CS 360: Programming Languages
Lecture 12: Abstracting Computation

Geoffrey Mainland

Drexel University

Section 1

Abstracting `map`

A map for Maybe

- The function map lets us perform the same transformation on every element of a list, returning the transformed list.
- How could we write a similar function for the Maybe type? Recall:

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

- How about a map for Trees?

```
data Tree a = EmptyTree
            | Node a (Tree a) (Tree a)
  deriving (Show, Read, Eq)
```

- There is certainly a recurring pattern here...what is it?
- How can we capture it as a reusable abstraction?

A map for Maybe

- The recurring pattern we saw can be captured by the Functor abstraction. A functor is a "container with holes."
- Here is how we describe a functor:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- In the definition of Functor, f is a **type variable**, not a function.
- Also note that we *apply* f to a type in the type signature of fmap.
- A functor is a "*type* with a hole." What is a type with a hole? It's a type constructor that takes a single argument!
- Let's write the Functor instance for the Maybe type...

More Functor instances

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

- We can also write Functor instances for Trees:

```
instance Functor Tree where
    fmap f EmptyTree    = EmptyTree
    fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
```

- And lists:

```
instance Functor [] where
    fmap = map
```

- Notice that the argument to Functor isn't a "regular type," like Maybe Bool, but a **type constructor**. This type constructor takes an argument, which is a type, and gives us back a type.
- For example, Maybe takes an argument, e.g., Int, and gives us a type, i.e., Maybe Int.
- Similarly, the type constructor [] takes a type, e.g., Char, and gives us a type, i.e., [Char].
- **Type constructors construct types, and data constructors construct values**.

Types and Kinds

- Type constructors can take more than one type as an argument. For example

```
data Either a b = Left a
                | Right b
```

- Could we define a `Functor` instance for `Either`? If so, how?
- Could we define a `Functor` instance for `Either Int`? For `Either a`?
- Just as types classify values, **kinds** classify types. Fortunately, it stops there (for us).
- The list type constructor, `[]`, has kind `* -> *`. That is, it takes a type and returns a type.
- What kind do you think the `Bool` type constructor has? How about `*`.
- What kind do you think the `Either` type constructor has?
- In the `Functor` class definition, what kind does the type `f` have?

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Section 2

Abstracting Over Computation

Maybe and Null Pointers

- Haskell's `Maybe` is a bit like a null pointer in languages like Java or C/C++. Why is that the case?
- A value of type `Maybe a` represents a value of type `a` along with the possibility of failure.
- Imagine that you are writing a lot of code that uses `Maybe` to ensure that functions are total, i.e., you want your program to handle failure gracefully rather than simply aborting execution by calling `error`.
- What should happen when a `Nothing` value is encountered? How should execution proceed?
- How can we write a (higher-order) function that "chains together" functions that produce `Maybe`'s? Let's call this function `applyMaybe`.
- Can can we inject a pure value into the `Maybe` world? Let's write a function that performs this injection, calling it `pureToMaybe`.

Abstracting `Maybe`

- It us useful to think of a value of type `Maybe a` as a **partial value**.
- A function that returns a value of type `Maybe a` is therefore a **partial computation**.
- We want to chain together a bunch of partial computations and "exit" the chain early—as soon as we encounter a `Nothing` value.
- We don't care what is "inside" the `Maybe`—whatever we do to chain together `Maybe`'s should work no matter what type of value is in the `Maybe`!
- You can think of `applyMaybe` as a function that takes the result of a partial computation and *binds* it to the input of another partial computation, thus hooking them together.
- You can think of `pureToMaybe` as a function that takes a pure value and injects it into the world of partial values.

“Debuggable” functions

- Consider these pure, Haskell functions

```
f, g :: Float -> Float
```

- Let’s say we want to make these functions “debuggable” by adding a string to their return value.

```
f', g' :: Float -> (Float, String)
```

- Combining `f` and `g` (running one after the other) is easy—we can write `f . g`. But how can we combine `f'` and `g'`? Let’s try.
- Here’s a solution:

```
g'Thenf' :: Float -> (Float, String)
g'Thenf' x = let (y, s1) = g' x
                 (z, s2) = f' y
             in
               (z, s1++s2)
```

Chaining debuggable functions

- What if we have a lot of debuggable function we want to chain together? Having to write everything this way would be a real pain.
- Let create an abstraction, which we will call `bind`, that takes a value of type `(Float, String)`, a function of type `Float -> (Float, String)`, and hooks them together. You can think of `bind` as a function that takes the result of a debuggable function and *binds* it to the input of another debuggable function, thus hooking them together.

```
bind :: (Float, String)
     -> (Float -> (Float, String)) -> (Float, String)
bind (x,s1) f = let (y,s2) = f x
                in
                  (y,s1++s2)
```

- We would also like a way to take a `Float` and "inject" it into our abstraction. Let's call this function `pure`

```
pure :: Float -> (Float, String)
pure x = (x, "")
```

- Note that `pure x `bind` f` $\equiv$ `f x`.

Making functions debuggable

- Finally, how can we take a function `h :: Float -> Float` and make it debuggable? Let's define a function `lift` for this purpose.

```
lift :: (Float -> Float) -> Float -> (Float, String)
lift f x = (f x, "")
```

- Note that `lift f` $\equiv$ `pure . f`.

Summary: debuggable function

- Debuggable functions have type `Float -> (Float, String)`.
- We defined a function `bind` for chaining together debuggable functions.
- The function `pure` allows us to inject values into the world of debuggable functions.
- The function `lift` lets us use regular old functions as debuggable functions.

Multi-valued functions

- Consider the the functions `sqrt` and `cbrt` that compute the square root and cube root, respectively, of a real number. They both have type `Float -> Float`.
- Now consider versions of these functions that work with complex numbers. Every complex number, besides zero, has two square roots. Similarly, every non-zero complex number has three cube roots. So we'd like `sqrt'` and `cbrt'` to return *lists* of values.

```
sqrt',cbrt' :: Complex Float -> [Complex Float]
```

- We will call these "multivalued" functions.
- Suppose we want to find the sixth root of a real number. We can just compose the cube root and square root functions, e.g., `sixthroot x = sqrt (cbrt x)`.
- But how do we define a function that finds all the sixth roots of a complex number using `sqrt'` and `cbrt'`? We want to first find the cube roots of a number, then find the square roots of all of these numbers in turn, combining together the results into one long list.

Multi-valued functions

- What we need is a function, say `bind`, to compose these functions. Let's write it.

```
bind :: [Complex Float]
     -> (Complex Float -> [Complex Float])
     -> [Complex Float]
```

- How can we define `pure` and `lift`?
- You may see a pattern developing...how curious.

Random Numbers

- In Haskell, the `random` function looks like this
  ```
  random :: Random a => StdGen -> (a, StdGen)
  ```
- The idea is that to generate a random value you need a seed, and after you've generated the random value, you need to update the seed to a new value.
- In an imperative language we could keep the seed in a global variable, put in a pure language, we need to pass it around.
- You can think of a function of type `StdGen -> (a, StdGen)` as a **randomized computation** that produces a value of type `a`.
- Conceptually, a function `a -> b` that uses randomness can then be written as
  ```
  a -> StdGen -> (b, StdGen)
  ```
- How can we compose random functions? We need to define a function `bind`...
  ```
  bind :: (StdGen -> (a, StdGen))
       -> (a -> StdGen -> (b, StdGen))
       -> StdGen -> (b, StdGen)
  ```
- How can we define `pure` and `lift`?

Common Structure

- Now let's think about what structure these example have in common.
- Define the following, and use the type variable `m` to represent `Debuggable`, `Multivalued`, or `Randomized`.

```
type Debuggable a  = (a,String)
type Multivalued a = [a]
type Randomized a  = StdGen -> (a,StdGen)
```

- In each case we were faced with the same problem: Given a function of type `a -> m b`, we needed to somehow apply this function to an object of type `m a` instead of one of type `a`.
- In each case we do so by defining functions `pure` and `bind`

```
bind :: m a -> (a -> m b) -> m b
pure :: a -> m a
```

- The triple (`m`, `pure`, `bind`) is a **monad**.

Monads

- We can write a type class for monads as follows:

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

- Note that what we called pure is called `return`, and what we called `bind` is called `(>>=)` and is an infix operator.
- We can now write our multivalued function example as follows

```
sqrt', cbrt' :: Complex Float -> [Complex Float]

sixthroot :: Complex Float -> [Complex Float]
sixthroot x = return x >>= sqrt' >>= cbrt'
```

do Notation

- Monad are so pervasive in Haskell, that there is special syntax for using them—do notation.
- do notation allows us to rewrite

```
sqrt', cbrt' :: Complex Float -> [Complex Float]

sixthroot :: Complex Float -> [Complex Float]
sixthroot = return x >>= sqrt' >>= cbrt'
```

  as

```
sixthroot :: Complex Float -> [Complex Float]
sixthroot x = do y <- sqrt' x
                 cbrt' y
```

- We're able to write what looks like ordinary non-multivalued code and the implicit bind functions that Haskell inserts automatically make it multivalued.

do Notation

- do notation is just syntactic sugar!
- ```
  do x <- y
     more code
  ```

  is rewritten as

  ```
  y >>= (\x -> do more code)
  ```
- Similarly,

  ```
  do let x = y
     more code
  ```

  is rewritten as

  ```
  let x = y in do more code
  ```
- Finally,

  ```
  do expression
  ```

  is just

  ```
  expression
  ```
  ```

The IO Monad

- If you have been reading LYAH, you may have noticed functions like this

```
putStr :: String -> IO ()
```

- The function putStr prints a string on the terminal. How is that possible in a pure language? It lives in the IO monad!
- Think of how we might model input and output using pure math.
- In this mental model, the function putStr takes two arguments, a string and a value of type World. It then changes the World, by printing to the terminal, and returns a new World and the value ().
- Conceptually, the IO monad is

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

- This is just like our random function example, except instead of updating the seed, we update the whole world!