Plan for Today

Discussion about Final (TIME HAS CHANGED!!)

– 2 sides of 8.5x11 sheet of paper
– Will NOT be posting example finals. Similar to midterm.
– Topics that could be on Final
  – Memory layout for MeggyJava programs, RTS and heap
  – LR(1) Parsing state diagrams and parsing tables
  – Performing an LR(1) parse using an LR(1) parse table
  – Symbol table and scoping questions
  – Operator precedence and associativity questions
  – Type checking and code gen for MeggyJava
– Friday will have review and can do examples.

Implementing IO monads
THE IO MONAD

Kathleen Fisher for cs242 at Tufts
Lightly edited with permission, Michelle Strout 4/13/15

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.
Monadic
Input and Output
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.
A value of type \((\text{IO } t)\) is an "action." When performed, it may do some input/output before delivering a result of type \(t\).
A value of type \((\text{IO } t)\) is an “action.” When performed, it may do some input/output before delivering a result of type \(t\).

\[
\text{type } \text{IO } t = \text{World } \rightarrow (t, \text{World})
\]
"Actions" are sometimes called "computations." An action is a first-class value. Evaluating an action has no effect; performing the action has the effect.

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

\[ \text{type IO } t = \text{World } \rightarrow (t, \text{World}) \]
Simple I/O

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

main :: IO ()
main = putChar 'x'
```
To read a character and then write it back out, we need to connect two actions.

The “bind” combinator lets us make these connections.
We have connected two actions to make a new, bigger action.

```
getChar >>>= putChar
```

declare
```
echo :: IO ()
echo = getChar >>= putChar
```
The ( >>= ) Combinator

- Operator is called bind because it binds the result of the left-hand action in the action on the right.

- Performing compound action \( a >>= \ x \rightarrow b \):
  - performs action \( a \), to yield value \( r \)
  - applies function \( \ x \rightarrow b \) to \( r \)
  - performs the resulting action \( b\{x <- r\} \)
  - returns the resulting value \( v \)
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 (>>>) Combinator

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

\[
(>>>) :: \text{IO } a \rightarrow \text{IO } b \rightarrow \text{IO } b
\]

\[
m \gg n = m \gg\gg (\_ \rightarrow n)
\]

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

```haskell
-- echoTwice :: IO ()
-- echoTwice = echo >> echo
```
Getting Two Characters

- We want to return \((c_1, c_2)\).
  - But, \((c_1, c_2) :: (\text{Char}, \text{Char})\)
  - And we need to return something of type \(\text{IO}(\text{Char}, \text{Char})\)

- We need to have some way to convert values of “plain” type into the I/O Monad.
The return Combinator

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

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

\[
\begin{aligned}
\text{getTwoChars} &:: \text{IO } (\text{Char},\text{Char}) \\
\text{getTwoChars} &= \text{getChar} \gg= \ \backslash c_1 \rightarrow \\
&\quad \text{getChar} \gg= \ \backslash c_2 \rightarrow \\
&\quad \text{return} \ (c_1,c_2)
\end{aligned}
\]
The “do” Notation

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

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

```haskell
-- 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 *only* adds syntactic sugar:

\[
\begin{align*}
\text{do } \{ x <- e; \ es \} &= e >>= \lambda x \rightarrow \text{do } \{ \ es \} \\
\text{do } \{ e; \ es \} &= e >> \text{do } \{ \ es \} \\
\text{do } \{ e \} &= e \\
\text{do } \{ \text{let } ds; \ es\} &= \text{let } ds \text{ in do } \{ \text{es} \}
\end{align*}
\]

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.
The following are equivalent:

```haskell
do { x1 <- p1; ...; xn <- pn; q }
```

```haskell
do x1 <- p1; ...; xn <- pn; q
```

```haskell
do x1 <- p1
   ...
   xn <- pn
   q
```

If the semicolons are omitted, then the generators must line up. The indentation replaces the punctuation.
The `getLine` function reads a line of input:

```haskell
g.getLine :: IO [Char]
g.getLine = do { c <- getChar ;
    if c == '\n' then
        return []
    else
        do { cs <- g.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.
GHC uses world-passing semantics for the IO monad:

```haskell
type IO t = World -> (t, World)
```

It represents the “world” by an un-forgeable token of type `World`, and implements `bind` and `return` as:

```haskell
return :: a -> IO a
return a = \w -> (a, w)

(>>=) :: IO a -> (a -> IO b) -> IO b
(>>=) m k = \w -> case m w of (r, w') -> k 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.”
But what does this mean?

Reference:

Summary
– IO Monad value for Haskell main is like a program AST
– The IO Monad value is evaluated in the sense that IO actions are bound together sequentially including some IO actions that contain lambda function values based on input.
– The ghc compiler then converts these IO Monad values into C code and executes the C code at runtime.

→ Show conversion to C example.