Haskell

Jethro Kuan

October 5, 2017

1 Introduction

1.1 Expressions

- Expressions include concrete values, variables, and also functions
- Functions are expressions that are applied to an argument, and hence can be *reduced* or *evaluated*

1.2 Infix/Prefix

```
div (prefix) -> `div` (infix)
+ (infix) -> (+) (prefix)
```

1.3 Let vs Where

Let introduces an expression, so it can be used wherever you can have an expression, but where is a declaration and is bound to a surrounding syntactic construct.

1.4 Typeclasses

Typeclasses are a way of adding additional functionality that is reusable across all the types that have instances of that typeclass. `Num` is a typeclass for most numeric types, that provide the basic operators `(+)`, `(-)`, `(*)` , `(/)` etc.

1.5 Datatype declaration

A datatype declaration defines a type constructor and data constructors. Data constructors are the values of a particular type, and are also functions that let us create data, or values, of a particular type.

1.6 Sectioning

Refers to the partial application of infix operators.

```
let x = 5
let y = (2^)
let z = (^2)

y x                             -- 32
z x                             -- 25

let celebrate = (++ " woot!")
celebrate "naptime" -- "naptime woot!"
celebrate "dogs" -- "dogs woot!"
```

1.7 Types

1.7.1 Polymorphism

1. Parametric polymorphism
    - Refers to type variables, or parameters, that are fully polymorphic

- When unconstrained by a typeclass, the final concrete type could be anything

2. Constrained polymorphism
   - Puts typeclass constraints on the variable, decreasing the number of concrete types it could be, but increasing what you can actually do with it by defining and bringing into scope a set of operations

Numeric literals are polymorphic and stay so until given a more specific type.

1.7.2 Parametricity

parametricity means that the behaviour of a function with respect to the types its (parametric polymorphic) arguments is uniform. The behaviour cannot change just because it was applied to an argument of a different type.

1.7.3 Making things more polymorphic

```
-- fromIntegral :: (Num b, Integral a) => a -> b
-- e.g.
6 / fromIntegral (length [1,2,3])
```

2 Laziness and Performance

Laziness can be a useful tool for improving performance, but more often than not it reduces performance by adding a constant overhead to everything. Because of laziness, the compiler can't evaluate a function argument and pass the value to the function, it has to record the expression in the heap in a suspension (or thunk) in case it is evaluated later. Storing and evaluating suspensions is costly, and unnecessary if the expression was going to be evaluated anyway.

One can force eager evaluation by prepending a bang(`!`) in front of the expression.

3 Typeclasses

Where a declaration of a type defines how that particular type is created, a declaration of a typeclass defines how a set of types are consumed or used in computations.

As long as a type implements, or instantiates a typeclass, then standard functions implemented on the typeclass can be used.

```
data DayOfWeek =
  Mon | Tue | Wed | Thu | Fri | Sat | Sun

-- day of week and numerical day of month

data Date =
  Date DayOfWeek Int
```

Because Eq is not derived in the typeclass, we need to instantiate one of our own:

```
instance Eq DayOfWeek where
  (==) Mon Mon = True
  (==) Tue Tue = True
  (==) Wed Wed = True
  (==) Thu Thu = True
  (==) Fri Fri = True
  (==) Sat Sat = True
  (==) Sun Sun = True
  (==) _ _ = False

instance Eq Date where
  (==) (Date weekday dayOfMonth) (Date weekday' dayOfMonth') =
    weekday == weekday' && dayOfMonth = dayOfMonth'
```

Typeclass instances are unique parings of the typeclass and a type. They define the ways to implement the typeclass methods for that type.

3.1 IO

An IO action is an action that, when performed, has side effects, including reading from input and printing to the screen, and will contain a return value.

In `IO ()`, `()` denotes an empty tuple, referred to as a *unit*. A unit is both a value and a type, that has only one inhabitant.

3.2 Summary

- A typeclass defines a set of functions and/or values;
- Types have instances of that typeclass
- The instances specify the ways that type uses the functions of the typeclass

4 Lists

```
data [] a = [] | a : [a]
```

4.1 Extracting portions of lists

```
take :: Int -> [a] -> [a]
drop :: Int -> [a] -> [a]
splitAt :: Int -> [a] -> ([a], [a])
```

```
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
```

4.2 Transforming lists of values

```
map :: (a -> b) -> [a] -> [b]
fmap :: Functor f => (a -> b) -> f a -> f b
```

```
map (+1) [1,2,3,4] -- [2,3,4,5]
map (1-) [1,2,3,4] -- [0,-1,-2,-3]
```

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter pred (x:xs)
  | pred x = x : filter pred xs
  | otherwise = filter pred xs
```

```
zip :: [a] -> [b] -> [(a,b)]
zip [1,2] [3,4] -- [(1,3), (2,4)]

zipWith (+) [1,2,3] [10,11,12] -- [11,13,15]
```

4.3 Folding lists

Folds as a general concept are called *catamorphisms*. *Catamorphisms* are a means of deconstructing data. If the spine of the list is the structure of a list, then a fold is what can reduce that structure.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z xs =
  case xs of
    [] -> z
    (x:xs) -> f x (foldr f z xs)
```

5 Algebraic Datatypes

A type can be thought of as an enumeration of constructors that have zero or more arguments.

Haskell offers sum types, product types, product types with record syntax, type aliases, and a special datatype called a newtype that offers a different set of options and constraints from either type synonyms or data declarations.

```
  data Bool = False | True
-- [1] [2] [3] [4] [5] [6]

  data [] a = [] | a : [a]
--      [7]   [8]    [9]
```

1. Keyword *data* to signal that what follows is a data declaration, or a declaration of a datatype
2. Type constructor (with no arguments)
3. Equals sign divides the type constructor from the data constructor
4. Data constructor. In this case, a data constructor that takes no arguments, so is called a *nullary* constructor.
5. Pipe denotes a sum type, which indicates a logical disjunction (colloquially *or*) in what values can have that type
6. Constructor for the value True, another nullary constructor
7. Type constructor with an argument. The argument is a polymorphic type variable, so the list's argument can be of different types
8. Data constructor for the empty list
9. Data constructor that takes two arguments, an a and also a [a]

5.1 Data and type constructors

Type constructors are used only at the type level, in type signatures and typeclass declarations and instances. Types are static and resolve at compile time.

Data constructors construct the values at term level, values you can interact with at runtime.

Type and data constructors that take no arguments are constants. They can only store a fixed type and amount of data.

5.2 Type constructors and kinds

Kinds are types of types, or types one level up. We represent kinds in Haskell with `*`. We know something is a fully applied, concrete type when it is represented as `*`. When it is `* -> *`, it is still waiting to be applied.

```
-- :k Bool
Bool :: *

-- :k [Int]
[Int] :: *

-- :k []
[] :: * -> *
```

Both `Bool` and [Int] are fully applied, concrete types, so their kind signatures have no function arrows.

5.3 Types vs Data

When data constructors take arguments, those arguments refer to other types.

```haskell
data Price =
  --  (a)
  Price Integer deriving (Eq, Show)
-- (b)  [1]
-- type constructor a
-- data constructor b
-- type argument [1]
```

5.4 What makes these datatypes algebraic?

Algebraic datatypes are so, because we can describe the patterns of argument structures using two basic operations: sum and product.

The cardinality of a datatype is the number of possible values it defines. Knowing how many possible values inhabit a type can help reason about programs.

The cardinality of `Bool` is 2, only being to take on `True` or `False`.

Datatypes that only contains a unary constructor always have the same cardinality as the type they contain.

```haskell
data Goats = Goats Int deriving (Eq, Show)
```

Here, `Goats` has the cardinality of `Int`.

5.5 Sum Types

Cardinality is obtained through summation. Example, Bool:

```haskell
data Bool = True | False
```

In this case, the cardinality of `Bool` is the sum of the cardinality of `True` and `False`.

5.6 Record syntax

```haskell
data Person =
  Person { name :: String
         , age :: Int }
         deriving (Eq, Show)
```

6 Signaling Adversity

6.1 Maybe

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

```haskell
type Name = String
type Age = Integer

data Person = Person Name Age Deriving (Eq, Show)

mkPerson :: Name -> Age -> Maybe Person
mkPerson name age
  | name /= "" && age >=0 = Just $ Person name age
  | otherwise = Nothing
```

mkPerson is a *smart constructor*. It allows us to construct values only if it meets a certain criteria.

6.2 Either

We use an `either` to figure out which criteria is not met:

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

```
data Person Invalid = NameEmpty | AgeTooLow deriving (Eq, Show)

mkPerson :: Name -> Age -> Either PersonInvalid Person
mkPerson name age
  | name /= "" && age >=0 - Right $ Person name age
  | name == "" = Left PersonInvalid
  | otherwise = Left AgeTooLow
```

`Left` is used as the invalid or error constructor. `Functor` will not map over the left type argument because it has been applied away.

6.2.1 Signalling Multiple errors

```
type Name = String
type Age = Integer
type ValidatePerson a = Either [PersonInvalid] a

data Person = Person Name Age deriving Show

data PersonInvalid = NameEmpty | AgeTooLow deriving (Eq, Show)

ageOkay :: Age -> Either [PersonInvalid] Age
ageOkay age = case age >= 0 of
  True -> Right age
  False -> Left [AgeTooLow]

nameOkay :: Name -> Either [PersonInvalid] Name
nameOkay name = case name == "" of
  True -> Left [NameEmpty]
  False -> Right name

mkPerson :: Name -> Age -> ValidatePerson Person
mkPerson name age =
  mkPerson' (nameOkay name) (ageOkay age)

mkPerson' :: ValidatePerson Name
          -> ValidatePerson Age
          -> ValidatePerson Person

mkPerson' (Right nameOk) (Right ageOk) = Right (Person nameOk ageOk)
mkPerson' (Left badName) (Left badAge) = Left (badName ++ badAge)
mkPerson' (Left badName) _ = Left badName
mkPerson' _ (Left badAge) = Left badAge
```

6.3 Anamorphisms

Anamorphisms are the dual of *catamorphisms*. Catamorphisms, or folds, break data structures down, anamorphisms builds up data structures.

```
-- iterate is like a very limited unfold that never ends
iterate :: (a -> a) -> a -> [a]

take 10 $ iterate (+1) 0
[0,1,2,3,4,5,6,7,8,9]

--unfoldr is more general
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]

take 10 $ unfoldr (\b -> Just (b, b+1)) 0
[0,1,2,3,4,5,6,7,8,9]
```

7 Monoids

In Haskell, algebras are implemented with typeclasses; the typeclasses define the set of operations. When we talk about operations over a set, the set is the *type* the operations are for.

One of those algebras we use in Haskell is Monoid.

`A monoid is a binary associative pattern with an identity.`

A monoid is a function that takes two arguments and follows two laws: associativity and identity.

1. Associativity: arguments can be regrouped or paranthesised in different orders and give the same result
2. Identity: there exists some value such that when it is passed as input to the function, the operation is rendered moot and the other value is returned. E.g. adding 0, multiplying by 1

Monoids are the pattern of summation, multiplication and list concatenation, among other things.

```
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
```

`mappend` is how any two values that inhabit the type can be joined together. `mempty` is the identity value for that mappend operation.

7.1 Examples of Monoids

7.1.1 List

```
mappend [1,2,3] [4,5,6]
-- [1,2,3,4,5,6]
mconcat [[1..3], [4..6]]
-- [1,2,3,4,5,6]
mappend "Trout" " goes well with garlic"
-- "Trout goes well with garlic"

instance Monoid [a] where
  mempty = []
  mappend = (++)
```

7.1.2 Integers

Integers form a monoid under summation and multiplication. Because it is unclear which rule is to be followed, there is no Monoid class under Integer, but there is the `Sum` and `Product` types that signal which Monoid instance is wanted.

7.2 Newtype

Using `newtype` constrains the datatype to having a single unary data constructor, and `newtype` guarantees no additional runtime overhead in "wrapping" the original type. The runtime representation of newtype and what it wraps are always identical.

```
(<>) :: Monoid m => m -> m -> m
```

`<>` is the infix version of `mappend`.

Monoid instances must abide by the following laws:

```
-- left identity
mappend mempty x = x

-- right identity
mappend x mempty = x
```

```haskell
-- associativity
mappend x (mappend y z) = mappend (mappend x y) z

mconcat = foldr mappend mempty
```

7.3 Monoid instances in `Bool`

```haskell
All True <> All True
-- All {getAll = True}

All True <> All False
-- All {getAll = False}

Any True <> Any False
-- Any {getAny = True}

Any False <> Any False
-- Any {getAny = False}
```

`All` represents boolean *conjuction*, while `Any` represents boolean disjunction.

For `Maybe`, `First` returns the "first" or leftmost non-Nothing value. `Last` returns the "last" or rightmost non-Nothing value.

```haskell
(First (Just 1)) <> (First (Just 2))
-- First {getFirst = Just 1}
```

```haskell
instance Monoid b => Monoid (a -> b)
instance (Monoid a, Monoid b) => Monoid (a,b)
instance (Monoid a, Monoid, b, Monoid c) => Monoid (a,b,c)
```

8 Semigroups

Semigroups are like monoids, but without the identity constraint. The core operation remains binary and associative.

```haskell
class Semigroup a where
  (<>) :: a -> a -> a

(a <> b) <> c = a <> (b <> c)
```

```haskell
data NonEmpty a = a :| [a] deriving (Eq, Ord, Show)
```

9 Functors

A functor is a way to apply a function over or around some structure that we don't want to alter. That is, we want to apply the function to the value that is "inside" some structure, and leave the structure alone.

This is why functors are generally introduced by way of fmapping over lists. No elements are removed or added, only transformed.

The typeclass `Functor` generalises this pattern, so that this basic idea can be used across different structures.

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

The argument `f a` is a Functor `f` that takes a type argument `a`. That is, the `f` is a type that has an instance of the Functor typeclass.

The return value is `f b`. It is the same `f` from `f a`, while the type argument b *possibly but not necessarily* refers to a different type.

fmap specialises to different types as such:

```
fmap :: (a -> b) -> f a -> f b
fmap :: (a -> b) -> [] a -> [] b
fmap :: (a -> b) -> Maybe a -> Maybe b
fmap :: (a -> b) -> Just a -> Just b
fmap :: (a -> b) -> Either a -> Either b
fmap :: (a -> b) -> (e,) a -> (e,) b
fmap :: (a -> b) -> Identity a -> Identity b
```

9.1 Functor Laws

9.1.1 Identity

```
fmap id == id
```

If we fmap the identity function, it should have the same result as passing our value to identity.

9.1.2 Composition

```
fmap (f . g) == fmap f . fmap g
```

9.1.3 Structure Preservation

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

The f is constrained by the typeclass Functor, but that is all we know about its type from this definition. Because the f persists through the type of `fmap`, whatever the type is, we know it must be a type that can take an argument, as in `f a` and `f b` and that it will be the "structure" we're lifting the function over when we apply it to the value inside.

9.2 Examples

```
data WhoCares a =
  ItDoesnt
  | Matter a
  | WhatThisIsCalled
  deriving (Eq, Show)
```

In the above datatype, only `Matter` can be *fmapped* over, because the others are nullary, and there is no value to work with inside the structure.

Here is a law-abiding instance of Functor.

```
instance Functor WhoCares where
  fmap _ ItDoesnt = ItDoesnt
  fmap _ WhatThisIsCalled = WhatThisIsCalled
  fmap f (Matter a) = Matter (f a)
```

This is a law-breaking instance:

```
instance Functor WhoCares where
  fmap _ ItDoesnt = WhatThisIsCalled
  fmap f WhatThisIsCalled = ItDoesnt
  fmap f (Matter a) = Matter (f a)
```

In this instance, the structure – not the values wrapped or contained within the structure – change.

9.3 Maybe and Either Functors

```
data Two a b = Two a b
```

Notice `Two` has the kind `* -> * -> *`, however, functors are of kind `* -> *`, and hence functors on the type Two would be invalid. we can reduce the kindness by doing the following:

```
instance Functor (Two a) where
  fmap f (Two a b) = Two a (f b)
```

Notice that we didn't apply `f` to `a`, because `a` is now part of the Functor structure, and is untouchable.

9.4 Ignoring possibilities

The Functor instances for the Maybe and Either datatypes are useful if you tend to ignore the left cases, which are typically the error or failure cases. Because fmap doesn't touch those cases, you can map your function right to the values that you intend to work with and ignore failure cases.

9.4.1 Maybe

```
incIfJust :: Num a => Maybe a -> Maybe a
incIfJust (Just n) = Just $ n + 1
incIfJust Nothing = Nothing

incMaybe :: Num a => Maybe a -> Maybe a
incMaybe = fmap (+1)
```

9.4.2 Either

```
incIfRight :: Num a => Either e a => Either e a
incIfRight (Right n) = Right $ n + 1
incIfRight (Left e) = Left e

-- can be simplified to
incEither :: Num a => Either e a => Either e a
incEither = fmap (+1)
```

9.5 Summary

`Functor` is a mapping between categories. In Haskell, this manifests as a typeclass which lifts a function between to types over two new types. This conventionally implies some notion of a function which can be applied to a value with more structure than the unlifted function was originally designed for. The additional structure is represented by the use of a higher kinded type *f*, introduced by the definition of the Functor typeclass.

To *lift over*, and later in Monad, to *bind over*, is a metaphor. One way to think about it is that we can lift a function into a context. Another is that we lift a function over some layer of structure to apply it.

```
fmap (+1) $ Just 1 -- Just 2
fmap (+1) [1,2,3] -- [2,3,4]
```

In both cases, the function we're lifting is the same. In the first case, we lift that function into a Maybe context in order to apply it, in the second case, into a list context.

The context determines how the function will get applied: the context is the datatype, the definition of the datatype, and the Functor instance we have for that datatype.

10 Applicative

Monoid gives us a means of hashing two values of the same type together.

Functor is for function application over some structure we don't want to have to think about.

The Applicative typeclass is a Monoidal Functor. The Applicative typeclass allows for function application lifted over structure (like Functor). But with Applicative the function we're applying is also embedded in some structure. Because the function and the value it's being applied to both have structure, we have to smash those structures together.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The `pure` function embeds something into functorial (applicative) structure.

`<*>` is an infix operation called 'apply'. This is very similar to the types of fmap.

```
-- fmap
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

the Control.Applicative library provides some convenience functions: `liftA`, `liftA2` and `liftA3`:

```
liftA :: Applicative f => (a -> b) -> f a -> f b
liftA2 :: Appplicative f => (a -> b -> c) -> f a -> f b -> f c
liftA2 :: Appplicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
```

`liftA` is just `fmap` with an Applicative typeclass constraint as opposed to a Functor typeclass constraint.

In `pure`, the left type is handled differently from the right:

```
pure 1 :: ([a], Int) -- ([], 1)
pure 1 :: Either a Int -- Right 1
```

The left type is part of the structure, and the structure is not transformed by the function application.

In a sense, Applicative is Monoid bolted onto a Functor to be able to deal with functions embedded in additional structure. In another, we're enriching function application with the very structure we were previously merely mapping over with Functor.

```
[(*2), (*3)] <*> [4,5] -- [2*4, 2*5, 3*4, 3*5]
= [8,10,12,15]
```

`<*>` takes a functor that has a function in it, and another functor and applies the function inside the functor. `<*>` is left associative.

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> something = fmap f something

instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
```

11 Monads

Monads are a natural extension to applicative functors. If you have a value with a context `m a`, how do you apply to it a function that takes a normal a and returns a value with a context?

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

In Prelude, `IO`, lists, and `Maybe` are members of the monadic classes.

```
-- infixl 1 >>=, >>
class Applicative m => Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
```

`»`, referred, to as bind, combines a Monad containing values of type `a`, and a function which operates on `a` and returns a monad of type `b`.

`»`, also sometimes called *Mr. Pointy*, is used when the function does not need the value of the first Monadic operator.

The precise meaning of bind depends on the monad. For example, in the `IO` monad, `x»=y` performs two actions sequentially, passing the result of the first into the second. For the lists and `Maybe` type, these monadic operations can be understood in terms of passing zero or more values from one calculation to the next.

The `do` syntax provides a simple shorthand for chains of monadic operations:

```
do e1 ; e2 = e1 >> e2
do p <- e1; e2 = e1 >>= (\v -> case v of p -> e2; _ -> fail "s")
```

The laws which govern `>>=` and `return` are:

```
return a >>= k          = k a
m >>= return            = m
xs >>= return . f       = fmap f xs
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

11.1 Built in Monads

11.1.1 Maybe

```
-- treating Maybe as unctors
fmap (++"!") (Just "wisdom") -- Just "wisdom!"
fmap (++"!") Nothing -- Nothing

-- treating Maybe as Applicatives
Just (+3) <*> Just 3 -- Just 6
Nothing <*> Just "greed" -- Nothing
max <$> Just 3 <*> Just 6 -- Just 6
max <$> Just 3 <*> Nothing -- Nothing

-- Upgrading to Monads
(\x -> Just (x + 1)) 1 -- Just 2
applyMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
applyMaybe Nothing f = Nothing
applyMaybe (Just x) f = f x

Just 3 `applyMaybe` \x -> Just (x + 1) -- Just 4
Nothing `applyMaybe` \x -> Just (x + 1) -- Nothing

-- applyMaybe is >>= for the Maybe monad
```

11.1.2 Lists

The monadic aspects of lists bring non-determinism into code in a clear and readable manner.

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
  fail _ = []
```

```
[3,4,5] >> = \x -> [x, -x]
-- [3, -3, 4, -4, 5, -5]
```

Non-determinism also includes support for failure. Here, the empty list `[]` is the equivalent of `Nothing`, because it signifies the absence of a result.

Just like with `Maybe` values, we can chain several lists with `>>=`:

```
[1,2] >>= \n -> ['a', 'b'] >>= \ch -> return (n,ch)
-- [(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]

do
  n <- [1,2]
  ch <- ['a', 'b']
  return (n,ch)
```

The list `[1,2]` gets bound to n, and `["a", "b"]` gets bound to ch. `return (n,ch)`, takes the tuple, and makes the smallest possible list that still presents (n,ch) as the result.

For lists, monadic binding involves joining together a set of calculations for each value in the list. When used with lists, the signature of `>>=` becomes:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
```

Given a list of `a`'s and a function that maps an a onto a list of `b`'s, `>>=` applies this function to each of the `a`'s in the input and returns the generated `b`'s concatenated into a list. The return function creates a singleton list.

The following two expressions are equivalent:

```
[(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]

do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x /= y)
   return (x,y)
```

11.2 Using Monads

We first analyse this state monad, built around a state type `s` that looks like this:

```
data SM a = SM (S -> (a, S)) -- The monadic type

instance Monad SM where
  -- defines state propogation
  SM c1 >>= fc2 = SM (\s0 -> let (r, s1) = c1 s0
                                 SM c2 = fc2 r
                             in
                               c2 s1)
  return k = SM (\s -> (k, s))

-- extracts the state from the Monad
readSM :: SM S
readSM = SM (\s -> (s, s))

-- updates the state of the monad
updateSM :: (S -> S) -> SM () -- alters the state
updateSM f = SM (\s -> ((), f s))

-- run a computation in the SM monad
runSM :: S -> SM a -> (a, S)
runSM s0 (SM c) = c s0
```

`SM` is defined to be a computation that implicitly carries a type `s`. `SM` consists of functions that take a state and produce two results: a returned value (of any type) and an updated state.

The instance declaration defines the 'plumbing' of the monad: how to sequence two computations and the definition of an empty computation.

Sequencing (`>>=`) defines a computation (denoted by the constructor `SM`) that passes the initial state, `s0` into `c1`, then passes the value coming out of this computation, `r`, to the function that returns the second computation, `c2`. Finally, the state coming out of `c1` is passed into `c2` and the overall result is the result of `c2`.

Here `return` doesn't change the state at all; it only serves to bring a value into the monad.

`readSM` brings the state out of the monad for observation while `updateSM` allows the user to alter the state in the monad.

12 Do Notation

```
routine :: Maybe Pole
routine = do
  start <- return (0,0)
  first <- landLeft 2 start
  second <- landRight 2 first
  landLeft 1 second
```