Advances in Programming Languages
APL9: Monads and I/O

Ian Stark

School of Informatics
The University of Edinburgh

Monday 8 February
Semester 2 Week 5

Foreword

Some Types in Haskell

This is the third of four lectures about some features of types and typing in Haskell types, specifically:

- Type classes

- Multiparameter type classes, constructor classes,

- Monads and interaction with the outside world

- Encapsulating stateful computation

Foreword

Some Types in Haskell

This is the third of four lectures about some features of types and typing in Haskell types, specifically:

- Type classes

- Multiparameter type classes, constructor classes,

- Monads and interaction with the outside world

- Encapsulating stateful computation

Outline

Maybe type

Haskell has a standard type constructor for describing optional values.

```
data Maybe a = Nothing | Just a    -- Datatype declaration

isJust      :: Maybe a -> Bool     -- Some example operations
isNothing   :: Maybe a -> Bool     -- from the Data.Maybe library

instance Functor Maybe where       -- Remember constructor classes
fmap f Nothing = Nothing           -- from the last lecture?
fmap f (Just x) = (Just (f y))     -- etc. etc.
```

The Maybe type encapsulates an optional value. A value of type Maybe a is either empty (Nothing) or contains a value x of type a (Just x).

For example, functions can indicate potential failure by returning a result of Maybe type.

Example Maybe computations

```
-- Prepare a list of numbers in a given range, if suitable
f :: Int -> Int -> Maybe [Int]
f n m = if n <= m then Just [n..m] else Nothing


-- Extract an even number, if any
g :: [Int] -> Maybe Int
g xs = case filter even xs of
            [] -> Nothing
            (y:ys) -> Just y


-- Present as a string, if not too long
h :: Int -> Maybe String
h x = let s = show x in if length s < 4 then Just s else Nothing
```

Chaining Maybe computations

```
-- Do all three, one after another
getSmallEven :: Int -> Int -> Maybe String
getSmallEven p q = case f p q of
                       Nothing  -> Nothing
                       Just  xs ->
                               case g xs of
                                   Nothing -> Nothing
                                   Just y -> h y
```

This will return an even number between p and q as a string of no more
than three characters, if possible.

A combinator to chain Maybe computations

We can capture this pattern of chaining Maybe-functions with a suitable higher-order function.

```
andThenMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
andThenMaybe (Just x) f = f x
andThenMaybe Nothing f = Nothing

getSmallEven' :: Int -> Int -> Maybe String
getSmallEven' p q = f p q 'andThenMaybe' g 'andThenMaybe' h
```

Here andThenMaybe acts as a *combinator* on computations.

Perhaps extending Maybe

We can extend the Maybe type to our own Perhaps type, which carries
either a value, or an explanation for the absence of a result.

```
data Perhaps a = Valid a | Invalid String
             deriving Show

isValid   :: Perhaps a -> Bool        -- Some suitable
isInvalid :: Perhaps a -> Bool        -- operations
reason    :: Perhaps a -> Maybe String

instance Functor Perhaps where        -- This is a
  fmap f (Valid x)   = Valid (f x)    -- functor too
  fmap f (Invalid s) = Invalid s
```

Example Perhaps computations

```
-- Prepare a list of numbers in a given range, if suitable
f :: Int -> Int -> Perhaps [Int]
f n m = if n < m then Valid [n..m] else Invalid "Not valid range"


-- Extract an even number, if any
g :: [Int] -> Perhaps Int
g xs = case filter even xs of
           [] -> Invalid "No even numbers in list"
           (y:ys) -> Valid y


-- Present as a string, if not too long
h :: Int -> Perhaps String
h x = let s = show x
      in if length s < 4 then Valid s else Invalid "String too long"
```

Chaining Perhaps computations

```
-- Do all three, one after another
getSmallEven :: Int -> Int -> Perhaps String
getSmallEven p q = case f p q of
                        Invalid e -> Invalid e
                        Valid  xs ->
                                case g xs of
                                    Invalid e -> Invalid e
                                    Valid y -> h y
```

This will return an even number between p and q as a string of no more
than three characters; or an explanation why not.

A combinator to chain Perhaps computations

As before, a suitable combinator can capture the work needed to chain together computations.

```
andThenPerhaps :: Perhaps a -> (a -> Perhaps b) -> Perhaps b
andThenPerhaps (Valid x) f = f x
andThenPerhaps (Invalid e) f = Invalid e

getSmallEven' :: Int -> Int -> Perhaps String
getSmallEven' p q = f p q 'andThenPerhaps' g 'andThenPerhaps' h
```

Note that the code for the final program getSmallEven' is now *just the same* as it was for the Maybe computations.

Monads

Both Maybe a and Perhaps a present an enriched form of value type a,
adding extra "computational" information: a *monad*.   [Moggi '88, Wadler '92]

```
class Monad m where                        –– See the Haskell 98
  (>>=) :: m a –> (a –> m b) –> m b  –– report for full details
  return  :: a –> m a                         –– of the Monad class

instance Monad Maybe where     |   instance Monad Perhaps where
  Just x   >>= f = f x              |     Valid x   >>= f = f x
  Nothing >>= f = Nothing     |     Invalid e >>= f = Invalid e
  return x = Just x                 |     return x = Valid x

getSmallEven p q = f p q >>= g >>= h
```

More and more monads

Many other type constructors wrap up general kinds of "computation" as a monad. All have associated return and chaining $>>=$ operations.

```
data Exceptional e a = Result a | Exception e
type State s a = s -> (s,a)        -- Pass on a mutable value of type s
type Environment e a = e -> a      -- Look up in an environment of type e
type Printing a = (String,a)       -- Build up a String of output
type Read i a = [i] -> ([i],a)     -- Read values from list, pass what's left
type NonDeterministic a = [a]      -- Generate one, none, or many results
```

Exercise: Complete these as datatype declarations and Monad instances, then test them in GHC

Get more for your monad

Advantages of monads

- Separate the plumbing infrastructure from the code proper
- Code becomes independent of which monad is being used.
- Features can be added to the monad without changing client code.

Other monad applications

Monads encapsulate code, which can be used for more than just execution.

- Parsers
- Interpreters
- Exact real arithmetic
- Infinite search in finite time
- Metaprogramming

Why wrap things up?

Why not use the *real* state?

Monads capturing imperative programming might look like too much hard work. Why not just add real read/write and I/O operations to Haskell?

- Feature interaction: impurity is pervasive, and changes all other language properties.
- Laziness: imperative effects depend on evaluation order.
- Real state isn't real anyway: caching, multicore, virtual machines.

In the end, we want the compiler to have as much information, and as much freedom to work, as possible.

In practice, standard rewriting and liveness analysis can mean that straight-line use of the state monad maps to direct use of memory anyway.

Outline

Sweetening monads

Haskell provides syntactic sugar for writing monadic code.

```
f :: Int -> Int -> Perhaps [Int]   -- List the range, if possible

g :: [Int] -> Perhaps Int          -- Extract an even number, if any

h :: Int -> Perhaps String         -- As a string, if not too long

getSmallEven :: Int -> Int -> Perhaps String
getSmallEven p q =
    do  range <- f p q                -- The Perhaps monad
        evenNumber <- g range         -- ensures that any
        stringForm <- h evenNumber    -- error message makes
        return stringForm             -- it through to the end
```

Monadic syntax for everything

The do-notation works with any monad: lists, for example.

```
> do { x <- [1,2,3]; return (2*x); }
[2,4,6]

> do { x <- [1,2,3]; y <- ['a','b']; return (x,y); }
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

As a program-control mechanism, this captures backtracking. However, it also works as an alternative to list comprehension.

This in turn leads to the notion of *monad comprehension*

Outline

Routes to Input/Output

Back in the day, there were many solutions to getting functional languages to interact with the world outside.

- Side effects: just let it all happen

  As used in Lisp and Standard ML, but breaks purity and laziness.

- Stream transformers: Program :: [Request] −> [Response]

  Uses infinite lists and sincere laziness. Liable to deadlock.

- Continuation passing: **type** Compute a = forall b . (a −> b) −> b

  Tremendously powerful, but inverts all control (and intuition).

📄 Andrew D. Gordon.
Functional Programming and Input/Output.
Distinguished Dissertations in Computer Science. Cambridge
University Press, 1994.

One monad to rule them all

The arrival of monads in Haskell changed all of this, overnight.

In particular, the IO monad handles all interaction with the outside world.

```
main :: IO t                        —— Main program to execute

putChar :: Char —> IO ()            —— Output to terminal
print :: Show a => a —> IO ()

getLine :: IO String                —— Read from terminal
readFile :: FilePath —> IO String   —— or arbitrary file
```

One monad to rule them all

The arrival of monads in Haskell changed all of this, overnight.

In particular, the IO monad handles all interaction with the outside world.

```
ioError :: IOError -> IO a            -- Raise exception
catch :: IO a -> (IOError -> IO a) -> IO a  -- Handle exception

getArgs :: IO [String]        -- initial program arguments
system :: String -> IO ExitCode  -- call external program

getCPUTime :: IO Integer       -- picoseconds of CPU time used
```

One monad to rule them all

The arrival of monads in Haskell changed all of this, overnight.

In particular, the IO monad handles all interaction with the outside world.

```
ioError :: IOError -> IO a              -- Raise exception
catch :: IO a -> (IOError -> IO a) -> IO a  -- Handle exception

getArgs :: IO [String]        -- initial program arguments
system :: String -> IO ExitCode   -- call external program

getCPUTime :: IO Integer        -- picoseconds of CPU time used
```

One monad to rule them all

The arrival of monads in Haskell changed all of this, overnight.

In particular, the IO monad handles all interaction with the outside world.

```
ioError :: IOError -> IO a                  -- Raise exception
catch :: IO a -> (IOError -> IO a) -> IO a  -- Handle exception

getArgs :: IO [String]         -- initial program arguments
system :: String -> IO ExitCode  -- call external program

getCPUTime :: IO Integer       -- picoseconds of CPU time used
```

Over time, the IO monad has accumulated everything too impure to be in the language itself.

Dorian Gray had his picture; Haskell has the IO monad.

[Wilde, 1891]

Metaprogramming

Working with monads introduces a level of *metaprogramming*: the programmer can alternate between writing code inside the monad; and high-level manipulation outside it.

```
sequence   :: Monad m => [m a] -> m [a]

liftM      :: (Monad m) => (a -> b) -> (m a -> m b)

zipWithM   :: (Monad m) => (a->b->m c) -> [a]->[b]->m [c]

filterM    :: Monad m => (a -> m Bool) -> [a] -> m [a]
```

An expression of type IO a is a computation which when executed will return a value of type a.

An interactive Haskell program defines a computation, of type IO a; running the program performs that computation.

If Haskell is pure, then how does the IO monad work?

```
data World = ...

type IO a = World -> (World, a)
```

Strict typing, and the lack of any constructors for the World datatype, mean that the World must be single-threaded, not duplicated or destroyed, through any computation IO a.

This is preserved through extensive program rewriting and optimization, down to the compiled code. Only a single, mutable, value of type World is ever required: conveniently, exactly one is available, and can be efficiently updated in-place.

> The philosophers have only interpreted the world in various ways —
>
> the point however is to change it. [Marx, 1845]

Outline

More to do

Challenges

- Combining monads: monad transformers, layering
- Monolithic: how to modularise IO
- Explicit: smoother integration? Monad inference?

Future directions

- Arrows, idioms
- Operations and algebraic effects
- Effect types, effect inference
- ...

Outline

Homework

Reading

For Thursday, read the following.

📄 Simon L. Peyton Jones and Philip Wadler.
Imperative Functional Programming
In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages, POPL '93*, pages 71–84. ACM Press, 1993.

Homework

Find an online tutorial or other explanation of monads in programming, and post a link on the blog. Write a comment reviewing whether you found the explanation helpful, or otherwise. Bonus points if the language is *not* Haskell.

Further Reading

📄 Simon Peyton Jones.
Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell
Presented at the 2000 Marktoberdorf Summer School. In *Engineering Theories of Software Construction*, pages 47–96. IOS Press, 2001.
Latest version, January 2009, available online at http://research.microsoft.com/en-us/um/people/simonpj/Papers/marktoberdorf/

📄 Oscar Wilde.
The Picture of Dorian Gray.
Ward Lock & Co, London, 1891
Available from Project Gutenberg
http://www.gutenberg.org/etext/174

References

📄 Eugenio Moggi.
Computational Lambda-Calculus and Monads
Technical Report ECS-LFCS-88-66, Laboratory for Foundations of
Computer Science. Edinburgh, 1988.

📄 Phil Wadler.
The Essence of Functional Programming
In *Conference Record of the Nineteenth Annual ACM
SIGPLAN-SIGACT Symposium on Principles of Programming
Languages POPL '92*, pages 1–14. ACM Press, 1992.