Lecture 9: From Maybe to Monads

CS4400 Programming Languages

Readings:

- Hutton: Programming in Haskell, Chapters 10 and 12
- Learn You a Haskell for Great Good, Chapters Input and Output & A Fistful of Monads

From Maybe to Monads

Since we introduced the Maybe types, we've been dealing more and more with the following pattern:

```
eval (Foo e1 e2) =
    case eval e1 of
         Just v1 ->
              case eval e2 of
                   Just v2 -> Just (... v1 ... v2 ...)
                   Nothing -> Nothing
         Nothing -> Nothing
```

How do we abstract this?

What is happening here?

We

1. evaluate e1, if it fails, fail, if not bind the result to v1
2. evaluate e2, if it fails, fail, if not bind the result to v2
3. combine v1 and v2 into a result.

First idea:

We abstract e1 and e2 and the expression in which we are using v1 and v2.

```
eval2 :: Expr -> Expr -> (Value -> Value -> Maybe Value) -> Maybe Value
eval2 e1 e2 f =
    case eval e1 of
         Just v1 ->
              case eval e2 of
                   Just v2 -> f v1 v2
                   Nothing -> Nothing
         Nothing -> Nothing
```

and our eval clause simplifies to:

```
eval (Foo e1 e2) = eval2 e1 e2 f
    where f v1 v2 = Just (... v1 ... v2 ...)
```

But we can do better by taking eval2 and making it more general and thinking of it as applying a partial binary function to two Maybe results.

```
apply2 :: (a -> b -> Maybe c) -> Maybe a -> Maybe b -> Maybe c
apply2 f (Just x) (Just y) = f x y
apply2 _ _ _ = Nothing
```

rewriting our eval case as:

```
eval (Foo e1 e2) = apply2 f (eval e1) (eval e2)
    where f v1 v2 = Just (... v1 ... v2 ...)
```

Compared to `eval2`, `apply2` gives as a little bit more flexibility as to what we apply it to. It still captures only one specific pattern of evaluation.

This takes care of evaluating arithmetic operators where we always first evaluate both operands before proceeding to apply the actual operation. But what about cases where the evaluation of one operand depends on the result of the other, such as `let`?

```
eval (Let x e1 e2) =
    case eval e1 of
         Just v1 -> eval (subst x v1 e2)
         Nothing -> Nothing
```

Here we cannot simply evaluate both e1 and e2, as the evaluation of e2 depends on the result of e1.

We can split it into two subexpressions, remembering the variables that are available to both:

```
-- known variables x e1 e2
eval e1 :: Maybe Value
eval (subst x v1 e2) :: Maybe Value
```

But where does v1 come from? What we are looking for is a way of "piping" the result from eval e1 into eval (subst x v1 e2) as v1. Think of Unix pipes, but with variable names.

We need to make the second expression dependent on v1, which we can do by introducing a lambda:

```
-- known variables x e1 e2
eval e1 :: Maybe Value
(\v1 -> eval (subst x v1 e2)) :: Value -> Maybe Value
```

Let's give them names: x and f and rewrite the case expression:

```
    case x of
         Just v -> f v
         Nothing -> Nothing
```

What happens if we abstract x and f?

```
pipe x f =
    case x of
         Just v -> f v
         Nothing -> Nothing
```

Let us rewrite the Let clause to use this function

```
eval (Let x e1 e2) =
    pipe (eval e1) (\v1 -> eval (subst x v1 e2))
```

It gets better if we use infix notation:

```
eval (Let x e1 e2) =
    (eval e1 `pipe` (\v1 -> eval (subst x v1 e2))
```

or even styled as

```
eval (Let x e1 e2) =
    eval e1 `pipe` \v1 ->
    eval (subst x v1 e2)
```

Note, how pipe is more general than apply2: we can express the add case as

```
eval (Add e1 e2) =
    eval e1 `pipe` \v1 ->
    eval e2 `pipe` \v2 ->
    Just (v1 + v2)
```

If we oonly used `pipe` with `eval`, its concrete type would be

```
pipe :: Maybe Value -> (Value -> Maybe Value) -> Maybe Value
```

However, there is nothing in the definition of `pipe` that says it has to only work with `Value`s so its type can be more general. In fact, there is nothing that says that the argument `x`'s type and `pipe`'s return type need to be the same: the function `f` can The actual, general type of `pipe` is:

```
pipe :: Maybe a -> (a -> Maybe b) -> Maybe b
```

This kind of "pipe" is so useful, that it's predefined in Haskell's base libraries as an operator of the `Monad` class:

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

When `>>=` is used with `Maybe`, we simply replace the `m` in the above type with `Maybe` and get `(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b` - the type of our `pipe` function. Now we can rewrite our `Add` evaluation as:

```
eval (Add e1 e2) =
    eval e1 >>= \v1 ->
    eval e2 >>= \v2 ->
    Just (v1 + v2)
```

What are monads? There are various ways of thinking about monads. One is thinking about them as containers: we have a type, e.g., `Value` and we wrap them up in a `Maybe` container as `Maybe Value`. This is somewhat misleading, however. A better way is to think of monads as "values with context". But it's probably best to think of monads in terms of what they can do for us: they allow us to express and chain computations.

In our eval example above, we evaluate `e1` and, if we succeed, remember its result as `v1` *then* we evaluate `e2` and, if we succeed, remember the result as `v2` and finally we signal success by wrapping the result of adding `v1` and `v2` in `Just`. If any of these evaluations returns `Nothing`, the computation will be short-circuited and return `Nothing` immediately. To emphasize this chaining aspect, Haskell provides special notation using the **`do`** keyword. The above `eval` clause can be rewritten as

```
eval (Add e1 e2) = do
    v1 <- eval e1
    v2 <- eval e2
    Just (v1 + v2)
```

In addition to >>=, the Monad class defines the function `return :: Monad m => a -> m a`. We can think of this as the tool that allows us to take a value and wrap it up as a computation. For `Maybe`, `return` is simply defined as

```
return :: a -> Maybe a
return x = Just x
```

Using this, the final version of our eval example is

```
eval (Add e1 e2) = do
    v1 <- eval e1
    v2 <- eval e2
    return (v1 + v2)
```

The same notation can be used with any monad. In particular, `IO`, the type of I/O computations (actions) is also a monad. This is why we use **`do`** to write interactive programs:

```
getAndGreet :: IO String
getAndGreet = do
    putStrLn "Enter your name: "
    name <- getLine
    putStrLn $ "Hello, " ++ name
    return name

main :: IO ()
```

```
main = do
    putStrLn "Welcome"
    name <- getAndGreet
    putStrLn $ "Bye, " ++ name
```

The above can be rewritten using `>>=`:

```
getAndGreet :: IO String
getAndGreet =
    putStrLn "Enter your name: " >>= \_ ->
    getLine >>= \name ->
    putStrLn ("Hello, " ++ name) >>= \_ ->
    return name

main :: IO ()
main =
    putStrLn "Welcome" >>= \_ ->
    getAndGreet >>= \name ->
    putStrLn $ "Bye, " ++ name
```

Back to Maybe: with Maybe the do notation has an additional advantage: we can use pattern match on the left-hand side of `<-` and if the match fails, our code will automatically return `Nothing`.

For example, say our `Value` type is defined as

```
data Value = Integer Integer
           | Boolean Bool
```

The we want to only allow addition for integer values:

```
eval (Add e1 e2) =
    case eval e1 of
         Just (Integer i1) ->
             case eval e2 of
                  Just (Integer i2) -> Just (Integer (i1 + i2))
                  _ -> Nothing
         _ -> Nothing
```

This can be rewritten simply as:

```
eval (Add e1 e2) = do
    Integer i1 <- eval e1
    Integer i2 <- eval e2
    return (Integer (i1 + i2)
```

Now, not only will the code return `Nothing` if either `eval` returns `Nothing`, it will also return `Nothing` if the value does not match the pattern on the left side of `<-`.