How to declare an imperative

Bibliography

"*How to declare an imperative*", Philip Wadler. ACM Computing Surveys, 29(3):240–263, September 1997.

"Monads for functional programming", Philip Wadler, 2001. PDF

Commands

Modeling commands

How can we express **imperative input/output** in a purely-functional language?

Let's use an **embedded domain specific language**:

- a data type for *commands*
- *some constants* of this type (for primitive commands)
- *combinators* (for putting together complex commands from simpler ones)

A type for commands

Our initial type of commands is:

```
IO ()
```

For now: ignore the `()`-parameter and think of this as an "opaque" type.

Print a character

Let us consider a function `putChar` of the following type.

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

For example,

```
putChar '?'
```

denotes a command that, *if it is ever performed*, prints a single question-mark character.

Combining two commands

We can build more complex commands from two simpler ones by combining them sequentially.

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

For example,

```
putChar '?' >> putChar '!'
```

denotes a command that, *if it is ever performed*, prints a question-mark followed by an exclamation mark.

Doing nothing

It is useful to have a “null” command that doesn’t do anything.

```
done :: IO ()
```

Note that `done` doesn’t actually do nothing; it just denotes the command that, *if it is ever performed*, won’t do anything.

Compare *thinking* about doing nothing with *actually* doing nothing — they’re not the same thing!

Printing a string

We can build complex commands from simple ones.

Example: print a string, one character at a time.

```
putStr :: String -> IO ()
putStr []     = done
putStr (x:xs) = putChar x >> putStr xs
```

For example, `putStr "?!"` is equivalent to

```
putChar '?' >> (putChar '!' >> done)
```

Using higher-order functions

We could also express `putStr` using higher-order functions over lists.

```
putStr :: String -> IO ()
putStr = foldr (>>) done . map putChar
```

E.g.:

```
   putStr "?!"
=  foldr (>>) done (map putChar ['?','!'])
=  foldr (>>) done [putChar '?', putChar '!']
=  putChar '?' >> (putChar '!' >> done)
```

Main

How are commands ever performed?

Answer: the runtime system executes a “special” command named `main`.

```
-- file Hello.hs
main :: IO ()
main = putStr "Hello!"
```

Note that only `main` is executed even thought there may be other values of type `IO ()` in our program.

Equational reasoning

Replacing equals by equals

In both Haskell and OCaml, the terms

```
  (1+2)*(1+2)
```

and

```
let x = 1+2 in x*x
```

are equivalent (both evaluate to 9).

Equational reasoning lost

In OCaml `print_string : string -> ()` performs output as a *side-effect*.

We loose *referential transparency*, i.e. the ability to exchange identical sub-expressions.

```
print_string "ah"; print_string "ah"
   (* prints "ahah" *)

let x = print_string "ah" in x; x end
   (* prints a single "ah" *)

let f () = print_string "ah"
in f (); f () end
   (* prints "ahah" *)
```

Equational reasoning regained

In Haskell (unlike OCaml), the terms

```
putStr "ah" >> putStr "ah"
```

and

```
let m = putStr "ah"
in m >> m
```

are also equivalent (both denote a command that prints “ahah”).

Commands with values

Return values

`IO ()` is the type of commands that *return no useful value*.

> Recall that `()` is the *unit type* with a single inhabitant also written `()`.

More generally, `IO a` is the type of commands that *return a value of type* `a`.

```
IO Char           --  returns a single charater
IO (Char,Char)    --  ... a pair of characters
IO Int            --  ... a single integer
IO [Char]         --  ... a list of charaters
```

Reading a character

A command for reading the next input character:

```
getChar :: IO Char
```

E.g., if the available input is `"abc"` then `getChar` will yield the value `'a'` and the input remaining will be `"bc"`.

Doing nothing and returning a value

The command

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

does nothing and but returns the given value.

E.g. performing

```
return 42 :: IO Int
```

yields the value 42 and leaves the input unchanged.

Combining commands with values

The operator `>>=` (pronunced "bind") combines two commands and passes a value from the first to the second.

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

For example, performing the command

```
getChar >>= \x -> putChar (toUpper x)
```

when the input is `"abc"` produces the output `"A"` and the remaning input is `"bc"`.

Bind in detail

If

```
m :: IO a
k :: a -> IO b
```

then

```
m >>= k
```

is a command that acts as follows:

1. perform command `m` yielding `x` of type `a`
2. perform command `k x` yielding `y` of type `b`
3. yield the final value `y`

Reading a line

A program to read input until a newline and yield the list of characters read.

```
getLine :: IO [Char]
getLine = getChar >>= \x ->
          if x == '\n' then
             return []
          else
             getLine >>= \xs ->
             return (x:xs)
```

Commands as special cases

The general combinators for commands are:

```
return :: a -> IO a
(>>=)  :: IO a -> (a -> IO b) -> IO b
```

The command `done` is a special case of `return` and `>>` is a special case of `>>=`:

```
done :: IO ()
done = return ()

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

An analogy with *let*

The operator `>>=` behaves similarly to `let` when the continuation is a lambda expression.

Compare two type rules for `let` and `>>=`:

$$\frac{\Gamma \vdash m :: a \qquad \Gamma, x :: a \vdash n :: b}{\Gamma \vdash \text{let } x = m \text{ in } n :: b} \qquad \frac{\Gamma \vdash m :: \text{IO } a \qquad \Gamma, x :: a \vdash n :: \text{IO } b}{\Gamma \vdash m \gg= \lambda x \to n :: \text{IO } b}$$

"Do" notation

Echoing input to output

A program that echoes each input line in upper-case.

```
echo :: IO ()
echo = getLine >>= \line ->
       if line == "" then
         return ()
       else
         putStrLn (map toUpper line) >>
         echo
```

"Do" notation

Here's the same program using *"do" notation.*

```
echo :: IO ()
echo = do {
         line <- getLine;
         if line == "" then
            return ()
         else do {
            putStrLn (map toUpper line);
            echo
            }
       }
```

Translating “do” notation

Each line “`x <- e; ...`” becomes “`e >>= \x -> ...`”.

Each line “`e; ...`” becomes “`e >> ...`”.

Example

```
do { x1 <- e1;
     x2 <- e2;
     e3;
     x4 <- e4;
     e5;
     e6  }
```

is equivalent to

```
e1 >>= \x1 ->
e2 >>= \x2 ->
e3 >>
e4 >>= \x4 ->
e5 >>
e6
```

Monads

Monoids

A *monoid* is a pair ($\star$, u) of an *associative operator* $\star$ with an *identity value* u that satisfy the following laws:

Left-identity $u \star x \;=\; x$

Right-identity $x \star u \;=\; x$

Associativity $(x \star y) \star z \;=\; x \star (y \star z)$

Examples

`(+)` and `0`

`(*)` and `1`

`(||)` and `False`

`(&&)` and `True`

`(++)` and `[]`

`(>>)` and `done`

Monads

A *monad* is a pair of functions (`>>=`, `return`) that satisfy the following laws:

Left-identity `return a >>= f` $=$ `f a`

Right-identity `m >>= return` $=$ `m`

Associativity `(m >>= f) >>= g` $=$ `m >>= (\x -> f x >>= g)`

Monad laws in "do" notation

```
-- (1) Left identity
do { x'<-return x ; f x' }   =   do { f x }

-- (2) Right identity
do { x <- m; return x }      =   do { m }
```

Monad laws in "do" notation

```
-- (3) Associativity
do { y <- do { x <- m; f x }
     g y
   }
  =
do { x <- m;
     do { y <- f x; g y }
   }
  =
do { x <- m;
     y <- f x;
     g y
   }
```

The monad type class

Monad operations in Haskell are overloaded in a *type class*.

```
-- in the Prelude
class Monad m where
   return :: a -> m a
   (>>=)  :: m a -> (a -> m b) -> m b

instance Monad IO where
   return = ... -- primitive ops
   (>>=)  = ...

-- other Monad instances
```

The partiality monad

The Maybe type

```haskell
data Maybe a = Nothing | Just a
```

A value of type `Maybe a` is either:

`Nothing` representing the absence of further information;

`Just x` with a further value `x :: a`

Examples

```haskell
Just 42 :: Maybe Int
Nothing :: Maybe Int

Just "hello" :: Maybe String
Nothing      :: Maybe String

Just (42, "hello") :: Maybe (Int,String)
Nothing            :: Maybe (Int,String)
```

Representing failure

Partial functions can return a `Maybe` value:

- `Nothing` if the result is undefined;
- `Just r` when the result is `r`.

```haskell
-- Example: lookup a key in key-value list
-- (from the Prelude)
lookup :: Eq a => a -> [(a,b)] -> Maybe b
lookup k ((x,v):assocs)
   | k == x    = Just v  -- key found
   | otherwise = lookup k assocs
lookup k []    = Nothing -- key not found
```

Representing failure (2)

```haskell
phonebook :: [(String, String)]
phonebook = [ ("Bob",   "01788 665242"),
              ("Fred",  "01624 556442"),
              ("Alice", "01889 985333"),
              ("Jane",  "01732 187565") ]
```

E.g.:

```haskell
> lookup "Bob" phonebook
Just "01788 665242"
```

```
> lookup "Alice" phonebook
Just "01889 985333"
> lookup "Zoe" phonebook
Nothing
```

Combining lookups

Lookup up a name…

1. first in the phonebook
2. then in an email list

Return the pair of *phone*, *email* and fail if *either* lookup fails.

Combining lookups (2)

```
getPhoneEmail :: String -> Maybe (String,String)
getPhoneEmail name =
   case lookup name phonebook of
     Nothing -> Nothing
     Just phone -> case lookup name emails of
        Nothing -> Nothing
        Just email -> Just (phone,email)
```

This works but gets very verbose quickly!

Monads to the rescue

We can simplify this pattern because `Maybe` is a monad.

```
-- define in the Prelude
instance Monad Maybe where
   return x      = Just x
   Nothing >>= k = Nothing
   Just x  >>= k = k x
```

Specific types of the monad operations:

```
return :: a -> Maybe a
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

Re-writing the combined lookup

The code gets much shorter with `>>=` handling the failure cases.

```
getPhoneEmail :: String -> Maybe (String,String)
getPhoneEmail name =
```

```
   lookup name phonebook >>= \phone ->
   lookup name emails >>= \email ->
   return (phone,email)
```

Re-writing the combined lookup

Gets even simpler by using "do" notation.

```
getPhoneEmail :: String -> Maybe (String,String)
getPhoneEmail name
  = do phone <- lookup name phonebook
       email <- lookup name emails
       return (phone,email)
```

The error monad

Representing errors

If we need represent computations that may result in *distinct errors* we can use an `Either` result value:

```
-- from the Prelude
data Either a b = Left a | Right b
```

We can use:

- `Left` to tag errors;
- `Right` to tag valid results.

Example

Write an integer division function that may fail because:

- the divisor is zero; *or*
- the result is not exact.

Example (cont.)

```
myDiv :: Int -> Int -> Either String Int
myDiv x y
    | y == 0        = Left "zero division"
    | x`mod`y /= 0 = Left "not exact"
    | otherwise     = Right (x`div`y)

> myDiv 42 2
Right 21
```

```
> myDiv 42 0
Left "zero division"
> myDiv 42 5
Left "not exact"
```

Monad instance for Either

As with `Maybe`, there is a monad instance in the Prelude for `Either`.

```
-- in the Prelude
instance Monad (Either e) where
   return x      = Right x
   Left e >>= k  = Left e
   Right x >>= k = k x
```

Idea: `Left` values behave similiarly to *exceptions*.

Note that `Either e` is a monad but `Either` itself is *not* a monad (wrong kind).

Examples

```
> Right 41 >>= \x -> return (x+1)
Right 42

> Left "boom" >>= \x -> return (x+1)
Left "boom"

> Right 100 >>= \x -> Left "no way!"
Left "no way!"
```

Exercise: prove the monad laws for the `Either` instance.

The state monad

Representing stateful computations

Recall that we can view stateful computations as functions:

$$\text{state} \longrightarrow (\text{result}, \text{new state})$$

The state monad

```haskell
--- from Control.Monad.State

newtype State s a = State (s -> (a, s))
  -- type for state computations

run :: State s a -> s -> (a, s)
run (State f) s = f s

instance Monad (State s) where
   return a = State (\s -> (a, s))
   m >>= k  = State (\s ->
                 let (x, s') = run m s
                 in run (k x) s')
```

NB: for something to be a monad it should also satisfy the three monad laws — these are *not* checked by the compiler!