Lazy functional programming for real

Tackling the Awkward Squad

Adapted by BCP from original slides by

Simon Peyton Jones, Microsoft Research

1

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 Awkward Squad

2

The direct approach

Do everything in "the usual way" (as in ML, Scheme, etc.)

- **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"

But…

3

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

4

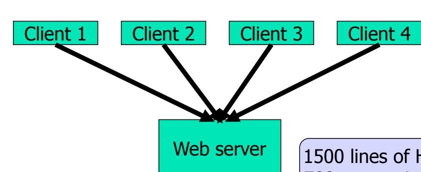
Tackling the awkward squad

- So lazy languages force us to take a different, more principled, approach to the Awkward Squad.
- These lectures and the accompanying notes describe that approach in detail for Haskell.

5

A web server

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



1500 lines of Haskell
700 connections/sec

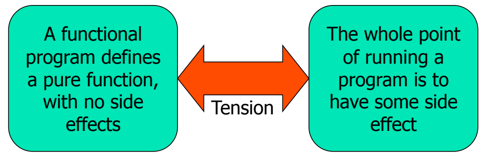
6

Monadic input and output

(review)

7

The problem



8

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

9

Functional I/O is awkward

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

10

Monadic I/O: the key idea

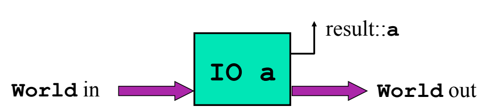
A value of type (`IO t`) is an "action" that, when performed, may do some input/output before delivering a result of type t.

11

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)
```



12

Actions are first class

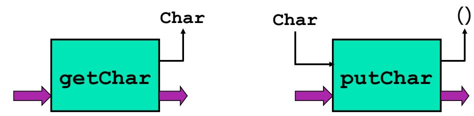
A value of type (`IO t`) is an "action" that, when performed, may do some input/output before delivering a result of type t.

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

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

13

Simple I/O



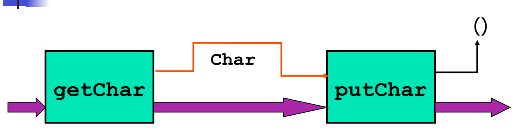

```
getChar :: IO Char
putChar :: Char -> IO ()

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

Main program is an action of type IO ()

14

Connecting actions up

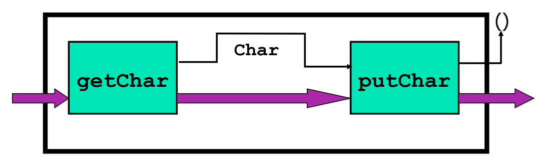


Goal: read a character and then write it back out

15

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
```

16

Printing a character twice

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

This is just noise...

17

The (>>) combinator

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

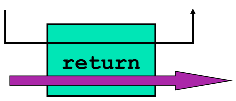
```
(>>) :: IO a -> IO b -> IO b
m >> n  =  m >>= (\_ -> n)
```

18

The return combinator

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

return :: a -> IO a



20

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);
```

21

Notational convenience

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

do notation adds only "syntactic sugar"

22

Desugaring do notation

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

23

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))`

26

First class actions

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

28

What does it all mean?

32

What does "mean" mean?

In linguistics, the structure of natural languages is described and studied at many levels…

Phonetics	What basic sounds (phonemes) are possible in a given language
Morphology	How phonemes fit together to make words
Syntax	How words are arranged into grammatical sentences
Semantics	What these sentences mean

33

What does "mean" mean?

Programming languages can be described in pretty much the same way…

Phonetics	What basic sounds (phonemes) are possible in a given language	Character set (ASCII)
Morphology	How phonemes fit together to make words	Lexing
Syntax	How words are arranged into grammatical sentences	Parsing
Semantics	What these sentences mean	Semantics

34

Semantics of programs

The meaning of programs can be described rigorously (i.e., mathematically) in different ways…

- **Denotational semantics**: The meaning of a program is a mathematical function from inputs to outputs.
- **Operational semantics**: The meaning of a program is the sequence of states that some "abstract machine" goes through when executing it.

35

Denotational Semantics

The meaning of an expression of type Int->Int is a function on the set of integers.

foo x = x*2+1 means foo = { ...,
(-2,-3),
(-1,-1),
(0,1),
(1,3),
(2,5),
... }

36

Denotational Semantics

This gives us a very natural way to talk about *program equivalence*

foo x = x*2+1

means the same as

foo' x = 1+((1+1)*x)

because both mean { ..., (-2,-3), (-1,-1), (0,1),
(1,3), (2,5), ... }

37

Denotational Semantics

The meaning of an expression of type Int->Int is a partial function on the set of integers.

```
fact x =
  if x=0
   then 1
   else x * fact (x-1)
```

means foo = { ...,
(-2,⊥),
(-1,⊥),
(0,1),
(1,1),
(2,2),
(3,6),
... }

pronounced "bottom"

38

Denotational Semantics

So the meaning of (fact -2) is ⊥.

I.e., all non-terminating programs mean the same thing.

Intuitively, this makes good sense...
(All we can "observe" about a non-terminating program is that it doesn't terminate!)

...as long as we are only talking about purely functional expressions.

But...

39

Denotational semantics of IO?

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

- A program that loops forever has meaning ⊥.
  A program that prints 'x' forever also has meaning ⊥!
- What is the meaning of two Haskell programs running in parallel?
- Denotational semantics does not scale well to concurrency, non-determinism, etc.

40

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**

42

Operational semantics

General form:

$$P \xrightarrow{\alpha} Q$$

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

43

Program states

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

Initially, it is just a term, {M}

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

Notation:
Curly braces = "here is a program state"

44

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 from stdin

45

Our first two rules

$$\{\texttt{putChar}\ ch\} \xrightarrow{!ch} \{\texttt{return}\ ()\}$$
$$\{\texttt{getChar}\} \xrightarrow{?ch} \{\texttt{return}\ ch\}$$

Now, what about this?

`{getChar >>= \c-> putChar c}` $\rightarrow$ **???**

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

46

Evaluation contexts

E ::= [.] | 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

47

Revised rules for put/get

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

$$\{\mathrm{E}[\texttt{putChar}\ ch]\} \xrightarrow{!ch} \{\mathrm{E}[\texttt{return}\ ()]\}$$
$$\{\mathrm{E}[\texttt{getChar}]\} \xrightarrow{?ch} \{\mathrm{E}[\texttt{return}\ ch]\}$$

`{getChar >>= \c-> putChar c}`
$\xrightarrow{?ch}$ `{return` ***ch*** `>>= \c-> putChar c}`

48

The return/bind rule

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

"Silent transition" (no IO)

`{getChar >>= \c-> putChar c}`
$\xrightarrow{?ch}$ `{return` ***ch*** `>>= \c-> putChar c}`
$\rightarrow$ `{(\c-> putChar c)` ***ch***`}`

Now we need to do some "ordinary evaluation"

49

The evaluation rule

V is the value of M

M wasn't already evaluated

"Inference rule" notation:
If the things above the line are true, then we can deduce the thing below the line

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

50

The evaluation rule

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

→ {(\c-> putChar c) *ch*}
→ {putChar *ch*}
$\xrightarrow{!ch}$ {return ()}

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

51

Semantics of Mutable State

With these basic tools in-hand, we can think about how to describe the semantics of other members of the "awkward squad."

Let's start with mutable state...

52

Mutable variables in C

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

x is a location, initalised with 3

read x, add 1, store back into x

53

Mutable variables in Haskell

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

x is a location, initialised with 3

read x

add 1, store back into x

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

54

Semantics for variables

Step 1: elaborate the program state

$$\begin{array}{llll} P,Q,R & ::= & \{M\} & \text{The main program} \\ & \mid & \langle M\rangle_r & \text{An } \texttt{IORef} \text{ named } r\text{, holding } M \\ & \mid & P \mid Q & \text{Parallel composition} \\ & \mid & \nu x.P & \text{Restriction} \end{array}$$

e.g. $\nu r,s.\ (\{M\} \mid \langle 3\rangle_r \mid \langle 89\rangle_s)$

Another IORef

Current set of names ("νr,s. ..." is shorthand for "νr. νs. ...")

An IORef named r, holding 3

The main program

55

Semantics for variables

Live demo – evaluation of

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

56

Semantics for variables

Step 2: add rules for reading, writing IORefs

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

57

Semantics for variables

Step 2: add rules for reading, writing IORefs

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

But what if the main program is not "sitting next to" the relevant variable? We might need to rearrange the program state so that the rules above can apply...

58

"Structural rules"

Intuition: A program state is a "soup" consisting of many IORefs and one main program

Step 3: add rules to bring "reagents" together

$$\begin{array}{rcl} P\ |\ Q & \equiv & Q\ |\ P \\ P\ |\ (Q\ |\ R) & \equiv & (P\ |\ Q)\ |\ R \end{array}$$

Stirring the soup

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

Can look under "|"

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

Can perform any transitions that could be performed on a stirred soup

59

Restriction

Step 4: creation of fresh IORef names

$\{E[\texttt{newIORef}\ M]\} \rightarrow \{E[\texttt{return}\ ?]\}\ |\ \langle M\rangle_?$

What can we use as the IORef name???

60

Restriction

Step 4: deal with fresh IORef names

Choose a name r that is not used already

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

Add r to the current set of names

Yield r as the result of newIORef

Put M in a new cell named r

61

More "Structural rules"

Step 5: structural rules for restriction

Can float "ν" outwards (towards the top of the soup)

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

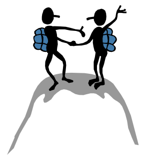
Can look under "ν"

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

62

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 and concurrency (as we will see next!)

63