10/21/08

Announcements

- Midterm: Wed. Oct. 22, 7-9pm, Gates B01
  - Closed book, but you may bring one, letter-sized page of notes, double sided.
  - SCPD students: if you are local, please come to campus to take the exam.
- Homework assigned 10/15 will be ungraded.
  - But we strongly urge you to do it!
  - Solutions will be passed out on 10/20
- Minor corrections to HW3 posted (#3 and 5).
- Reminder: you can work on homework in pairs.

THE IO MONAD

Kathleen Fisher

Reading: “Tackling the Awkward Squad” Sections 1-2
“Real World Haskell,” Chapter 7: I/O

Thanks to Simon Peyton Jones for many of these slides.

Beauty...

Functional programming is beautiful:

- Concise and powerful abstractions
  - Higher-order functions, algebraic data types, parametric polymorphism, principled overloading,
  - Close correspondence with mathematics
  - Semantics of a code function is the math function
  - Equational reasoning if \( x = y \), then \( f(x) = f(y) \)
  - Independence of order-of-evaluation (Church-Rosser)

...and the Beast

- But to be useful as well as beautiful, a language must manage the “Awkward Squad”:
  - Input/Output
  - Imperative update
  - Error recovery
    - (eg. timing out, catching divide by zero, etc.)
  - Foreign-language interfaces
  - Concurrency

The compiler can choose the best order in which to do evaluation, including skipping a term if it is not needed.

The Direct Approach

- Do everything the “usual way”:
  - I/O via “functions” with side effects:
    - `putchar 'x' + putchar 'y'
  - Imperative operations via assignable reference cells:
    ```
    z = ref 0: a := 1e + j:
    f(z): w = !z (* What is the value of w? *)
    ```
  - Error recovery via exceptions
  - Foreign language procedures mapped to “functions”
  - Concurrency via operating system threads
  - Ok if evaluation order is baked into the language.

The Lazy Hair Shirt

In a lazy functional language, like Haskell, the order of evaluation is deliberately undefined, so the “direct approach” will not work.

- Consider: `res = putchar 'x' + putchar 'y'
  - Output depends upon the evaluation order of (+).
- Consider: `ls = [putchar 'x', putchar 'y']
  - Output depends on how the consumer uses the list. If only used in `length ls`, nothing will be printed because `length` does not evaluate elements of list.
Laziness and side effects are **incompatible**.
- Side effects are **important**!
- For a long time, this tension was embarrassing to the lazy functional programming community.
- In early 90’s, a surprising solution (**the monad**)
  emerged from an unlikely source (**category theory**).
- Haskell’s **IO monad** provides a way of tackling the awkward squad: I/O, imperative state, exceptions, foreign functions, & concurrency.

**A Web Server**
- The reading uses a web server as an example.
- Lots of I/O, need for error recovery, need to call external libraries, need for concurrency

![Web server diagram](image_url)

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

**Before Monads**
- **Streams**
  - Program issues a stream of requests to OS, which responds with a stream of inputs.
- **Continuations**
  - User supplies continuations to I/O routines to specify how to process results.
- **World-Passing**
  - The “World” is passed around and updated, like a normal data structure.
  - Not a serious contender because designers didn’t know how to guarantee single-threaded access to the world.
- Stream and Continuation models were discovered to be inter-definable.
- Haskell 1.0 Report adopted Stream model.

**Stream Model: Basic Idea**
- Move side effects outside of functional program
- If Haskell main :: String -> String
  - But what if you need to read more than one file? Or delete files? Or communicate over a socket? ...
Stream Model

- Enrich argument and return type of main to include all input and output events.

  ```haskell
  main :: [Response] -> [Request]
  data Request = ReadFile Filename |
                  WriteFile FileName String |
  data Response = RequestFailed |
                  ReadOk String |
                  WriteOk |
                  Success |
  ```

- Wrapper program interprets requests and adds responses to input.

Example in Stream Model

- Haskell 1.0 program asks user for filename, echoes name, reads file, and prints to standard out.

  ```haskell
  main :: [Response] -> [Request]
  main ~(Success : ~(Str userInput)) = 
  AppendChan stdout "enter filename\n", 
  ReadChan stdin, 
  AppendChan stdout name, 
  ReadFile name, 
  AppendChan stdout (case r4 of 
  Str contents -> contents 
  Failure ioerr -> "can't open file") 
  where (name : _) = lines userInput
  ```

- The `~` denotes a lazy pattern, which is evaluated only when the corresponding identifier is needed.

Stream Model is Awkward!

- Hard to extend: new I/O operations require adding new constructors to Request and Response types and modifying the wrapper.

- No close connection between a Request and corresponding Response, so easy to get “out-of-step,” which can lead to deadlock.

- The style is not composable: no easy way to combine two “main” programs.

- ... and other problems!!!

Monadc I/O: The Key Idea

- A value of type IO t is an “action.” When performed, it may do some input/output before delivering a result of type t.

A Helpful Picture

- A value of type IO t is an “action.” When performed, it may do some input/output before delivering a result of type t.
Actions are First Class

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

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

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

Simple I/O

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

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

Connection Actions

To read a character and then write it back out, we need to connect two actions.

\[
\text{Echo} :: IO () \leftarrow \text{getChar} \circlearrowleft (\text{putChar} \circlearrowleft (\text{getChar}))
\]

The (\(\gg\gg\)) Combinator

- Operator is called **bind** because it binds the result of the left-hand action in the action on the right.
- Performing compound action \(a \gg\gg (\lambda x \to b)\):
  - performs action \(a\), to yield value \(r\)
  - applies function \(\lambda x \to b\) to \(r\)
  - performs the resulting action \(b\{x \leftarrow r\}\)
  - returns the resulting value \(v\)

<table>
<thead>
<tr>
<th>a</th>
<th>(\gg\gg)</th>
<th>b</th>
</tr>
</thead>
</table>

Printing a Character Twice

- The parentheses are optional because lambda abstractions extend "as far to the right as possible."
- The putChar function returns unit, so there is no interesting value to pass on.
The \((\gg)\) Combinator

- The "then" combinator \((\gg)\) does sequencing when there is no value to pass:

\[
(\gg) : \ IO \ a \rightarrow IO \ b \rightarrow IO \ b \\
m \gg n = m \gg (\_ \rightarrow n)
\]

\[
echoDup :: IO ()
\]
\[
echoDup = getChar \gg \_ \rightarrow putChar \ c \gg putChar \ c
\]

\[
echoTwice :: IO ()
\]
\[
echoTwice = echo \gg echo
\]

Getting Two Characters

- We want to return \((c1, c2)\).
- But, \((c1, c2) :: (Char, Char)\)
- And we need to return something of type \(IO(Char, Char)\)
- We need to have some way to convert values of "plain" type into the I/O Monad.

\[
getTwoChars :: IO (Char, Char)
\]
\[
getTwoChars = getChar \gg \_ \rightarrow getChar \gg \_ \rightarrow return (c1, c2)
\]

The \textit{return} Combinator

- The action \((\text{return } v)\) does no IO and immediately returns \(v\):

\[
\text{return} :: a \rightarrow IO \ a
\]

\[
\text{return}
\]

\[
getTwoChars :: IO (Char, Char)
\]
\[
getTwoChars = getChar \gg \_ \rightarrow getChar \gg \_ \rightarrow return (c1, c2)
\]

The "do" Notation

- The "do" notation adds syntactic sugar to make monadic code easier to read.

\[
\text{-- Plain Syntax}
\]
\[
getTwoChars :: IO (Char, Char)
\]
\[
getTwoChars = getChar \gg \_ \rightarrow getChar \gg \_ \rightarrow return (c1, c2)
\]

\[
\text{-- Do Notation}
\]
\[
getTwoCharsDo :: IO (Char, Char)
\]
\[
getTwoCharsDo = do {c1 <- getChar ; c2 <- getChar ; return (c1, c2)}
\]

- Do syntax designed to look imperative.

Desugaring "do" Notation

- The "do" notation 	extit{only} adds syntactic sugar:

\[
\text{do} \{ x \leftarrow e; \ es \} = e \gg \_ \rightarrow \text{do} \{ \ es \}
\]
\[
\text{do} \{ e; \ es \} = e \gg \text{do} \{ \ es \}
\]
\[
\text{do} \{ e \} = e
\]
\[
\text{do} \{ \text{let} \ ds; \ es \} = \text{let} \ ds \text{ in} \ \text{do} \{ \es \}
\]

- The scope of variables bound in a generator is the rest of the "do" expression.

- The last item in a "do" expression must be an expression.

Syntactic Variations

- The following are equivalent:

\[
\text{do} \{ x1 \leftarrow pl; \ldots; xn \leftarrow pn; q \}
\]
\[
\text{do} x1 \leftarrow pl; \ldots; xn \leftarrow pn; q
\]

\[
\text{do} x1 \leftarrow pl; \ldots; xn \leftarrow pn; q
\]

If the semicolons are omitted, then the generators must line up. The indentation replaces the punctuation.
Bigger Example

- The `getline` function reads a line of input:

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

Note the "regular" code mixed with the monadic operations and the nested "do" expression.

An Analogy: Monad as Assembly Line

- Each action in the IO monad is a possible stage in an assembly line.
- For an action with type `IO a`, the type
  - tags the action as suitable for the IO assembly line via the `IO` type constructor.
  - indicates that the kind of thing being passed to the next stage in the assembly line has type `a`.
- The `bind` operator "snaps" two stages `s1` and `s2` together to build a compound stage.
- The `return` operator converts a pure value into a stage in the assembly line.
- The assembly line does nothing until it is turned on.
- The only safe way to "run" an IO assembly is to execute the program, either using ghci or running an executable.

Powering the Assembly Line

- Running the program turns on the IO assembly line.
- The assembly line gets "the world" as its input and delivers a result and a modified world.
- The types guarantee that the world flows in a single thread through the assembly line.

```
ghci or compiled program
Result
```

Control Structures

- Values of type `IO t` are first class, so we can define our own control structures.

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

Example use:

```
Main> repeatN 5 (putChar 'h')
```

For 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
```

Example use:

```
Main> for [1..10] (\x -> putStrLn (show x))
```

Sequencing

- A list of IO actions returning a list.

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

Example use:

```
Main> sequence [getChar, getChar, getChar]
```
First Class Actions

**Slogan:** First-class actions let programmers write application-specific control structures.

IO Provides Access to Files

- The IO Monad provides a large collection of operations for interacting with the "World."
- For example, it provides a direct analogy to the Standard C library functions for files:
  ```haskell
  openFile :: String -> IOMode -> IO Handle
  hPutStr :: Handle -> String -> IO ()
  hGetLine :: Handle -> IO String
  hClose :: Handle -> IO ()
  ```

References

- The IO operations let us write programs that do I/O in a strictly sequential, imperative fashion.
- **Idea:** We can leverage the sequential nature of the IO monad to do other imperative things!
  ```haskell
  data IORef a = IORef a
  instance Monad IORef where
    return x = IORef x
    (IORef x) >>= f = f x
  ``
  ```haskell
  newIORef :: a -> IO (IORef a)
  readIORef :: IORef a -> IO a
  writeIORef :: IORef a -> a -> IO ()
  ```
- A value of type `IORef a` is a reference to a mutable cell holding a value of type `a`.

Example Using References

```haskell
import Data.IORef -- import reference functions
-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
  r <- newIORef 0;
  loop r 1
where
  loop :: IORef Int -> Int -> IO Int
  loop r i | i > n = readIORef r
  | otherwise = do
    v <- readIORef r;
    writeIORef r (v + i);
    loop r (i+1)
```

But this is terrible! Contrast with: sum [1..n]. Claims to need side effects, but doesn’t really.

A Second Example

- Track the number of chars written to a file.

```haskell
import Data.IORef -- import reference functions
-- Compute the sum of the first n integers
count :: Int -> IO Int
count n = do
  r <- newIORef 0;
  loop r 1
where
  loop :: IORef Int -> Int -> IO Int
  loop r i | i > n = readIORef r
  | otherwise = do
    v <- readIORef r;
    writeIORef r (v + length cs);
    hPutStr h cs
```

Just because you can write C code in Haskell, doesn’t mean you should!
All operations return an IO action, but only bind (>>=) takes one as an argument.

Bind is the only operation that combines IO actions, which forces sequentiality.

Within the program, there is no way out!

return :: a -> IO a
(>>=) :: IO a -> (a -> IO b) -> IO b
getChar :: IO Char
putChar :: Char -> IO ()
... more operations on characters ...
openFile :: [Char] -> IOMode -> IO Handle
... more operations on files ...
newIORef :: a -> IO (IORef a)
... more operations on references ...

Suppose you wanted to read a configuration file at the beginning of your program:

The problem is that readFile returns an IO String, not a String.

Option 1: Write entire program in IO monad.
But then we lose the simplicity of pure code.

Option 2: Escape from the IO Monad using a function from IO String -> String.
But this is the very thing that is disallowed!

configFileContents :: [String]
configFileContents = lines (readFile "config")

This situation arises sufficiently often that Haskell implementations offer one last unsafe I/O primitive: unsafePerformIO.

GHC uses world-passing semantics for the IO monad:

 unsafePerformIO :: IO a -> a
 unsafePerformIO = \\
 Invent World \act \rightarrow \Dissard World \Result

As its name suggests, unsafePerformIO breaks the soundness of the type system.

z :: IORef c -- This is bad!
z = unsafePerformIO (newIORef (error "urk"))
cast :: a -> b
cast x = unsafePerformIO (do { writeIORef r x; readIORef r })

So claims that Haskell is type safe only apply to programs that don’t use unsafePerformIO.

Similar examples are what caused difficulties in integrating references with Hindley/Milner type inference in ML.

The operator has a deliberately long name to discourage its use.
Its use comes with a proof obligation: a promise to the compiler that the timing of this operation relative to all other operations doesn’t matter.

Implementation

GHC uses world-passing semantics for the IO monad:

 unsafePerformIO :: IO a -> a
 unsafePerformIO = \\
 Invent World \act \rightarrow \Dissard World \Result

It represents the "world" by an unforgeable token of type World, and implements bind and return as:

 unsafePerformIO (return a) = a
 unsafePerformIO (a >>= k) = \\
 return \a \rightarrow IO a \rightarrow (a \rightarrow IO b) \rightarrow IO b
 unsafePerformIO (a >>= k) = \\
 return \a \rightarrow IO a \rightarrow (a \rightarrow IO b) \rightarrow IO b
(>>=) a k x w = \case m w of (r,w') \rightarrow x r w'

Using this form, the compiler can do its normal optimizations. The dependence on the world ensures the resulting code will still be single-threaded.

The code generator then converts the code to modify the world "in-place."
Monads

- What makes the IO Monad a Monad?
- A monad consists of:
  - A type constructor M
  - A function `bind :: M a -> (a -> M b) -> M b`
  - A function `return :: a -> M a`
- Plus:
  - Laws about how these operations interact.

Monad Laws

```haskell
return x >>= f = f x
m >>= return = m
m1 >>= (λx.m2 >>= (λy.m3)) =
  (m1 >>= (λx.m2)) >>= (λy.m3)
x not in free vars of m3
```

Derived Laws for ` >>= ` and `done`

```haskell
(m >> n) = m >>= (
  \_ -> n)
done :: IO ()
done = return ()
```

Reasoning

- Using the monad laws and equational reasoning, we can prove program properties.

Proposition:

```
putStr :: String -> IO ()
putStr [] = done
putStr (c:cs) = putChar c >> putStr cs
```

Proposition:

```
putStr r >> putStr s = putStr (r ++ s)
```

Proof: By induction on r.

Base case: r is []

```
putStr [] >> putStr s
  = (definition of putStr)
  done >> putStr s
  = (first monad law for `>>`)
  putStr s
  = (definition of `++`)
  putStr ([] ++ s)
Immediate: r is (c:cs)
```

Summary

- A complete Haskell program is a single IO action called `main`. Inside IO, code is single-threaded.
- Big IO actions are built by gluing together smaller ones with `bind (>>=)` and by converting pure code into actions with `return`.
- IO actions are first-class.
  - They can be passed to functions, returned from functions, and stored in data structures.
  - So it is easy to define new “glue” combinators.
- The IO Monad allows Haskell to be pure while efficiently supporting side effects.
- The type system separates the pure from the effectful code.
A Monadic Skin

- In languages like ML or Java, the fact that the language is in the IO monad is baked in to the language. There is no need to mark anything in the type system because it is everywhere.
- In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.
- So it is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.

Monads

- So far, we have only seen one monad, but there are many more!
- We’ll see a bunch more of them on Wednesday.