Fun With Haskell: IO

Nathaniel Wesley Filardo

January 17, 2012

Metadata
Questions?

- Any questions from last time?
- Any questions about the errata email?
    - Sorry about that.

Programming With Actions
A First Interactive Example

A classic example (and also [2, p59]):

- Prompt the user for their name.
- Accept the user's input.
- Say hi.
- Exit.

Programming With Actions
A First Interactive Example

A classic example (and also [2, p59]):

- Prompt the user for their name.
    - `putStrLn :: String -> IO ()`
- Accept the user's input.
- Say hi.
    - `putStrLn :: String -> IO ()`
- Exit.

Programming With Actions
A First Interactive Example

A classic example (and also [2, p59]):

- Prompt the user for their name.
  - `putStrLn :: String -> IO ()`
- Accept the user's input.
  - `getLine :: IO String`
- Say hi.
  - `putStrLn :: String -> IO ()`
- Exit.

Programming With Actions
A First Interactive Example

A classic example (and also [2, p59]):

- Prompt the user for their name.
    - `putStrLn :: String -> IO ()`
- Accept the user's input.
    - `getLine :: IO String`
- Say hi.
    - `putStrLn :: String -> IO ()`
- Exit.
    - A few things this could mean.
    - We will take it to be "return control flow"

Programming With Actions
A First Interactive Example

A classic example (and also [2, p59]):

- Prompt the user for their name.
- Accept the user's input.
- Say hi.
- Exit.

Programming With Actions
A First Interactive Example

A classic example (and also [2, p59]):

- Prompt the user for their name.
  - `putStrLn "Hello!  What's your name?"`
- Accept the user's input.

- Say hi.
  - `putStrLn $ "Hello, " ++ name`
- Exit.

Programming With Actions
A First Interactive Example

A classic example (and also [2, p59]):

- Prompt the user for their name.
  - `putStrLn "Hello!  What's your name?"`
- Accept the user's input.
  - `name <- getLine`
- Say hi.
  - `putStrLn $ "Hello, " ++ name`
- Exit.

Programming With Actions
A First Interactive Example

- Now all we have to do is glue it together.

Programming With Actions
A First Interactive Example

- Now all we have to do is glue it together.
- Using do notation:

HelloName.hs

```
main = do
  putStrLn "Hello! What's your name?"
  name <- getLine
  putStrLn $ "Hello, " ++ name
```

Programming With Actions
Aside: Alternate Syntaxes

- Given an action like

```
foo = do
  e1
  r <- a
  s <- b  -- b  may reference r
  e2      -- e2 may reference r and s
```

Programming With Actions
Aside: Alternate Syntaxes

- Given an action like

```
foo = do
  e1
  r <- a
  s <- b  -- b  may reference r
  e2      -- e2 may reference r and s
```

- Equivalent, without "two-dimensional syntax":

```
foo = do { e1 ; r <- a; s <- b; e2 }
```

Programming With Actions
Aside: Alternate Syntaxes

- Given an action like

```
foo = do
  e1
  r <- a
  s <- b   -- b  may reference r
  e2       -- e2 may reference r and s
```

- Equivalent, without "two-dimensional syntax":

```
foo = do { e1 ; r <- a; s <- b; e2 }
```

- Fully de-sugared form (note structure!):

```
foo = e1 >> a >>= (\r -> b >>= (\s -> e2))
```

Programming With Actions
Control Flow, Monadically

- Everybody have “and then” down?
- What about
  - if/then or if/then/else,
  - Loops (while, for),
  - Continuations (not going to cover this today, sorry)

Programming With Actions
Control Flow, Monadically

Another example from Hal [2, p60]:

- The number guessing game.
- Given an `Int`,
    - Ask the user for a guess
    - If incorrect, inform the user of the sign of error and go again.
    - If correct, congratulate the user and stop.

Programming With Actions
Control Flow, Monadically

- Ask the user for a guess
    - `getLine` will get us a `String`.
    - Slideware: use `read` to get an `Int`.
    - Real software: use `reads` or other, more safe, parser.
    - Real, Security software: don't use `Int`, `reads` an `Integer` and check for overflow (or do comparisons as `Integers`); timeout if the user takes too long, ...

Programming With Actions
Control Flow, Monadically

- Ask the user for a guess
  - `getLine` will get us a `String`.
  - Slideware: use `read` to get an `Int`.
  - Real software: use `reads` or other, more safe, parser.
  - Real, Security software: don't use `Int`, `reads` an `Integer` and check for overflow (or do comparisons as `Integers`); timeout if the user takes too long, ...
- Could use something like:

```
getRead = getLine >>= \l -> return (read l)
```

Programming With Actions
Control Flow, Monadically

- Ask the user for a guess
  - `getLine` will get us a `String`.
  - Slideware: use `read` to get an `Int`.
  - Real software: use `reads` or other, more safe, parser.
  - Real, Security software: don't use `Int`, `reads` an `Integer` and check for overflow (or do comparisons as `Integers`); timeout if the user takes too long, ...
- Could use something like:

```
getRead = getLine >>= \l -> return (read l)
```

- Standard library has a slightly nicer `readLn`.

Programming With Actions
Control Flow, Monadically

- See how the user fared. One possibility:

```
guessingGame :: Int -> IO ()
guessingGame n = do
  putStrLn "Take a guess!"
  n’ <- readLn
  if n == n’
   then {- ... -}
   else if n > n’
         then {- ... -}
         else {- ... -}
```

Programming With Actions
Control Flow, Monadically

```
guessingGame :: Int -> IO ()
guessingGame n = do
  putStrLn "Take a guess!"
  n' <- readLn
  case n `compare` n' of
    LT -> {- ... -}
    GT -> {- ... -}
    EQ -> {- ... -}
```

- Better, maybe. Why?

Programming With Actions
Control Flow, Monadically

```
guessingGame :: Int -> IO ()
guessingGame n = do
  putStrLn "Take a guess!"
  n' <- readLn
  case n `compare` n' of
    LT -> {- ... -}
    GT -> {- ... -}
    EQ -> {- ... -}
```

- Better, maybe. Why?
- Compiler can check that we didn't miss anything. ("Non-exhaustive match warnings")

Programming With Actions
Control Flow, Monadically

- "OK, sure, but what goes in those comments you've refused to expand for us?"

Programming With Actions
Control Flow, Monadically

- "OK, sure, but what goes in those comments you've refused to expand for us?"
- The `EQ` case is easy:

```
do putStrLn "Yes!"
```

- The "do" is actually superfluous.

Programming With Actions
Control Flow, Monadically

- The `LT` and `GT` cases require that we *loop*. How?
- Well, what do we want?

Programming With Actions
Control Flow, Monadically

- The `LT` and `GT` cases require that we *loop*. How?
- Well, what do we want?
- We want the guessing game action again:

```
do putStrLn "Too Low!"
   guessingGame n
```

Programming With Actions
Control Flow, Monadically

- The `LT` and `GT` cases require that we *loop*. How?
- Well, what do we want?
- We want the guessing game action again:

```
do putStrLn "Too Low!"
   guessingGame n
```

- "while" and "until" loops are typically expressed as *recursion*.
  - At the end of a do block / bind chain.

Programming With Actions
Control Flow, Monadically

NumberGuessGame.hs

```
guessingGame n = do
  putStrLn "Take a guess!"
  n’ <- (readLn :: IO Int)
  case n ‘compare‘ n’ of
    EQ -> putStrLn "Yes!"
    LT -> do putStrLn "Too high!"
             guessingGame n
    GT -> do putStrLn "Too low!"
             guessingGame n
```

Programming With Actions
The Point of No Return

- Coming from “those other” languages, the output of this IO action might be surprising:

```
foo x = do
  putStr "Test..."
  case {- ... -} of
    Nothing -> return ()
    Just x -> putStr $ show x
  putStrLn "...ing"
```

- What is the output?

Programming With Actions
The Point of No Return

- Coming from "those other" languages, the output of this IO action might be surprising:

```
foo x = do
  putStr "Test..."
  case {- ... -} of
    Nothing -> return ()
    Just x -> putStr $ show x
  putStrLn "...ing"
```

- What is the output?
- `return` makes a pure value into an action. It does not alter control flow.

Programming With Actions
Brain Teaser From Last Time

- Anybody try running this program?

```
twice a = a >> a
main = twice (putStrLn "Hello, World")
```

- What happens?

Programming With Actions
Brain Teaser From Last Time

- Anybody try running this program?

```
twice a = a >> a
main = twice (putStrLn "Hello, World")
```

- What happens?
- The type of twice is interesting:

```
twice :: Monad m => m a -> m a
```

- *Given* an action in the monad `m`, *produce* an action in the monad `m`.

Programming With Actions
Brain Teaser From Last Time

```
twice :: Monad m => m a -> m a
twice a = a >> a
```

- *Given* an action in the monad `m`, *produce* an action in the monad `m`.
- How about other actions?
  - Maybe and Reader aren't especially interesting. Why?

Programming With Actions
Brain Teaser From Last Time

```
twice :: Monad m => m a -> m a
twice a = a >> a
```

- *Given* an action in the monad `m`, *produce* an action in the monad `m`.
- How about other actions?
    - Maybe and Reader aren't especially interesting. Why?
    - `twice (modify (+1))`?

Programming With Actions
Brain Teaser From Last Time

- "*Given* an action in the monad `m`, *produce* an action in the monad `m`."
- This philosophy is (almost) unique to Haskell: the program we build does not *perform* any actions, it *describes* the actions (and any interrelations).
- That's what it means to *run* a program, though!
  - Program-centric view: Hook the program up to the real world.
  - World-centric view: *Interpret* the program's descriptions of actions within the real world.

Programming With Actions
Other Effectful Operators

- Actions are *values*.
- Some actions are just clever synonyms for pure code:
  - Maybe, Reader, State, ...
- IO actions may not be safely run *by* the program.
  - Program composes IO actions together and the outside world runs them at runtime.
- Some are in-between (e.g. `ST`)

Programming With Actions
Other Effectful Operators

- Haskell programs spend a lot of ink to combine actions.
- Of course: there are functions to help out.
- Notably, the `Control.Monad` module.

Programming With Actions
Other Effectful Operators

- "Do this when...":

```
when :: Monad m => Bool -> m () -> m ()
```

- For example, an Easter Egg:

```
main = do
  name <- getLine
  when (name == "Joshua") $ do
    putStrLn "Ah, Professor Falcon!"
  putStrLn $ "Hello, " ++ name
```

Programming With Actions
Other Effectful Operators

- "Do everything on this list in order:. . . ":

```
sequence  :: Monad m => [m a] -> m [a]
sequence_ :: Monad m => [m a] -> m ()
```

- And some utility forms:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
forM :: Monad m => [a] -> (a -> m b) -> m [b]
```

- So a for loop:

```
forM [1..10] (\x -> twice (print x))
```

Side Effects
Mutable References in IO

- Sometimes, mutation is exactly what you want;
- Accept no substitutes.

Side Effects
Mutable References in IO

- Enter `Data.IORef`.
- Basic operations:

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

- If you like: indexed get and put functions.
- Note: All of these are IO actions.

Side Effects
Mutable References in IO

- Suppose

```
mystery :: IO () -> IO ()
```

- And we want to count how many times mystery runs its argument (please ignore exceptions):

```
countMystery a = do
  r <- newIORef 0
  mystery (modifyIORef r (+1) >> a)
  c <- readIORef r
  putStrLn $ "Mystery ran its argument "
              ++ show c ++ "times"
```

Side Effects
Mutable Arrays in IO

- `Data.Array.IO` provides both
  - "Boxed" (non-strict), `IOArray`, and
  - "unboxed", `IOUArray`.
- Accessed via `MArray` class:

```
newListArray :: (MArray a e m, Ix i)
             => (i,i) -> [e] -> m (a i e)
readArray :: ... => a i e -> i -> m e
writeArray :: ... => a i e -> i -> e -> m e
...
```

Side Effects
The ST Monad

- Sufficiently often, we have code that is "externally pure" but may internally use mutation.
- This is OK if none of the mutation "escapes"
- The `ST` monad provides this encapsulation.
- `ST` is not `State`; please do not confuse them.

Side Effects
The ST Monad

- ST monad: `Control.Monad.ST`.

```
data ST s a = --hidden...
```

- The type parameter `s` is a **phantom**: it is not actually used in the definition. (How mysterious!)

Side Effects
The ST Monad

- ST monad: `Control.Monad.ST`.

```
data ST s a = --hidden...
```

- The type parameter `s` is a **phantom**: it is not actually used in the definition. (How mysterious!)
- ST monad references: `Data.STRef`.

```
newSTRef :: a -> ST s (STRef s a)
...
```

- Also `ST`-based arrays: `Data.Array.ST`.

Side Effects
The ST Monad

- Unlike IO, we *can* get "out" of `ST`.
- With this funny-looking function:

```
runST :: (forall s . ST s a) -> a
```

- A trick of quantification. Roughly: ensures that the type `a` does not mention the type `s`.
- Since all `STRefs` *do* mention `s`, ...
- Since `ST` does not (safely) have access to the `RealWorld`, every `runST a` should yield the same result.

Side Effects
Catching Exceptions in IO

- I'm going to ignore the Haskell 98 exception mechanism in favor of the more modern `Control.Exception`.
  - The original system handles only IO errors.
- Uses interesting type trickery to get one-level sub-typing.
  - A type class `Exception`.
- Lots of exception types:
  - `ArithException` includes `Overflow`, `DivideByZero`.
  - `PatternMatchFail`
  - `ErrorCall` (`error`, `head`, . . .)
  - `SomeException` (existential)

Side Effects
Catching Exceptions in IO

- In general, exceptions are control flow mechanism.
- Should be used as such! Try to avoid throwing them from pure code.
    - The standard library is old and has its own ideas.
    - Sorry.
- Formally: exceptions coming *from* pure code have *set* semantics, with nondeterministic representative selection.
    - "If this block of pure code throws an exception, you are guaranteed only that it was one that it *could* throw."

Side Effects
Catching Exceptions in IO

- Alright, now, on to catching.
- Primitive function:

```
catch :: Exception e
      => IO a -> (e -> IO a) -> IO a
```

- Compiler wants to know what *type* of exception (e) we want to catch.
    - Often inferred from use inside the handler.
    - Can be explicitly labeled, too.

Side Effects
Catching Exceptions in IO

- For example:

```
import Control.Exception as E
foo = E.catch (error "foo")
      (\(ErrorCall str) -> return $
     "caught error call: " ++ str)
```

- This call to `E.catch` catches only `ErrorCall`s.
  - Transparent to other exceptions.
- (**Qualified name** `E` needed to avoid `Prelude`'s `catch`.)

Side Effects
Catching Exceptions in IO

- For example:

```
import Control.Exception as E
foo = E.catch (error "foo")
      (\(ErrorCall str) -> return $
     "Caught: " ++ str)
```

- Rough transliteration:

```
try {
    throw new ErrorCall("foo");
} catch (ErrorCall e) {
    return ("Caught: " + e.toString());
}
```

Side Effects
Catching Exceptions in IO

From `catch` we can build up other combinators (this *is* Haskell!):

- `handle` flips the arguments to `catch`.
- `try` converts exceptions to `Either`:

```
try :: Exception e => IO a -> IO (Either e a)
```

- `bracket` lets us build exception-safe allocate&release:

```
      -- allocate    release            use
bracket :: IO a -> (a -> IO b) -> (a -> IO c)
        -> IO c
```

Side Effects
Laziness vs. Exceptions

- Consider

```
E.catch (return (error "Explode"))
        (\(ErrorCall _) -> return "Nope")
```

Side Effects
Laziness vs. Exceptions

- Consider

```
E.catch (return (error "Explode"))
        (\(ErrorCall _) -> return "Nope")
```

- Explodes! Huh!?

Side Effects
Laziness vs. Exceptions

- Consider

```
E.catch (return (error "Explode"))
        (\(ErrorCall _) -> return "Nope")
```

- Explodes! Huh!?
- Nothing we did inside the action triggered the explosion.

Side Effects
Laziness vs. Exceptions

- Fix here with the `evaluate` function:

```
evaluate :: a -> IO a
```

- To wit:

```
E.catch (evaluate (error "Explode"))
        (\(ErrorCall _) -> return "Nope")
```

- Other cases may be tricker (e.g. evaluating a pair).
- See the `deepseq` package.

Side Effects
A Small Example Using Files

- Files?

Side Effects
A Small Example Using Files

- Files?
- Oh right, IO. Specifically, `System.IO`:

```
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
hIsEOF :: Handle -> IO Bool
stdin, stdout, stderr :: Handle
```

Side Effects
A Small Example Using Files

- Files?
- Oh right, IO. Specifically, `System.IO`:

```
openFile :: FilePath -> IOMode -> IO Handle
hClose :: Handle -> IO ()
hIsEOF :: Handle -> IO Bool
stdin, stdout, stderr :: Handle
```

- `Handle`-taking variants of functions:

```
hPutStrLn :: Handle -> String -> IO ()
hGetLine :: Handle -> IO String
```

Side Effects
A Small Example Using Files

- Open a file
- For each line, read it as an Integer
- Sum them up in an IORef
- Output the result
- . . . ?

(For those of you following along on your laptops, this example is available as `LineSum.hs` on the website.)

Side Effects
A Small Example Using Files

- Open a file
- For each line, read it as an Integer
- Sum them up in an IORef
- Output the result
- . . . ?
- Close the file! . . . ?

(For those of you following along on your laptops, this example is available as `LineSum.hs` on the website.)

Side Effects
A Small Example Using Files

- Open a file
- For each line, read it as an Integer
- Sum them up in an IORef
- Output the result
- . . . ?
- Close the file! . . . ?
- On parse exception, warn and keep going!

(For those of you following along on your laptops, this example is available as `LineSum.hs` on the website.)

Side Effects
A Small Example Using Files

- The "open, do something, close" pattern is so common it's called `withFile`:

```
withFile :: FilePath -> IO Mode
         -> (Handle -> IO r) -> IO r
withFile fp im = bracket (openFile fp im)
                         (hClose)
```

- (Actual definition!)

Side Effects
A Small Example Using Files

- Would like to say

```
\ref str -> modifyIORef ref (+(read str))
```

- But that's not going to be exception safe.

Side Effects
A Small Example Using Files

- Would like to say

```
\ref str -> modifyIORef ref (+(read str))
```

- But that's not going to be exception safe.
- Try this instead:

```
\ref str -> do
    val <- evaluate $ read str
    modifyIORef ref (+val)
```

Side Effects
A Small Example Using Files

Wrap it with exception-catching goodness:

```
step ref str = handle
  (\(ErrorCall e) -> putStrLn $ "Warn: " ++ str)
  $ do
    val <- evaluate $ read str
    modifyIORef ref (+val)
```

Side Effects
A Small Example Using Files

- Define a combinator for looping over lines of a file.
- Type first:

```
eachLine :: (String -> IO ()) -> Handle -> IO ()
```

Side Effects
A Small Example Using Files

- Define a combinator for looping over lines of a file.
- Type first:

```
eachLine :: (String -> IO ()) -> Handle -> IO ()
```

- Now definition:

```
eachLine f h = do
  e <- hIsEOF h
  when (not e) $ do
    line <- hGetLine h
    f line
    eachLine f h
```

Side Effects
A Small Example Using Files

Define main:

```
main = do
  ref <- newIORef (0 :: Integer)
  withFile "LineSum.txt" ReadMode $
    eachLine (step ref)
  -- Handle automatically closed for us!
  total <- readIORef ref
  putStrLn $ "Total: " ++ (show total)
```

Next time

I think we should talk about concurrency:

- `forkIO` and explicit threading.
- `Data.Parallel.Strategies`
- Software Transactional Memory.

(I am sort of willing to be overruled, tho'.)

Bib

- Available from: http://courses.cms.caltech.edu/cs11/material/haskell/index.html.

- Hal Daumé III.
  Yet another haskell tutorial.
  2002–2006.
  Available from: http://www.cs.utah.edu/~hal/htut/.