```
{getChar >>= \c-> putChar c} □ ???
```

Want to say "look at the action in the leftmost position"

Evaluation contexts

$$E ::= [.] \mid E >>= M$$

An **evaluation context** E is a term with a "hole" in it.
For example:

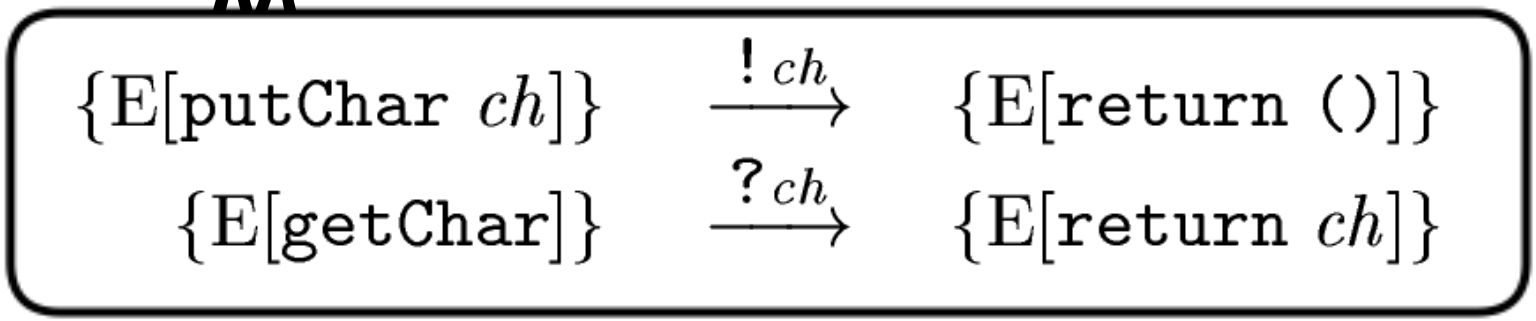
E1 = [.] >>= M
E2 = ([.] >>= M1) >>= M2

E[M] is the evaluation context E with the hole filled in by M. So

E1[getChar] = getChar >>= M
E2[getChar] = (getChar >>= M1) >>= M2

Revised rules for put/get

E ::= [.] | E >>=




```
{getChar >>= \c-> putChar c}
[] {return ch >>= \c-> putChar c}
```

?ch

The return/bind rule

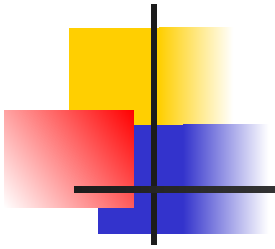
$$\{\mathrm{E}[\texttt{return}\ N\ \texttt{>>=}\ M]\} \quad \rightarrow \quad \{\mathrm{E}[M\ N]\}$$

Silent transition

```
{getChar >>= \c-> putChar c}
? ch
 {return ch >>= \c-> putChar c}
 {(\c-> putChar c) ch}
```

Now we need to do some "ordinary evaluation"

The evaluation rule



$\mathcal{E}[\![M]\!]$ is the denotational semantics for M

V is the value of M

M wasn't already evaluated

If the things above the line are true, then we can deduce the thing below the line

$$\frac{\mathcal{E}[\![M]\!] = V \quad M \not\equiv V}{\{\mathrm{E}[M]\} \rightarrow \{\mathrm{E}[V]\}}$$

The evaluation rule

$$\frac{\mathcal{E}[\![M]\!] = V \quad M \neq V}{\{\mathrm{E}[M]\} \rightarrow \{\mathrm{E}[V]\}}$$

ε ⟦M⟧ is the denotational semantics for M

® {(\c-> putChar c) *ch*}

® {putChar *ch*}

!*ch* ® {return ()}

Treat primitive IO actions as "constructors"; so (**putChar ch**) is a value

Mutable variables in C

```
int x =
3;
x = x+1;
```

x is a location,initalised with 3

read x, add 1, store into x

Mutable variables in Haskell

```
do { x <- newIORef 3;
     v <- readIORef x;
     writeIORef x (v+1) }
```

x is a location, initalised with 3

read x, add 1, store into x

```
newIORef    :: a -> IO (IORef a)
readIORef   :: IORef a -> IO a
writeIORef  :: IORef a -> a -> IO
 ()
```

Semantics for variables

Step 1: elaborate the program state

P, Q, R	$::=$	$\{M\}$	The main program
	$\mid$	$\langle M \rangle_r$	An `IORef` named r, holding M
	$\mid$	$P \mid Q$	Parallel composition
	$\mid$	$\nu x.P$	Restriction

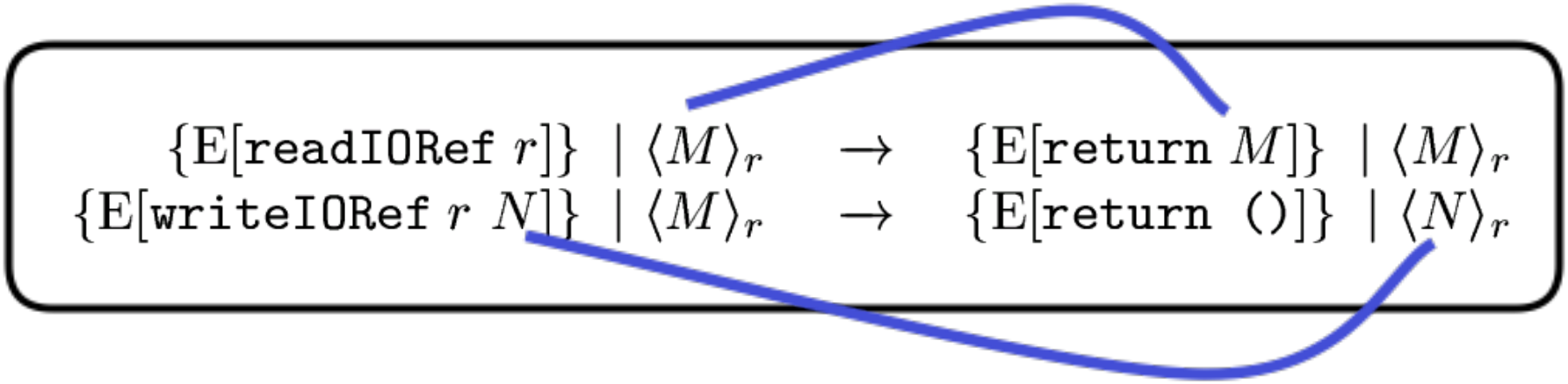
e.g. □r,s. ({M} | <3>r | <89>s)

The main program

An IORef called r, holding 3

Semantics for variables

Step 2: add rules for read, write IORefs



$$\{\mathrm{E}[\texttt{readIORef}\ r]\} \mid \langle M \rangle_r \rightarrow \{\mathrm{E}[\texttt{return}\ M]\} \mid \langle M \rangle_r$$
$$\{\mathrm{E}[\texttt{writeIORef}\ r\ N]\} \mid \langle M \rangle_r \rightarrow \{\mathrm{E}[\texttt{return}\ ()]\} \mid \langle N \rangle_r$$

Semantics for variables

Step 2: add rules for read, write IORefs

$$\begin{array}{rcl} \{\mathrm{E}[\texttt{readIORef}\ r]\} \mid \langle M \rangle_r & \rightarrow & \{\mathrm{E}[\texttt{return}\ M]\} \mid \langle M \rangle_r \\ \{\mathrm{E}[\texttt{writeIORef}\ r\ N]\} \mid \langle M \rangle_r & \rightarrow & \{\mathrm{E}[\texttt{return}\ ()]\} \mid \langle N \rangle_r \end{array}$$

But what if the main program is not "next to" the relevant variable?

"Structural rules"

Can stir the mixture

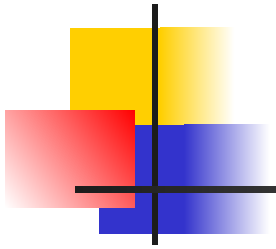
Step 3: add rules to bring "reagents" together

$$P \mid Q \equiv Q \mid P$$
$$P \mid (Q \mid R) \equiv (P \mid Q) \mid R$$

Can look under "|"

$$\frac{P \equiv P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \equiv Q}{P \xrightarrow{\alpha} Q} \; (EQUIV)$$

$$\frac{P \xrightarrow{\alpha} Q}{P \mid R \xrightarrow{\alpha} Q \mid R} \; (PAR)$$

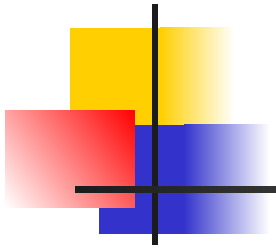


Restriction

Step 4: deal with fresh IORef names

$\{E[\text{newIORef } M]\} \square \{E[\text{return } ?]\} \mid \langle M \rangle ?$

What can we use as the IORef name???



Restriction

Step 4: deal with fresh IORef names

r is not used already

$$\frac{r \notin fn\,(\mathrm{E}, M)}{\{\mathrm{E}[\texttt{newIORef}\ M]\} \ \rightarrow\ \nu r.(\{\mathrm{E}[\texttt{return}\ r]\}\ \mid \langle M \rangle_r)}$$

Restriction

More "Structural rules"

Step 5: structural rules for restriction

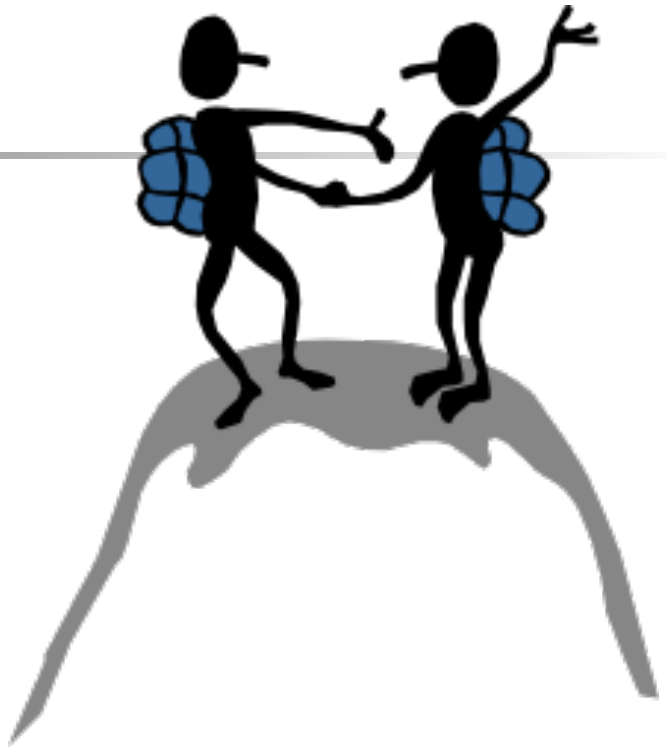
Can float "□" outwards

$$\begin{array}{rcll} \nu x.\nu y.P & \equiv & \nu y.\nu x.P & \\ (\nu x.P) \mid Q & \equiv & \nu x.(P \mid Q), & x \notin fn(Q) \\ \nu x.P & \equiv & \nu y.P[y/x], & y \notin fn(P) \end{array}$$

Can look under "□"

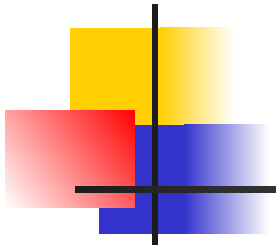
$$\frac{P \xrightarrow{\alpha} Q}{\nu x.P \xrightarrow{\alpha} \nu x.Q}\ (NU)$$

Phew!



Quite a lot of technical machinery! But:

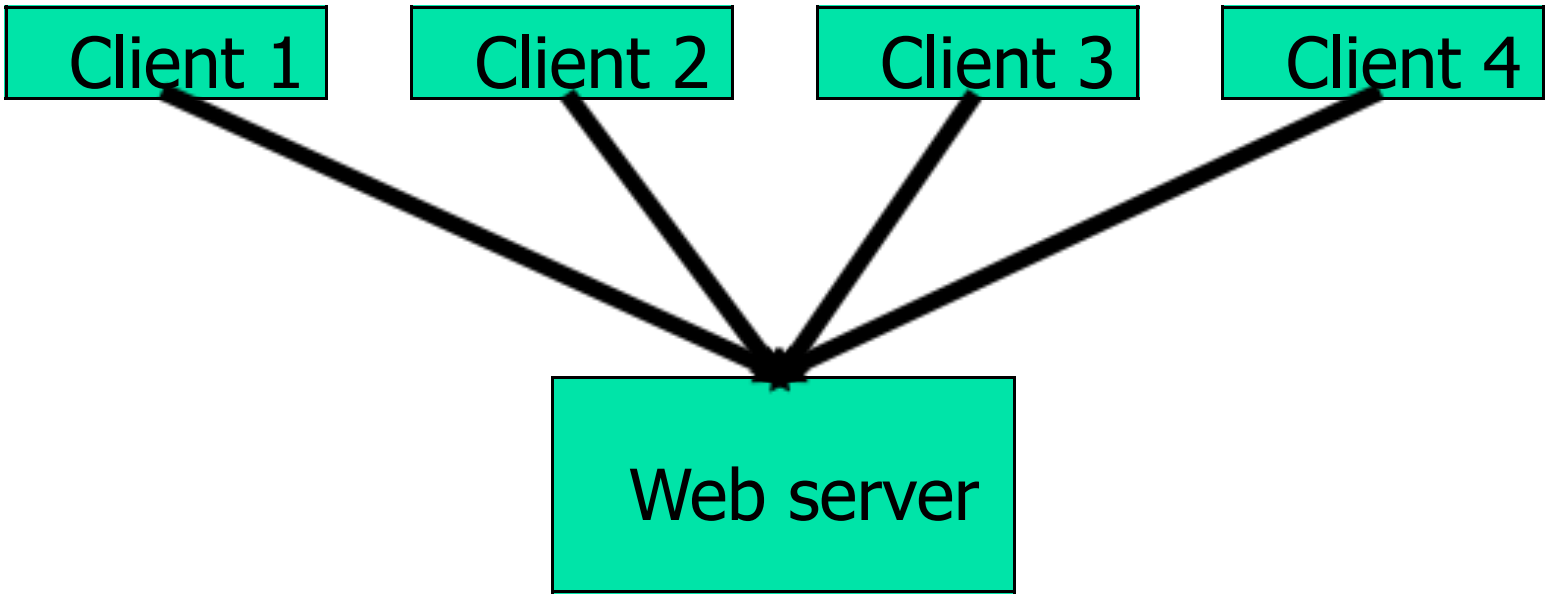
- It's standard, widely-used, machinery (esp in process calculi), so it's worth getting used to
- It scales to handle non-determinism, concurrency
- The machinery (esp structural rules) doesn't get in the way most of the time

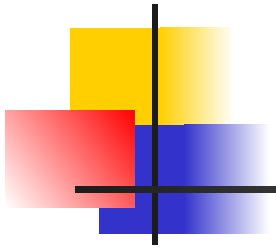


Concurrency

The need for concurrency

- Want one thread ("virtual server") per client
- Threads largely independent, but share some common resources (e.g. file system)





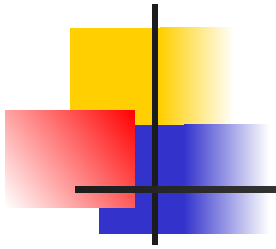
Concurrency vs parallelism

Parallel functional programming:

- Aim = performance through multiple processors (e.g. `e1+e2` in parallel)
- No semantic changes; deterministic results

Concurrent functional programming

- Aim = concurrent, I/O-performing threads
- Makes perfect sense on a uniprocessor
- Non-deterministic interleaving of I/O is inevitable



Concurrent web service

```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket
   = forever (do { conn <- accept socket ;
                   forkIO (serviceConn config conn) })
```

```
forkIO :: IO a -> IO ThreadId
```

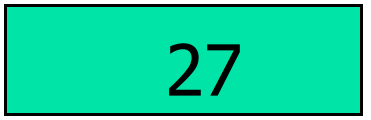
- forkIO spawns an independent, I/O performing, thread
- No parameters passed; free variables do fine

Communication and sharing

- What if two threads want to communicate? Or share data?
- Example: keep a global count of how many client threads are running
  - Increment count when spawning
  - Decrement count when dying

Communication and sharing

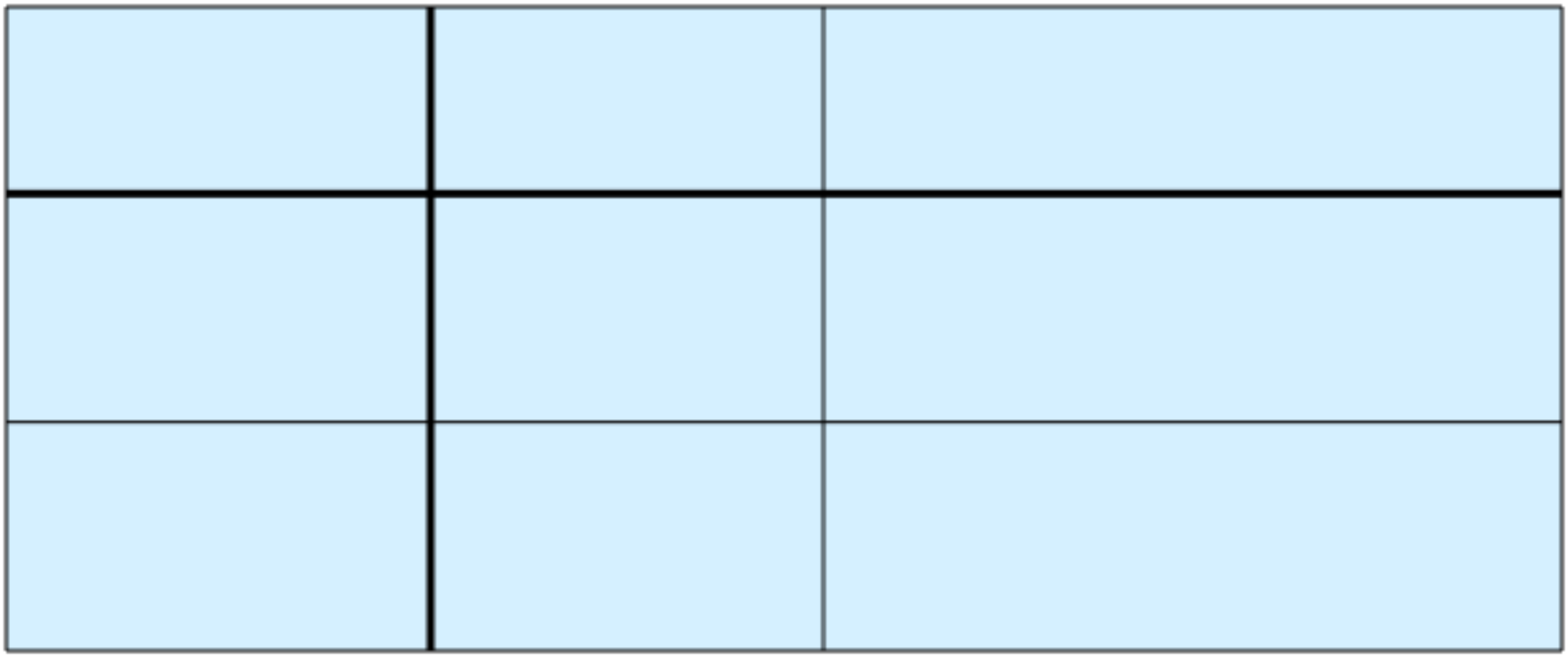
```
data MVar a
newEmptyMVar    :: IO (MVar a)
putMVar     :: MVar a -> a -> IO ()
takeMVar    :: MVar a -> IO a
```

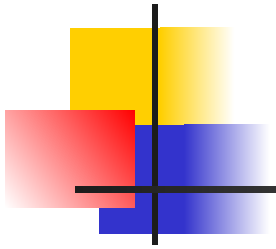
- A value of type `(MVar t)` is a **location** that is either
  - **empty**, or 
  - **holds a value** of type t 

Communication and sharing

takeMVar
putMVar
Block
MVar contents, leave MVar empty

```
takeMVar :: MVar a -> IO a
putMVar  :: MVar a -> a -> IO ()
```





Using MVars

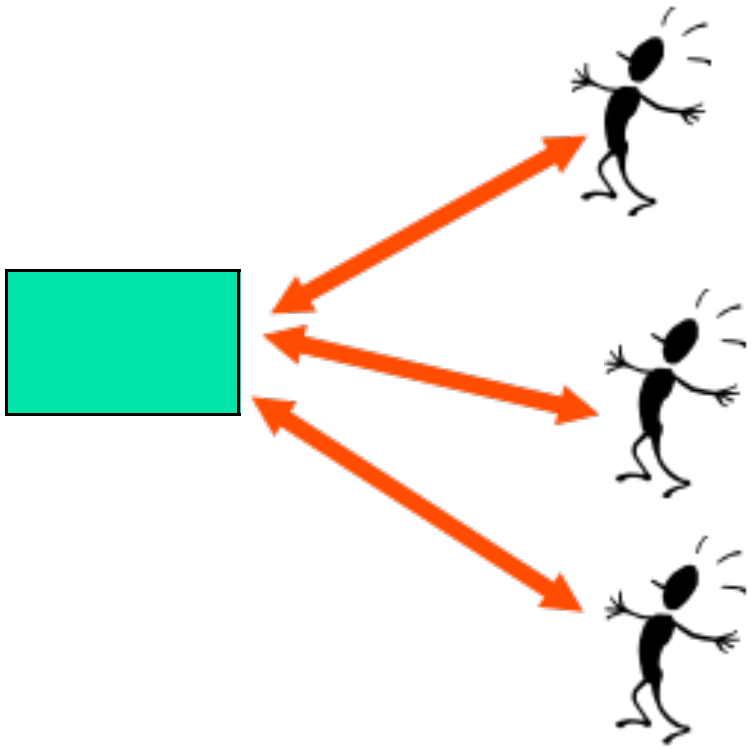
```
acceptConnections :: Config -> Socket -> IO ()
acceptConnections config socket
 = do { count <- newEmptyMVar ;
        putMVar count 0 ;
        forever (do { conn <- accept socket ;
                      forkIO (do { inc count ;
                                   serviceConn config conn ;
                                   dec count}) }

inc,dec :: MVar Int -> IO ()
inc count = do { v <- takeMVar count; putMVar count (v+1) }
dec count = do { v <- takeMVar count; putMVar count (v-1) }
```

MVar is empty at this point, hence no race hazard

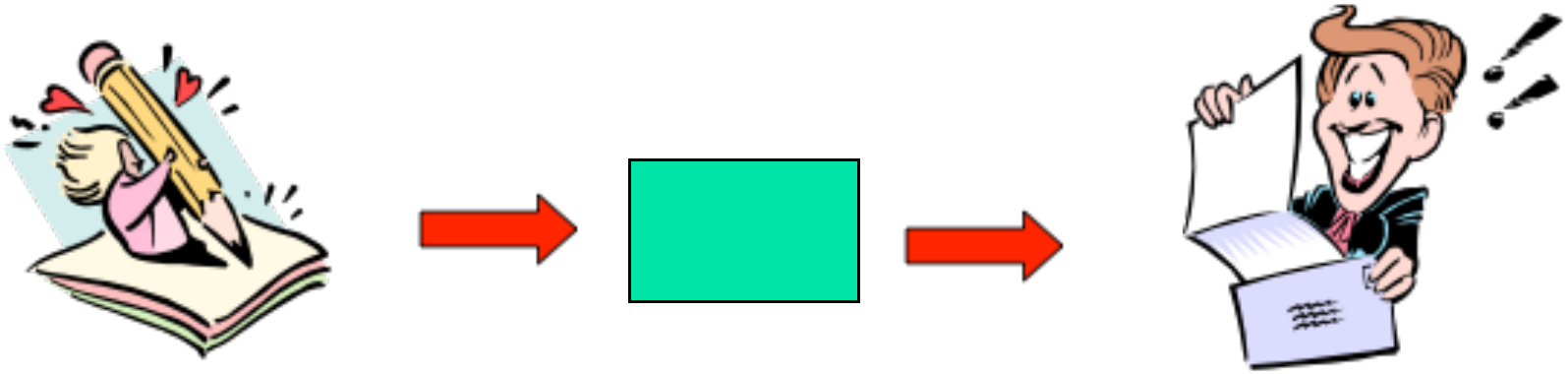
MVars as channels

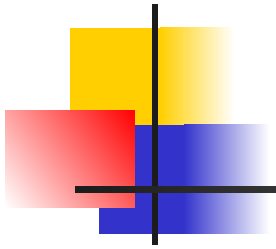
An MVar directly implements:

- a shared data structure



- a one-place channel





Semantics

Fortunately, most of the infrastructure is there already!

Step 1: elaborate the program state

$$
\begin{array}{lcll}
P, Q, R & ::= & \ldots & \\
 & | & \{M\}_t & \text{A thread called } t \\
 & | & \langle M \rangle_m & \text{An } \texttt{MVar} \text{ called } m \text{ containing } M \\
 & | & \langle\rangle_m & \text{An empty } \texttt{MVar} \text{ called } m
\end{array}
$$

e.g. `{putChar 'c'}t1 | {putChar 'd'}t2`

Semantics

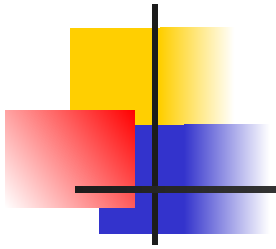
Step 2: a rule for for forkIO

$$\frac{u \notin fn\,(M, \mathrm{E})}{\{\mathrm{E}[\texttt{forkIO}\ M]\}_t \ \rightarrow\ \nu u.(\{\mathrm{E}[\texttt{return}\ u]\}_t \mid \{M\}_u)}\ (FORK)$$

Restrict the new thread name

Return the thread name to the caller

The new thread

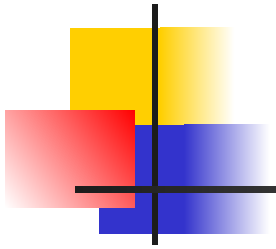


Semantics

Step 3: a rules for new, take, put

$$\{\mathrm{E}[\texttt{takeMVar}\ m]\}_t \mid \langle M\rangle_m \rightarrow \{\mathrm{E}[\texttt{return}\ M]\}_t \mid \langle\rangle_m$$
$$\{\mathrm{E}[\texttt{putMVar}\ M]\}_t \mid \langle\rangle_m \rightarrow \{\mathrm{E}[\texttt{return}\ ()]\}_t \mid \langle M\rangle_m$$

- Same as readIORef, writeIORef, except that MVar is filled/emptied
- Blocking is implicit
- Non-determinism is implicit

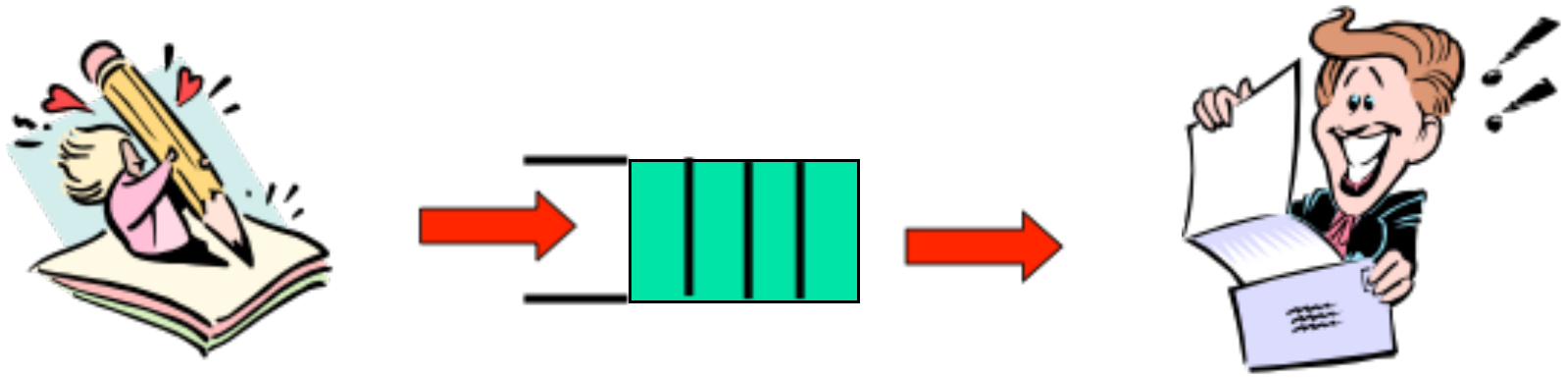


Building abstractions

- MVars are primitive
- Want to build abstractions on top of them
- Example: a buffered channel

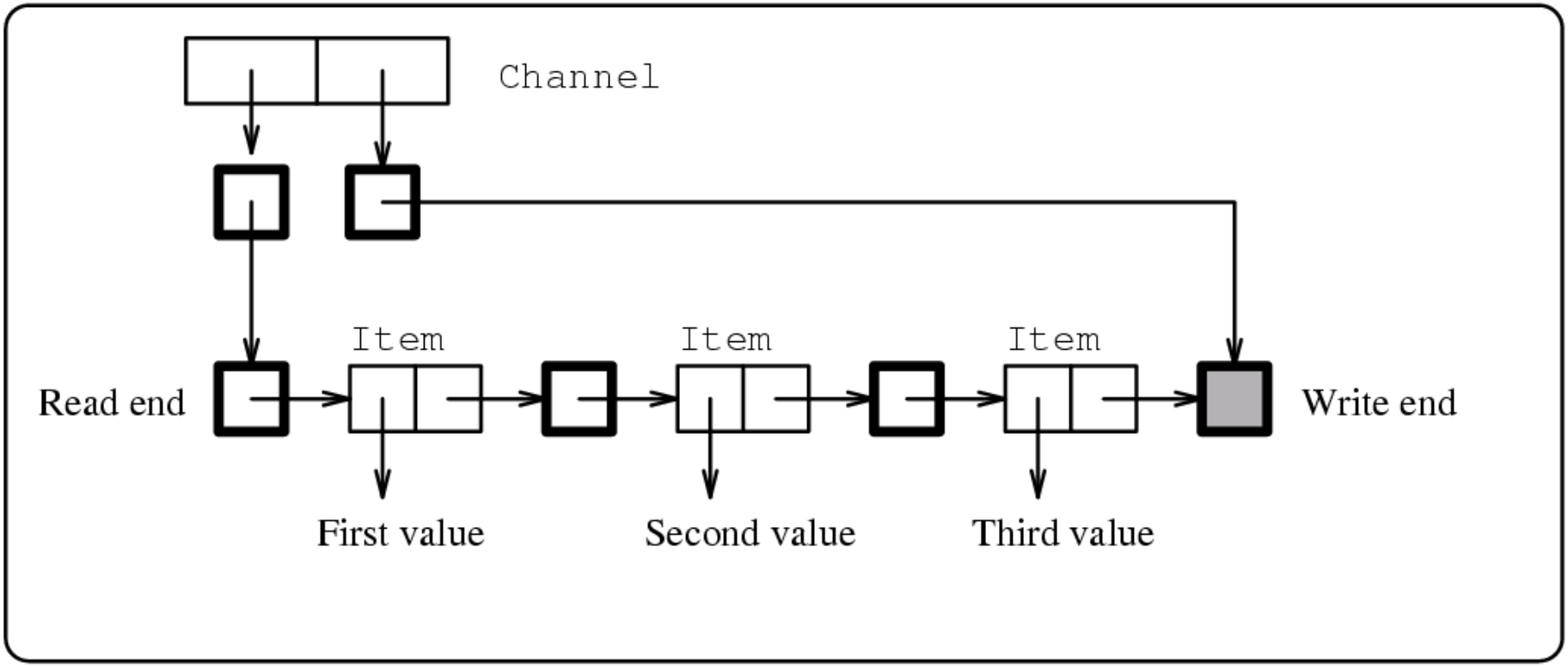
A buffered channel

```
type Chan a
newChan     :: IO (Chan a)
putChan     :: Chan a -> a -> IO
()
getChan     :: Chan a -> IO a
```



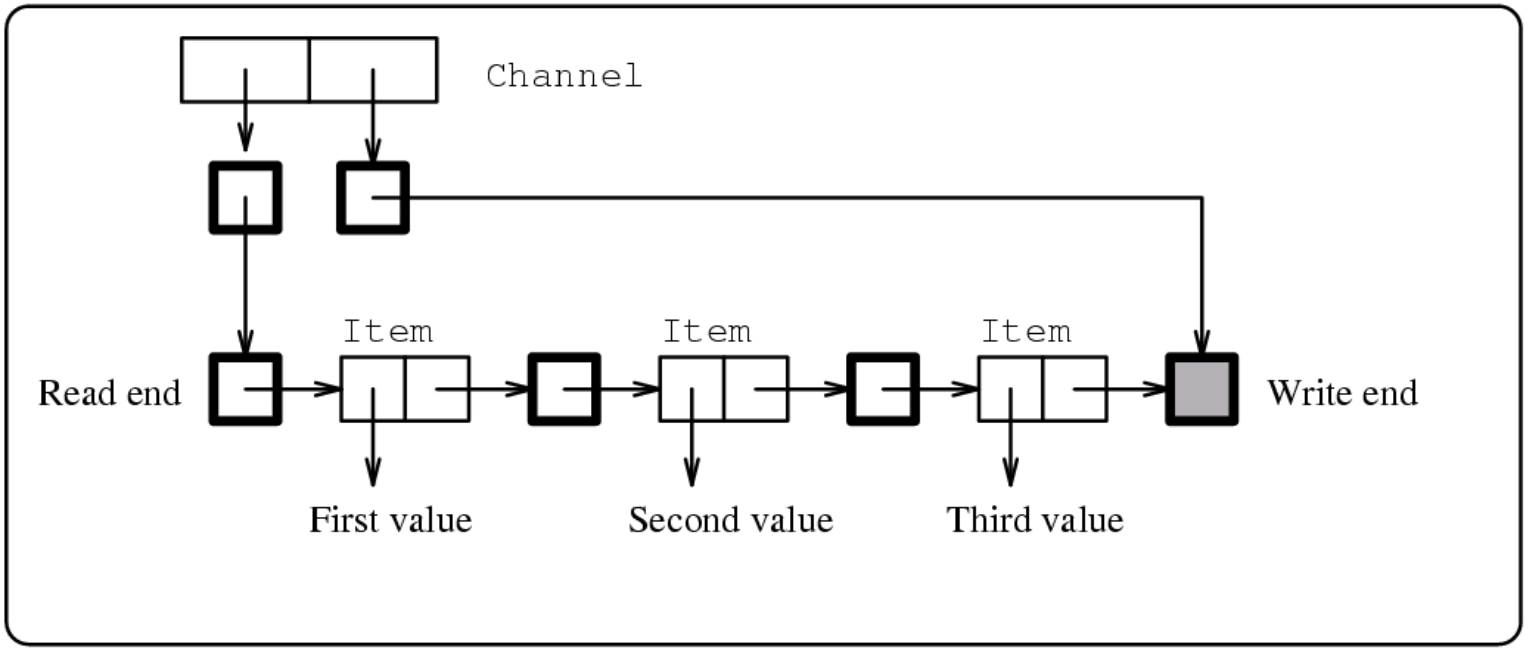
A buffered channel

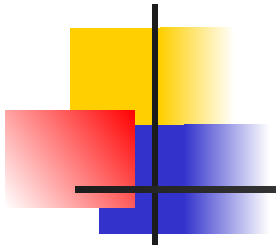
```
type Chan a   = (MVar (Stream a),MVar (Stream a))
type Stream a = MVar (Item a)
data Item a   = MkItem a (Stream a)
```



A buffered channel

```
putChan :: Chan a -> a -> IO ()
putChan (read,write) val
   = do { new_hole <- newEmptyMVar ;
   old_hole <- takeMVar write ;
   putMVar write new_hole
   putMVar old_hole (MkItem val new_hole) }
```

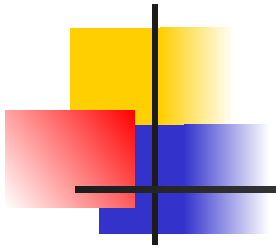




Summary

- forkIO + MVars are very simple.
- MVars are primtive, but surprisingly often Just The Right Thing
- Other excellent references
  - Concurrent programming in ML (Reppy, CUP)
  - Concurrent programming in Erlang (Armstrong, Prentice Hall, 2nd edition)

Exceptions

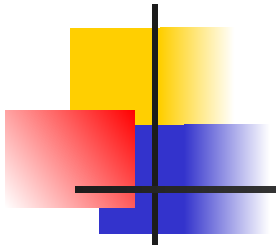


Why do we need exceptions?

Robust programs deal gracefully with "unexpected" conditions. E.g.

- Disk write fails because disk is full
- Client goes away, so server should time out and log an error
- Client requests seldom-used service; bug in server code gives pattern-match failure or divide by zero

Server should not crash if these things happen!



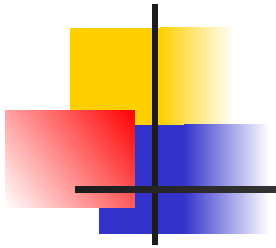
Approach 1: virtue

"A robust program never goes wrong"
(e.g. test for disk full before writing)

BUT:

- Can't test for all errors (e.g. timeouts)
- Some problems are, by definition, errors in the server code

Need a way to recover from ANY error



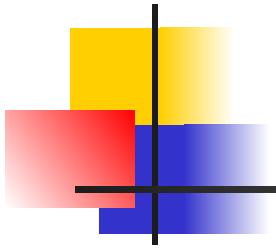
Approach 2: exceptions

Provide a way to say "execute this code, but if anything (at all) goes wrong, abandon it and do this instead".

I call this

"**Exceptions for disaster recovery**"

- Exception handler typically covers a large chunk of code
- Recovery action typically aborts a whole chunk of work



Aside: bad uses of exceptions

Exceptions are often (mis-) used in a different way:

"**Exceptions for extra return values**"

e.g. Look up up something in a table, raise "NotFound" if it's not there.

- Exception handler often encloses a single call
- Recovery action typically does not abort anything

Exceptions in Haskell 98

Haskell 98 supports exceptions in I/O actions:

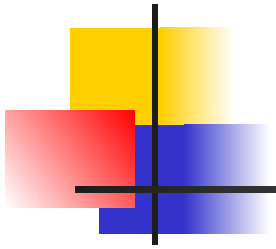
```
catch     :: IO a -> (IOError -> IO a) -> IO a
userError :: String -> IOError
ioError   :: IOError -> IO a
```

```
catch    (do { h <- openFile
"foo";
       processFile h })
   (\e -> putStr "Oh dear")
```

Protected code

Exception handler

Dynamic scope: exceptions raised in processFile are caught

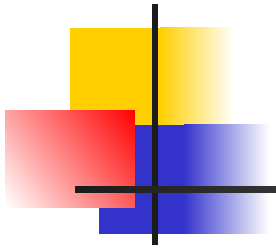


Semantics

Step 1: add a new evaluation context

E ::= [.] | E >>= M | **catch E M**

Says: "evaluate inside the first argument of **`catch`**"



Semantics

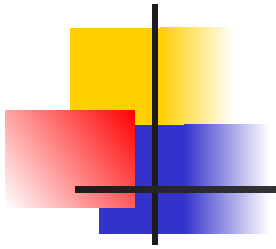
Step 2: add propagation rule for **`ioError`**

$$\{\mathrm{E}[\texttt{ioError}\ e\ \texttt{>>=}\ M]\}_t \quad \rightarrow \quad \{\mathrm{E}[\texttt{ioError}\ e]\}_t$$

An exception before the (>>=)...

...discards the part after the (>>=)

Standard stack-unwinding implementation is possible



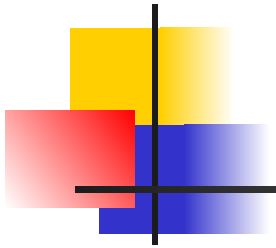
Semantics

What to do if an exception ***is*** raised

Step 3: add rules for `catch`

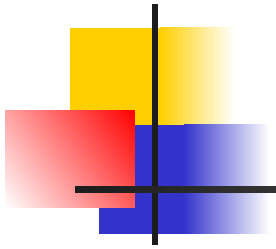
$$\{\mathrm{E}[\texttt{catch}\ (\texttt{ioError}\ e)\ M]\}_t \rightarrow \{\mathrm{E}[M\ e]\}_t$$
$$\{\mathrm{E}[\texttt{catch}\ (\texttt{return}\ N)\ M]\}_t \rightarrow \{\mathrm{E}[\texttt{return}\ N]\}_t$$

What to do if an exception ***is not*** raised



Synchronous vs asynchronous

- A synchronous exception is raised as a direct, causal result of executing a particular piece of code
  - Divide by zero
  - Disk full
- An asynchronous exception comes from "outside" and can arrive at any moment
  - Timeout
  - Stack overflow

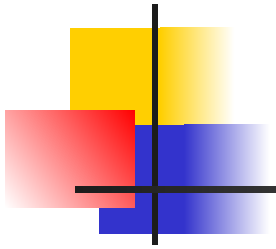


Haskell 98 isn't enough

Haskell 98 deals only with synchronous exceptions in the IO monad

Two big shortcomings

- Does not handle things that go wrong in purely-functional code
- Does not deal with asynchronous exceptions



Exceptions in pure code

Embed exceptions in values

Idea: embed exceptions in values

```
throw :: Exception -> a

divide :: Int -> Int -> Int
divide x y = if y==0 then throw
DivZero
             else x/y
```

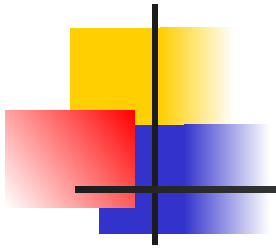
result type unchanged

A value is

- either an "ordinary" value
- or an "exception" value, carrying an exception

(Just like NaNs in IEEE floating point.)

In a lazy language an exception value might hide inside an un-evaluated data structure, but that's OK.



Catching exceptions

New primitive for catching exceptions: BAD BAD!

```
getException :: a -> ExVal a

data ExVal a = OK a
             | Bad Exception
```

Example

```
f x = case getException (goop x) of
        OK result -> result
        Bad exn   -> recovery_goop x
```

A well-known problem

What exception is raised by "+"?

```
(throw ex1) + (throw ex2)
```

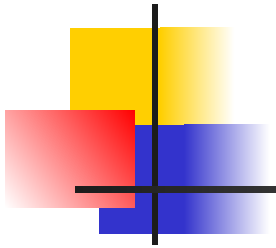
Usual answer: fix evaluation order

BAD ENOUGH for call-by-value languages

- loss of code-motion transformations
- need for effect analyses

TOTAL CATASTROPHE for Haskell

- evaluation order is deliberately unspecified
- key optimisations depend on changing evaluation order



A cunning idea

Return *both* exceptions!

A value is

- either a "normal value"
- or an "exceptional value"

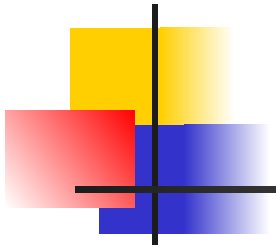
containing a **set** of exceptions

Operationally, an exceptional value is

- represented by a single representative
- implemented by the usual stack-unwinding stuff

c.f. infinite lists:

semantically infinite, operationally finite



Semantics without exceptions

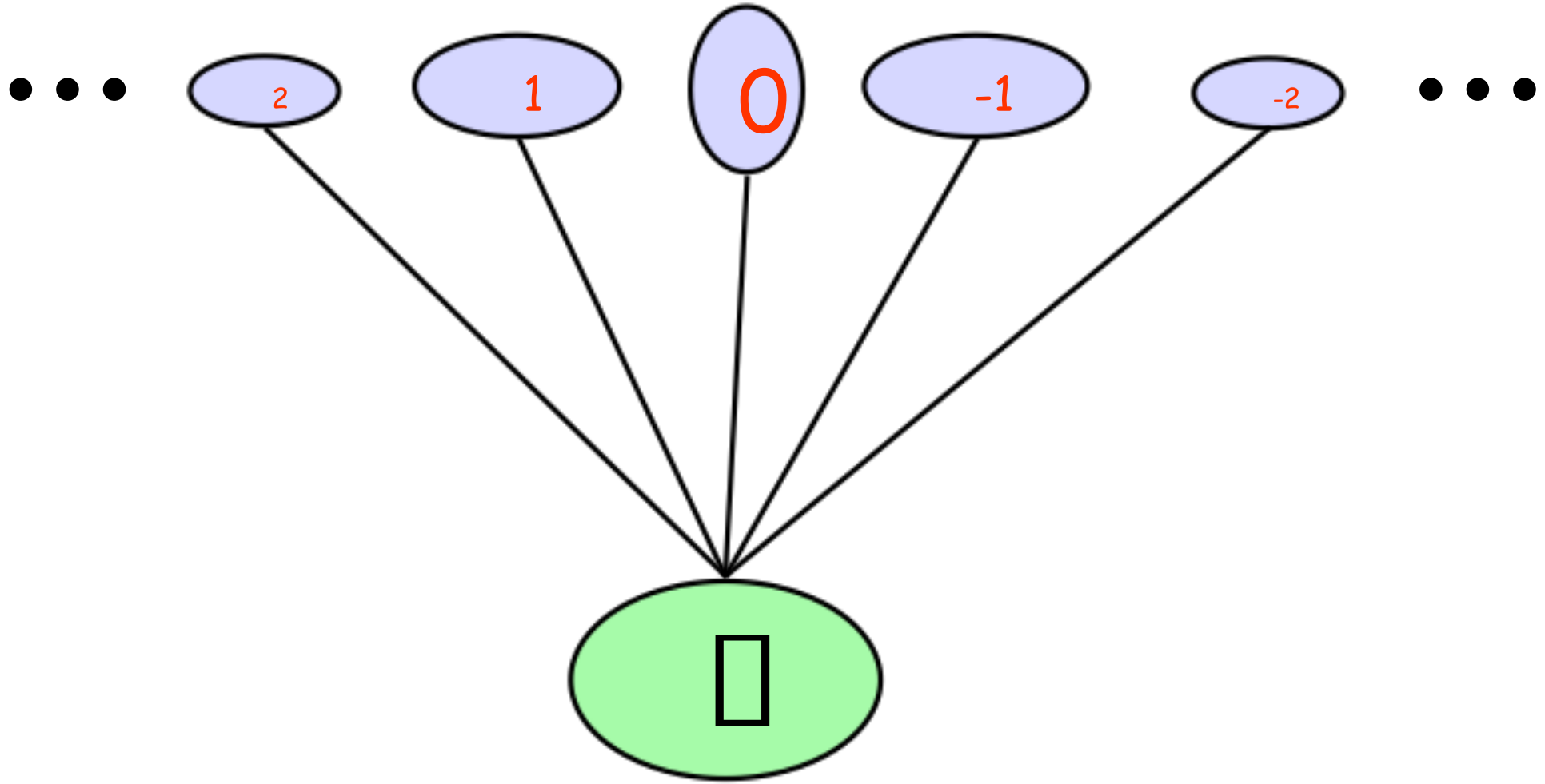
Denotations of Haskell types, ⟦T⟧

⟦Int⟧ ≅ M **Z**
⟦t1→t2⟧ ≅ M (⟦t1⟧ → ⟦t2⟧)
⟦(t1,t2)⟧ ≅ M (⟦t1⟧ × ⟦t2⟧)

M t = t → {⊥}

e.g. ⟦Int->Int⟧ = M (M Int -> M Int)

⟦Int⟧ : Ordinary Haskell



Semantics with exceptions

Denotations of Haskell types, ⟦T⟧

Same as before

⟦Int⟧ = M **Z**
⟦t1→t2⟧ = M (⟦t1⟧ → ⟦t2⟧)
⟦(t1,t2)⟧ = M (⟦t1⟧ × ⟦t2⟧)

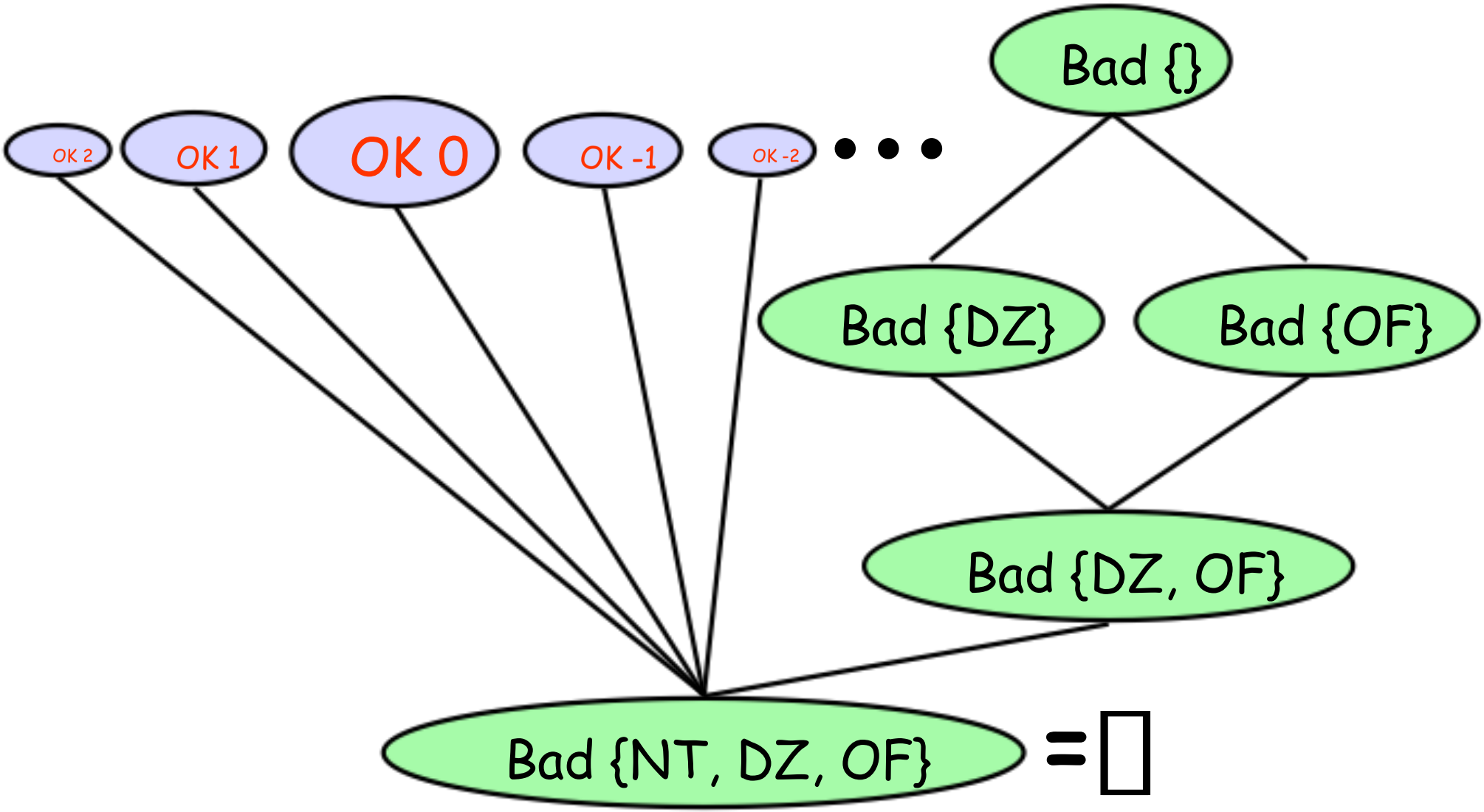
Bottom

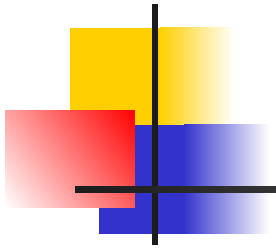
M t = {Ok x | x ∈ t} ∪
{Bad s | s ⊆ **E**} ∪
{Bad (**E** ∪ {NonTermination})}

E = {Overflow, DivZero, …} (fixed set)
Z = the integers

e.g. ⟦Int->Int⟧ = M (M Int -> M Int)

⟦Int⟧ : Exceptional Haskell





Semantics

$[\![e1 + e2]\!]\ \square$ = OK (m + n) if OK m = $[\![e1]\!]\ \square$
OK n = $[\![e2]\!]\ \square$

= Bad (S ($[\![e1]\!]\ \square$) $\square$ S($[\![e2]\!]\ \square$)) otherwise

S(Bad s) = s
S(OK n) = {}

Payoff: $[\![e1 + e2]\!]\ \square = [\![e2 + e1]\!]\ \square$

Whoa! What about getException?

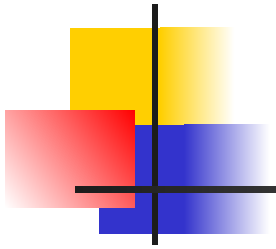
Problem: which exception does getException choose from the set of possibilities?

```
getException :: a -> ExVal a
?????
data ExVal a = OK a| Bad Exception
```

Solution 1: choose any. But that makes getException non-deterministic. And that loses even □-reduction!

```
let x = getException e in x==x          =    True

(getException e) == (getException e)    □    True
```

Verdict: Cure worse than disease.



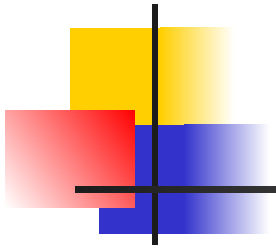
Using the IO monad

Solution 2: put getException in the IO monad:

```
evaluate :: a -> IO a
```

`evaluate` evaluates its argument;

- if it is an ordinary value, it returns it
- if it is an exceptional value, it chooses one of the set of exceptions and raises it as an IO monad exception

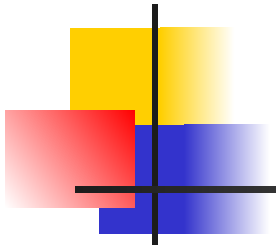


Using the IO monad

Key idea:

The choice of which exception to raise is made in the IO monad, so it can be non-deterministic (like so much else in the IO monad)

```
evaluate :: a -> IO a
```



Using evaluate

```
main = do {     i <- getInput;

   catch (do {     r <- evaluate (goop i);
                   do_good_stuff r })

         (\ ex -> recover_from ex)
       }
```

You have to be in the IO monad to use **evaluate**

You do **not** have to be in the IO monad to use **throw**

Semantics

Add rules for **evaluate**

An ordinary value

$$\frac{\mathcal{E}[\![M]\!] = Ok\ V \quad M \not\equiv V}{\{\mathrm{E}[\texttt{evaluate}\ M]\}_t \rightarrow \{\mathrm{E}[\texttt{return}\ V]\}_t}$$

$$\frac{\mathcal{E}[\![M]\!] = Bad\ S \quad e \in S}{\{\mathrm{E}[\texttt{evaluate}\ M]\}_t \rightarrow \{\mathrm{E}[\texttt{ioError}\ e]\}_t}$$

An exceptional value

Watch out!

$$\frac{\mathcal{E}[\![M]\!] = V \quad M \neq V}{\{\mathrm{E}[M]\} \rightarrow \{\mathrm{E}[V]\}}$$

We've just changed what values look like!

But what if M evaluates to (Bad S)???

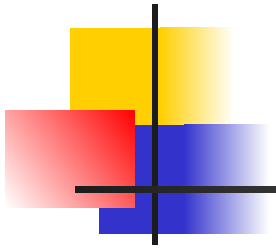
Watch out!

Ordinary value

$$\frac{\mathcal{E}[\![M]\!] = Ok\ V \quad M \not\equiv V}{\{\mathrm{E}[M]\}_t \ \rightarrow\ \{\mathrm{E}[V]\}_t}\ (FUN1)$$

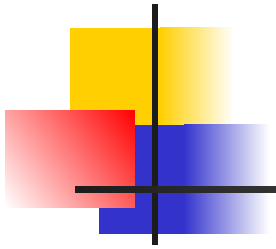
Exceptional value

$$\frac{\mathcal{E}[\![M]\!] = Bad\ S \quad e \in S}{\{\mathrm{E}[M]\}_t \ \rightarrow\ \{\mathrm{E}[\mathtt{ioError}\ e]\}_t}\ (FUN2)$$



Imprecise exceptions

- A decent treatment of exceptions in purely-functional code
- Quite a lot more to say (see PLDI'99 paper)
- No transformations lost!
- Good for disaster recovery, poor for extra return values



Asynchronous exceptions

Asynchronous exceptions

A flexible form of asynchronous exception:

```
throw     :: Exception -> IO a
throwTo     :: ThreadId -> Exception ->
IO a
```

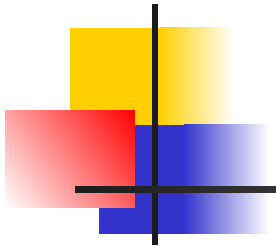
Timeouts

```
timeout :: Int -> IO a -> IO (Maybe a)
timeout n a
 = do { t <- myThreadId ;

   s <- forkIO (do {     sleep n ;
                      throwTo t TimeOut });

   catch (do {     r <- a;
                  throwTo s Kill;
                  return (Just r) });
         (\ex -> Nothing)
   }
```

Fork a thread that sleeps and then throws an exception to its parent

Do the action, and then kill the timeout

The timeout won!



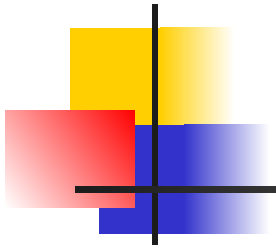
Semantics

Add a rule for **`throwTo`**

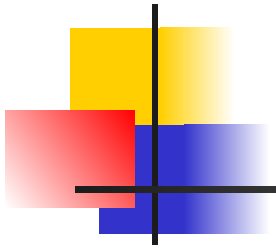
Make sure we replace the **innermost** "current action"

$$\frac{M \neq (N_1 \texttt{>>=} N_2) \qquad M \neq (\texttt{catch}\ N_1\ N_2)}{\{\mathrm{E}_1[\texttt{throwTo}\ t\ e]\}_s \mid \{\mathrm{E}_2[M]\}_t \rightarrow \{\mathrm{E}_1[\texttt{return}\ ()]\}_s \mid \{\mathrm{E}_2[\texttt{ioError}\ e]\}_t}$$

Replace "current action" in target thread with ioError



What have we achieved?



My motivation

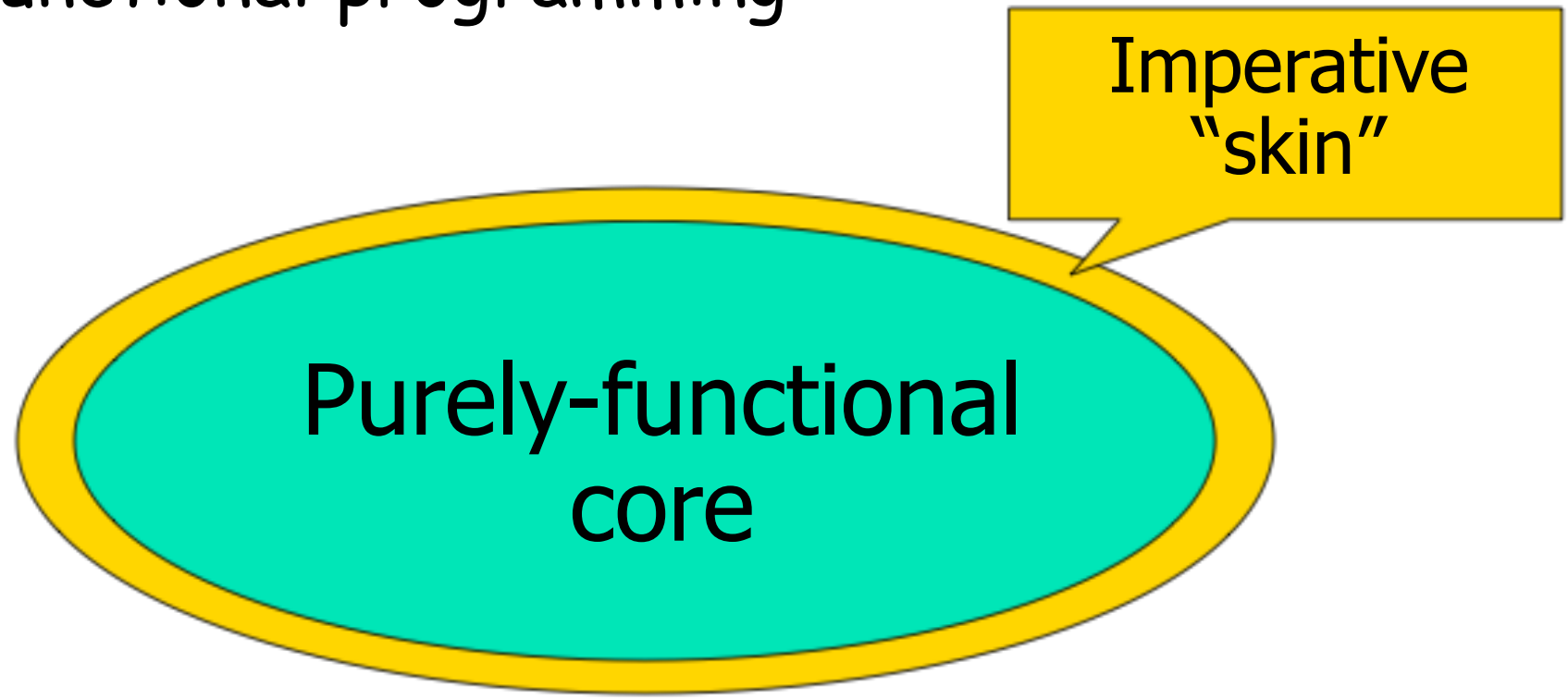
Functional programming is SO much fun.

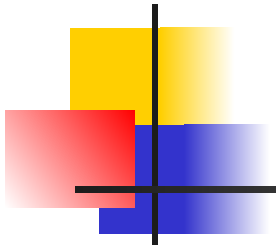
Plan of attack

- Find an application
- Try to write it in Haskell
- Fail
- Figure out how to fix Haskell
- Abstract key ideas, write a paper
- Repeat from (2)

What have we achieved?

- The ability to mix imperative and purely-functional programming





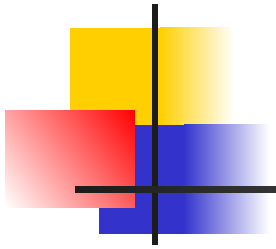
What have we achieved?

- ...without ruining either
- All laws of pure functional programming remain unconditionally true, even of actions

e.g. let x=e in ...x....x...

=

....e....e.....



What we have not achieved

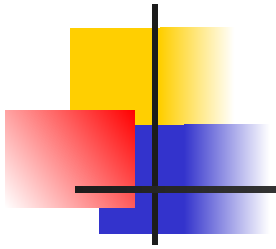
- Imperative programming is no easier than it always was

e.g. `do { ...; x <- f 1; y <- f 2; ...}`

?=?

`do { ...; y <- f 2; x <- f 1; ...}`

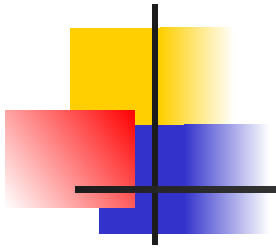
- ...but there's less of it!
- ...and actions are first-class values



Not covered in the lectures

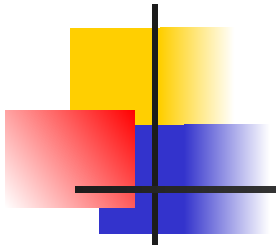
...But in the notes

- Asynchronous exceptions
- Foreign language interfacing



What next?

- Write applications
- Real reasoning about monadic Haskell programs; proving theorems
- Alternative semantic models (trace semantics)
- More refined monads (the IO monad is a giant sin-bin at the moment)



What next?

http://research.microsoft.com/~simonpj
http://haskell.org

Have lots more fun