Lecture 8 CSE P505 Autumn 2016 Dan Grossman

Acknowledgments

• Slide-and-code content *liberally* appropriated with permission from Kathleen Fisher, Tufts University

• She in turn acknowledges Simon Peyton Jones, Microsoft Research, Cambridge “for many of these slides”

• And then I probably introduced errors and weaknesses as I changed them…

References

• “Real World Haskell”,
  – Particularly Chapters 0 & 7

• “Tackling the Awkward Squad”
  – Particularly Sections 1 & 2

Haskell

• Haskell is a programming language that is:
  – Similar to ML: general-purpose, strongly typed, higher-order, functional, supports type inference, …
  – Different from ML: purely functional core, lazy evaluation, monadic IO, type classes, …
  – These differences are why we will use it for Homework 5 and what we will focus on

• Designed by committee in 1980s and 1990s to unify research efforts in lazy languages. Continues to evolve.
  – Haskell 1.0 in 1990, Haskell ’98, Haskell ongoing.
  – “A History of Haskell: Being Lazy with Class” HOPL 3

Successful Research Languages

These “graphs” aren’t mine and aren’t based on real data, but they’re fun [and make a meta-point ?]
Committee languages

C++, Java, Perl, Ruby

Haskell

Function types mean more

Thanks to purity, a function type is a stronger spec in Haskell:

- If \( f :: A \rightarrow B \), then for every \( e :: A \), we know \( f e \)
  - \( Equals \) some \( v :: B \), or
  - Does not terminate [hand-wave exceptions, …]

- If \( e1 = e2 \), then \( f e1 = f e2 \)
  - A “bigger deal than it looks” — “no side effects or implicit state”
  - \( \text{let } x = f e \text{ in } (x,x) \) is indistinguishable from \( (f e, f e) \)

Syntax differences from OCaml

- \( x :: \text{Int} \) means “\( x \) has type \( \text{Int} \)”
- \( y : \text{ys} \) means “cons \( y \) onto list \( \text{ys} \)”
- \( \lambda x \rightarrow x + 1 \) “\( \lambda \)” means lambda
- Required upper/lowercase:
  - Expression identifiers are lowercase
  - Type constructors (names) are uppercase
  - Type variables are lower case (and no ’)
- Comments:
  - \(--\) to end of line
  - \{– … –\}
- At top-level no “let” for bindings
- In other scopes, let or where with latter common
- Whitespace relevant (no | on case branches, …)
List comprehensions

• “Not a big deal” but convenient syntax for maps, filters, and zips
  – Could “desugar”

```haskell
myData = [1,2,3,4,5,6,7]
twiceData = [2 * x | x <- myData]
-- [2,4,6,8,10,12,14]
twiceEvenData = [2 * x| x <- myData, x `mod` 2 == 0]
-- [4,8,12]
crossProductDataEvens = [(i,j)| i <- myData, j <- myData, (i+j) `mod` 2 == 0]
-- [(1,1),(1,3),(1,5),(1,7),(2,2),(2,4),…]
```

Laziness

• Haskell is a lazy language

• Functions (and data constructors) do not evaluate their arguments until they need them
  – Then “store the result” to avoid re-execution
  – By default this happens “everywhere”

• Theoretical “best approach” in pure language
  – Humans struggle to determine “when evaluation happens”
  – But thanks to purity it doesn’t matter (!)
  – And laziness is powerful for “infinite data structures”

If OCaml vs. Haskell

```haskell
if’ :: Bool -> a -> a -> a
if’ b e1 e2 = case b of True -> e1 | False -> e2

(* WRONG: always evaluates e1 and e2 *)
let if’ b e1 e2 = match b with true -> e1 |
| false -> e2

(* RIGHT but no memoization (fine here) and caller
must thunk *)
let if’ b e1 e2 = match b with true -> e1 () |
| false -> e2 ()

(* using Lazy library (but avoiding special syntax)
and caller must thunk and use Lazy.from_fun *)
let if’ b e1 e2 = match b with true -> Lazy.force e1 |
| false -> Lazy.force e2
```

Implementing OCaml lazy

• Lazy module no big deal:

```haskell
type 'a t1 = Done of 'a | NotDone of unit -> 'a
type 'a t = 'a t1 ref (* export abstractly *)
let from_fun f = ref (NotDone f)
let force p = match !p with
  Done v -> v
  NotDone f -> p := Done (f());
  force p
```

• The point is this is the semantics in Haskell for every function
call and data argument (forced only when its known that “result
of program” needs it)

Examples

```haskell
loop x = loop x
xs = 3+2 : loop 7 : 1+4 : []
x1 = head xs
x2 = (head (tail xs))
x3 = (head (tail (tail xs)))
three = length xs
prefix_sums acc ys =
  case ys of
    [] -> []
    y : ys -> (acc+y) : prefix_sums (acc+y) ys
five = head (prefix_sums 0 xs)
main :: IO a
  print x1; print x3; print three; print five
  -- ; print x2
```

Lazy programming

• Do not worry about creating (thunks that create) large, even
  infinite data structures
  – Then use only what you need

• Example: streams

  ```haskell
  ones = 1 : ones
  nats = prefix_sums 0 ones
  a_few = tail (take 7 nats)
  ```

• Example: search problems [not shown]
  – “Natural” separation between “generator” of [potentially-
    infinite] moves and “consumer” (search strategy)
Back to purity

• Pure functions are easy to test – “no side effects”

• Example: If \( xs = reverse(reverse \; xs) \), then you can replace one with the other with high confidence

• And testing this property cannot depend on any state because if \( reverse \) is pure (and everything in “core Haskell” is pure), then it cannot depend on that state

Purity is beautiful

• Like in OCaml:
  higher-order functions, algebraic data types, parametric polymorphism, ...

• Plus equational reasoning due to “no side effects” and “only needed computations evaluated”
  – If \( x = y \), then \( f \; x = f \; y \)
  – Order of evaluation is irrelevant, so don’t have to “think about it being lazy” except for termination/performance

... and the beast

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

> The whole point of a running a program is to affect the real world, an “update in place” of something

Direct approach

• Could allow side effects “the usual way” and discourage them
  – Example: \( \text{putchar} :: \text{Char} \rightarrow () \)
  – And similar for references, exceptions, ffi, concurrency

• In practice, this works fine in an eager language (cf. OCaml) but is unworkable in a lazy language
  – Makes evaluation order relevant again
  – And laziness is hard to reason about
  – And compiler wants freedom to optimize away laziness when it can tell “it won’t matter”

• This also doesn’t work at the semantics level if we define our language to have “undefined evaluation order” rather than lazy
  – As Haskell does…

Examples

Evaluation order of function arguments and data constructor arguments does not matter (and isn’t defined) when functions are pure.

Example:
\[
(\lambda x \ y. \ y) \, (\text{putchar} \, 'w') \, (\text{putchar} \, 'x', \ \text{putchar} \, 'y')
\]

With lazy implementation output still depends on how result is used

By the way:
- What about exceptions?
- Non-deterministic evaluation order “so any exception might happen” works okay in practice
- Example: \( y = [3 \ \text{`div'} 0, \ \text{head} \ (\text{tail} [4])] \)

Tackling the “Awkward Squad”

• Laziness and side effects are incompatible

• Side effects are important!

• For a long time, this tension was embarrassing to the lazy functional programming community
  – [will skip earlier solutions that “worked okay for I/O in terms of lazy streams”]

• 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.
Monadic I/O: The Key Idea

- \( \text{IO} \) is a type constructor
  - \( \text{IO} \ t \) is a type where \( t \) is a type

- Think of \( \text{IO} \ t \) as describing an “action” or “computation” that when performed produces a result of type \( t \)

- Now manipulate values of type \( \text{IO} \ t \) in your pure lazy language
  - Pass them around, combine them, etc.
    - With helpful functions and sugar
    - But cannot “do an IO action” inside a program
  - Only \( \text{main} :: \text{IO} \ a \), can be “performed”
    - By “running the program”

A helpful picture

- \( \text{IO} \) is an abstract type constructor, but think of it as:
  
  \[
  \text{type IO} \ t = \text{World} \rightarrow (t, \text{World})
  \]

- “An action” that, when performed, takes “a world” and returns “a \( t \) and a [new] world”

- Thanks to abstraction, there is no way to “get a world”, so you can’t “store or copy a world” (woah!!)

Actions are first class

- Evaluating an \( \text{IO} \ t \) produces an action
  - Evaluation has no side effects
  - Does not perform-the-action, which [probably] has side effects

Simple I/O

- \( \text{getChar} :: \text{IO} \ \text{Char} \)
- \( \text{putChar} :: \text{Char} \rightarrow \text{IO} \ () \)
- \( \text{main} :: \text{IO} () \)
  
  \( \text{main} = \text{putChar} \ ‘x’ \)

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

Connection actions

- To read a character and then write it back out, we need to connect two actions
  
  ![Diagram](diagram.png)

- This is done with the bind combinator…

Bind

- “Provided” (as are \( \text{getChar} \) and \( \text{putChar} \) are)
  
  \[
  (\gg=) :: \text{IO} \ a \rightarrow (a \rightarrow \text{IO} \ b) \rightarrow \text{IO} \ b
  \]

- Semantics is exactly “the compound sequenced action” you would expect from the type
More on `>>=`

- Called bind because it binds the result of the left-hand action in the action on the right.
- The result of calling `>>=` is an action that, when performed:
  - Performs the action on the left, producing result `r1`.
  - Applies the function on the right to `r1` to get another action.
  - Applies that action, to get another result `r2`.
  - Returns `r2`.

```
  e1 >>= \x -> e2
```

Printing a character twice

- Parentheses are optional for usual lambda-concrete-syntax reasons.
- "Do notation" is syntactic sugar for exactly the same thing.
  - Designed to "look imperative"; will extend it soon.
  - It’s just sugar for creating actions with bind, not performing them.

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

More sugar / helper functions

- The "then" combinator sequences actions when there is no value to pass forward.

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

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

The return combinator

- [I won’t try to justify the name “return” – it’s not what you think even though it sorta kinda sounds right].
- The “action” `return v` just produces result `v` (no side effects).

```
  return :: a -> IO a
```

Getting Two Characters

- `(c1,c2) :: (Char, Char)` but we need the `????` to be replaced with something of type `IO (Char, Char)`.
- Need a way to convert “plain” values into IO actions.
  - Should be fine: “performing the action in a world” is just “evaluate the expression” [ignoring the world].

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

Yet more sugar

- Can omit braces for do-notation.
- Can use indentation instead of semicolons.
- … some more.
- But the simple stuff is “just”:
  - `x <- e1; e2 for e1 >>= \x. e2`
  - `e1; e2 for e1 >> e2`
  - `return e` [not necessarily just at end because it’s not the “return” you are used to].
Bigger Example

• [Of course in practice, you would provide this as a faster primitive]
• Key points:
  – Recursion as usual 😊
  – “Mixing in” regular code that produces actions

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

A helpful picture [again]

• IO is an abstract type constructor, but think of it as:
  ```haskell
type IO t = World -> (t, World)
```
  – “An action” that, when performed, takes “a world” and returns “a t and a [new] world”
  ```mermaid
  digraph {
    IO t -> result :: t
  }
  ```
  – Thanks to abstraction, there is no way to “get a world”, so you can’t “store or copy a world” (woah!!)
  – Enforces “single path” through a sequence of actions

Control Structures

• More examples showing how “first-class actions” can be composed to build your own control structures
  – Think: treating code [actions] as data and building up compound data that can later be ‘run’

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

• Example use:
  ```haskell
  repeatN 5 (putChar '#')
  ```

More first-class fun

• Showing general idea of “first-class actions” lets the programmer define structures of [arbitrary] actions
  – No need to bake more than >>= and return into the language

```haskell
sequence :: [IO a] -> IO [a]
sequence xs = case xs of
  [] -> return []
  y:ys = do { r <- y;
             rs <- sequence ys;
             return (r:rs) }
```

• Example use:
  ```haskell
  sequence [getRow, putChar '>' >> return [], getRow]
  ```

Growing the IO monad

• The IO monad is “built-in” to Haskell via main :: IO()
• It is “one-stop shopping” for “all the stuff that needs a well-defined sequence when performed”
  – a.k.a. “the sin bin” combined with “the outside world”
• Just a flavor:

```haskell
openFile :: FilePath -> IOMode -> IO Handle
hPutStr :: Handle -> String -> IO ()
hGetLine :: Handle -> IO String
hClose :: Handle -> IO ()
data IORef a = -- Abstract type
newIORef :: a -> IO (IORef a)
readIORef :: IORef a -> IO a
writeIORef :: IORef a -> a -> IO ()
```

So we have an imperative language

So now you could write this

```haskell
count :: (a -> Bool) -> [a] -> IO Int
count f xs = do { r <- newIORef 0; help r xs }
  where
    help r xs = case xs of
      [] -> readIORef r
      | x:xs -> if f x
        then do { old <- readIORef r;
                  writeIORef r (old+1);
                  help r xs }
        else help r xs
```
But…

• Just because you can write imperative code doesn’t mean you should

```haskell
count :: (a -> Bool) -> [a] -> Int
count f xs =
  case xs of
    [] -> 0
    x:xs -> (if f x then 1 else 0)
        + count f xs
```

The Roach Motel 😊

• “Once you get in to the IO monad, you can’t get out”
  – Bind lets you use a value “in there” but “leaves you in there”
  – Return “gets anything you want in there”

• So you find yourself “wanting to cheat”, looking for a
  `magic_escape :: IO a -> a`

• The presence of such a function would “break everything”
  because it would have to “perform the action” [no other way it could find an `a`, but then we have side effects in allegedly pure code, which was the whole thing we were trying to avoid]

Examples with this problem

• Suppose you want to read some configuration options from a file but treat the values as “pure constants”

```haskell
configFileContents :: [String]
configFileContents = lines (readFile "config") -- NO!
useOptimization :: Bool
useOptimization = elem "optimize" configFileContents
```

• This doesn’t and shouldn’t type-check:
  `readFile :: String -> IO String`

• Leaves only two options:
  – Put all code depending on file contents in IO monad
  – Cheat

The Cheat Exists (!)

• They call it `unsafePerformIO not magic_escape`

```haskell
magic_escape :: IO a -> a
```

• Any code that uses it has an obligation to “know” that “it doesn’t matter”
  – When we perform the IO action
  – How many times we perform the IO action
  – Relative order of performing this action vs. other actions

• The operator has a deliberately long to discourage its use

BTW, It really is a cheat

• You can use `unsafePerformIO` to circumvent the type system arbitrarily
  – Exact same issue arises in OCaml without “the value restriction”
  – OCaml has to avoid this; Haskell can say “unsafePerformIO is unsafe”

```haskell
r :: forall a. IORef a -- This is bad!
r = unsafePerformIO (newIORef (error "urk"))
```

• The compiler front-end and optimizer doesn’t know that the IO monad is special
  – It can be restrained by using an unkown “World” type that is “threaded through”
  – Then the back-end code generator can convert the “World”-y code to in-place imperative operations

```haskell
type IO t = World -> (t, World) -- in compiler front
return :: a -> IO a
(a,w) = (w) -> (a,w)
(m k = \w -> case m w of (r,w') -> k r w')
```
Summary

- A Haskell program is a single IO action called main. Inside the IO monad, evaluation order is defined.
- 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 “Outer Layer”

- In languages like ML or Java, the fact that the language is in the IO monad is baked into 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.

Now from here [time permitting]

- There are lots of other monads:
  - All monads have >>= and return.
  - They can differ on “what else they have”
  - Do-notation can be used with “any monad”
- You can write code that is “generic over” “which monad”:
  - Ridiculously powerful idiom for “threading things” without syntactic clutter (cf. when I showed you “state monad” in OCaml)
- Monad is a “typeclass”:
  - Haskell supports “other [user-defined] typeclasses too”
  - Integrates overloading with polymorphic lambda calculus.