Lazy functional programming for real

Tackling the Awkward Squad

Adapted by BCP from original slides by 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 Awkward Squad

The direct approach

- Do everything in "the usual way" (as in ML, Scheme, etc.)
  - I/O via "functions" with side effects
  - 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.
- These lectures and the accompanying notes describe that approach in detail for Haskell.

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

Client 1  Client 2  Client 3  Client 4

Web server

1500 lines of Haskell
700 connections/sec
Monadic input and output

The problem

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 direct 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.
Actions are first class

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

\[
\text{type } \text{IO } a = \text{World} \to (a, \text{World})
\]

- "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

Simple I/O

<table>
<thead>
<tr>
<th>Char</th>
<th>getChar</th>
<th>()</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>putChar</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{getChar} : : \text{IO } \text{Char} \\
\text{putChar} : : \text{Char} \to \text{IO } ()
\]

Main program is an action of type \(\text{IO } ()\)

\[
\text{main} : : \text{IO } () \\
\text{main} = \text{putChar } 'x'
\]

Connecting actions up

Goal: read a character and then write it back out

The \((\gg=)\) combinator

\[
(\gg=) : : \text{IO } a \to (a \to \text{IO } b) \to \text{IO } b
\]

\[
\text{echo} : : \text{IO } () \\
\text{echo} = \text{getChar } \gg= \text{putChar}
\]

Printing a character twice

\[
\text{echoDup} : : \text{IO } () \\
\text{echoDup} = \text{getChar } \gg= (\lambda c \to \\
\quad \text{putChar } c \gg= (\lambda () \to \\
\quad \quad \text{putChar } c))
\]

This is just noise...

The \((\gg)\) combinator

\[
(\gg) : : \text{IO } a \to \text{IO } b \to \text{IO } b
\]

\[
\text{echoDup} : : \text{IO } () \\
\text{echoDup} = \text{getChar } \gg \text{putChar } c \gg \text{putChar } c
\]

\[
(\gg) = \text{IO } a \to \text{IO } b \to \text{IO } b
\]

\[
\text{m} \gg \text{n} = \text{m} \gg= (\lambda \_ \to \text{n})
\]
Lazy functional programming for real

The return combinator

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

return :: a -> IO a
```

Notational convenience

- By design, the layout looks imperative
  ```haskell
c1 = getchar();
c2 = getchar();
return (c1,c2);
```

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

  ```haskell
do notation adds only "syntactic sugar"
```

Desugaring do notation

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

Loops

Values of type (IO t) are first class
So we can define our own "control structures"

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

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

First class actions

```
Slogan: first-class actions let us write application-specific control structures
```
What does it all mean?

What does “mean” mean?

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

<table>
<thead>
<tr>
<th>Phonetics</th>
<th>What basic sounds (phonemes) are possible in a given language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morphology</td>
<td>How phonemes fit together to make words</td>
</tr>
<tr>
<td>Syntax</td>
<td>How words are arranged into grammatical sentences</td>
</tr>
<tr>
<td>Semantics</td>
<td>What these sentences mean</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Phonetics</th>
<th>What basic sounds (phonemes) are possible in a given language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morphology</td>
<td>How phonemes fit together to make words</td>
</tr>
<tr>
<td>Syntax</td>
<td>How words are arranged into grammatical sentences</td>
</tr>
<tr>
<td>Semantics</td>
<td>What these sentences mean</td>
</tr>
</tbody>
</table>

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.

Denotational Semantics

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

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

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

```plaintext
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), ... }
```
Denotational Semantics

The meaning of an expression of type \texttt{Int $\rightarrow$ Int} is a partial function on the set of integers.

\[
\text{fact } x = \begin{cases} 
0 & \text{if } x = 0 \\
1 & \text{else if } x = 1 \\
x \ast \text{ fact } (x-1) & \text{else}
\end{cases}
\]

\[
\text{foo } = \{ \ldots, (-2, \perp), (-1, \perp), (0, 1), (1, 1), (2, 2), (3, 6), \ldots \}
\]

pronounced "bottom"

So the meaning of \texttt{(fact -2)} is \(\perp\).

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...

Denotational semantics of IO?

\[
\text{type IO } a = \text{World } \rightarrow (a, \text{World})
\]

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

Operational semantics

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

Equivalence of programs becomes similarity of behaviour instead of identity of meaning.

Program states

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

Initially, it is just a term, \((M)\)

\[
e.g. \{ \text{putChar 'x'} \gg \text{putChar 'y'} \}
\]

Notation:

Curly braces = "here is a program state"
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

Our first two rules

\[
\begin{align*}
\{ \text{putChar } c \} & \xrightarrow{ch} \{ \text{return } () \} \\
\{ \text{getChar} \} & \xrightarrow{?ch} \{ \text{return } ch \}
\end{align*}
\]

Now, what about this?

\( \{ \text{getChar} >>= \lambda c \to \text{putChar } c \} \to ??? \)

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

Evaluation contexts

An evaluation context \( E \) is a term with a "hole" in it.

For example:

\[
\begin{align*}
E_1 &= [.] >>= M \\
E_2 &= ([.] >>= M_1) >>= M_2
\end{align*}
\]

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

\[
\begin{align*}
E_1[\text{getChar}] &= \text{getChar} >>= M \\
E_2[\text{getChar}] &= (\text{getChar} >>= M_1) >>= M_2
\end{align*}
\]

Revised rules for put/get

\[
\begin{align*}
\{ \text{putChar } c \} & \xrightarrow{1ch} \{ \text{return } c \} \\
\{ \text{getChar} \} & \xrightarrow{?ch} \{ \text{return } ch \}
\end{align*}
\]

\[
\begin{align*}
\{ \text{getChar} >>= \lambda c \to \text{putChar } c \} & \xrightarrow{?ch} \{ \text{return } ch >>=} \lambda c \to \text{putChar } c
\end{align*}
\]

The return/bind rule

\[
\begin{align*}
\{ E[\text{return } N >>= M] \} & \to \{ E[M \ N] \}
\end{align*}
\]

\[
\begin{align*}
\{ \text{getChar} >>= \lambda c \to \text{putChar } c \} & \xrightarrow{?ch} \{ \text{return } ch >>=} \lambda c \to \text{putChar } c \\
& \xrightarrow{?ch} \{ \lambda c \to \text{putChar } c \}(ch)
\end{align*}
\]

Now we need to do some "ordinary evaluation"

The evaluation rule

\[
\begin{align*}
E[M] = V \\
E[M] \neq V
\end{align*}
\]

\[
\begin{align*}
E[M] & \to \{ \text{V} \}
\end{align*}
\]
The evaluation rule

\[ \varepsilon[M] = V \quad M \neq V \quad \varepsilon[M] \rightarrow \{ \varepsilon[V] \} \]

- \((\text{c}\rightarrow \text{putChar c}) \quad \text{ch}\)
- \(\text{putChar ch}\)
- \(\text{ich} \rightarrow \{\text{return} ()\}\)

Treat primitive IO actions as “constructors”; so is a value.

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...

Mutable variables in C

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

x is a location, intialised with 3

read x, add 1, store back into x

Mutable variables in Haskell

```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 ()

Semantics for variables

Step 1: elaborate the program state

\[ P, Q, R \ni \{ M \} \quad \text{The main program} \]
\[ \{ M \}_r \quad \text{An IORef named } r, \text{ holding } M \]
\[ P \mid Q \quad \text{Parallel composition} \]
\[ \nu x. P \quad \text{Restriction} \]

e.g. \( \nu r.s. \{ (M) \mid <3> \mid <89> \} \)

Another IORef

Current set of names (“\(r, s \_\_\_\_\)” is shorthand for “\(s, s \_\_\_\_\_\)”)

An IORef named \(r\), holding 3

The main program

Live demo – evaluation of

```haskell
do { x <- newIORef 3;
v <- readIORef x;
writeIORef x (v+1) }
```
Step 2: add rules for reading, writing IORefs

\[
\{E[\text{readIORef } r] | (M), \rightarrow \{E[\text{return } M] | (M)\}, \\
\{E[\text{writeIORef } \nu] | (M), \rightarrow \{E[\text{return } O] | (N)\}\}
\]

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...

“Structural rules”

Step 3: add rules to bring "reagents" together

\[
P \equiv Q \\
P \mid (Q \mid R) \equiv (P \mid Q) \mid R \\
P \equiv P' \quad P' \vdash Q' \quad Q' \equiv Q \\
(P \equiv Q) \mid (Q \equiv Q) \\
P \Rightarrow Q \mid R \\
(P \Rightarrow Q) \mid (Q \Rightarrow R)
\]

Restriction

Step 4: creation of fresh IORef names

\[
\{E[\text{newIORef } M] \leftrightarrow \{E[\text{return } ?] | \text{<M>}\},
\]

What can we use as the IORef name???

Restriction

Step 4: deal with fresh IORef names

Choose a name \( r \) that is not used already

Add \( r \) to the current set of names

Put \( r \) in a new cell named \( r \)

Yield \( r \) as the result of

More “Structural rules”

Step 5: structural rules for restriction

\[
vz. v\gamma.P \equiv v\gamma.vz.P \\
(vz.P) \mid Q \equiv vz.(P \mid Q) \\
vz.P \equiv v\gamma.f(y[z]), \\
x \not\in f(y(Q)) \\
vz.P \Rightarrow vz.Q \quad (NU)
\]

Can float \( \nu \) upwards (towards the top of the soup)

Can look under \( \nu \)
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 and concurrency (as we will see next!)