Lecture Notes: Haskell

Colin Unger

Contents

Haskell
Basics .......................................................... 1
Functions ......................................................... 1
Temporary Variables .......................................... 3
Control Flow ................................................... 4
Algebraic Data Types ......................................... 4
Type Aliases .................................................... 6
Function Composition & Partial Application ................. 6
Polymorphism .................................................... 7
Laziness & Purity ............................................... 8

Monads
Monad Lecture Recap .......................................... 10
Implementing Monads w/ Parameteric Polymorphism .......... 12
Implementing Monads w/ Virtual Tables ....................... 12
Typeclasses .................................................... 16
Using Monads ................................................... 17
  Example Monad Instances .................................... 17
do-notation .................................................... 30

Beyond Monads
Functor .......................................................... 32
Typeclass Constraints ......................................... 33
Applicative ...................................................... 33

Haskell
Basics

Functions
A Haskell program is made up of a series of function definitions. To define a function, at the top level of the file we state the function name, any arguments, and the result of the function. Note that all functions must have a result, i.e., there are no void functions. For example, to define a function called dup'\(^1\) that takes a single element and returns a 2-tuple of that element, we would write

\[
\text{dup}' \; x = (x, x)
\]

Function application is done by juxtaposition, e.g.,

\(^1\)The ' in dup' is part of the function name and has no special meaning. In other code a a trailing apostrophe will often be used to avoid any conflicts with preexisting function names. However, since I’ve told Haskell to only import a small piece of the standard library, in this document we will not need to include apostrophes.
Binary functions can also be applied infix using backticks:

```haskell
myBinaryFunction x y = x + y

1 `myBinaryFunction` 2
3
```

Haskell is a strongly-typed language, but Haskell’s type inference is powerful enough that you frequently do not need to explicit add type annotations. For example, in ghci we can ask for the inferred type of an expression using :type, e.g.,

```
:type (dup' "hi")
(dup' "hi") :: (String, String)
```

where :: should be read as “has type”. To avoid ambiguity and increase readability, functions in Haskell are frequently annotated with their types, e.g.,

```
dup' :: String -> (String, String)
dup' x = (x, x)
```

where a -> b denotes the type of a function from a type a to a type b.

As we have already seen, Haskell also has tuple types, denoted with surrounding parentheses and elements separated by commas, e.g.,

```
myTwoTuple :: (String, String)
myTwoTuple = ("hello", "world")

myThreeTuple :: (String, String, String)
myThreeTuple = ("hello", "world", "!")
```

Note that tuples of different lengths have different types, e.g.,

```
myTwoTuple == myThreeTuple
<interactive>:71:15: error:
    Couldn't match expected type: (String, String)
      with actual type: (String, String, String)
    In the second argument of (==), namely myThreeTuple
    In the expression: myTwoTuple == myThreeTuple
    In an equation for it: it = myTwoTuple == myThreeTuple
```

There even exist tuples with zero elements, e.g.,

```
myZeroTuple :: ()
myZeroTuple = ()
```

Zero-length tuples are useful because there is only one valid value of type (): (), a.k.a. unit. As such, returning a value of type () from a function is essentially equivalent to returning nothing, as the value returned is always the same, i.e., (). Thus, () is the closest Haskell has to the void return type in C/C++.

To access the elements of a tuple, we can use “pattern matching”—in a function’s parameter list, instead of listing variables names we can instead list the “shapes” of the variables with variable names for each piece, e.g.,

```
swap :: (Int, Char) -> (Char, Int)
swap (l, r) = (r, l)
```

To ignore a piece of an argument, we can instead give it the name _, e.g.,

```
2Unlike Python where _ is just a variable name like any other, in Haskell _ has special meaning. Don’t expect it to work like
```
We can define anonymous functions (i.e., lambdas) through the notation \( \text{\texttt{\textbackslash arg0 arg1 ... \rightarrow body}} \), e.g.,

```haskell
fst :: (Int, Char) -> Int
fst (l, _) = l
```

As show above lambdas also support pattern matching. We can also see that listing all of a function's parameters is unnecessary, as long as the function type works out correctly.

Haskell also allows us to define custom operators similar to how we define functions, e.g.,

```haskell
(>#<) :: Int -> Int -> Int
l #>\< r = l + 2 * r
```

1 #>\< 2
5

**Temporary Variables**

In more complex functions, it can be convenient to introduce additional temporary variables. In Haskell we can do this either via `let-in`, e.g.,

```haskell
sum3 :: (Int, Int, Int) -> Int
sum3 (x, y, z) = let intermediate = x + y
                in intermediate + z
```

or via a `where` clause, e.g.,

```haskell
sum3' :: (Int, Int, Int) -> Int
sum3' (x, y, z) = intermediate + z
where
  intermediate :: Int
  intermediate = x + y
```

Note that variables in Haskell are immutable, so we can’t reassign them in a `let` or `where`, e.g.,

```haskell
sum4 :: (Int, Int, Int, Int) -> Int
sum4 (w, x, y, z) = let intermediate = w + x
                     intermediate = intermediate + y
                     in intermediate + z
```

```
<interactive>:110:25: error:
Conflicting definitions for intermediate
Bound at: <interactive>:110:25-36
<interactive>:111:25-36
```

so we would instead have to assign them different names, e.g.,

```haskell
sum4' :: (Int, Int, Int, Int) -> Int
sum4' (w, x, y, z) = let intermediate = w + x
                      intermediate' = intermediate + y
                      in intermediate' + z
```

In fact, all values in Haskell are immutable. For example, to increment a number in a tuple we create a new tuple with the new value, but leave the old tuple unmodified, e.g.,

```haskell
incFst :: (Int, Int) -> (Int, Int)
incFst (l, r) = (l + 1, r)
```

```haskell
eampleTuple :: (Int, Int)
eampleTuple = (3, 5)
a normal variable!
```
Control Flow

Instead of `while` and `for` loops we have recursion:

```haskell
fibonacci :: Int -> Int
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n = fibonacci (n - 1) + fibonacci (n - 2)
```

We do, however, have conditionals in the form of `if`:

```haskell
if (1 + 1 == 2) then "math works" else "um..."
"math works"
```

and `case` (which performs pattern matching, similar to functions):

```haskell
isEven :: Int -> Bool
isEven 0 = True
isEven n = isOdd (n - 1)

isOdd :: Int -> Bool
isOdd 0 = False
isOdd n = isEven (n - 1)

exampleNum :: Int
exampleNum = 3

case (isEven exampleNum, isOdd exampleNum) of
  (True, False) -> "math works! it's even"
  (False, True) -> "math works! it's odd"
  _ -> "um...
"math works! it's odd"
```

Note that because every expression must have a value, every `if` must have both a `then` and an `else` branch.

Algebraic Data Types

Haskell supports algebraic data types (ADTs), which are analogous to the combination of C's `struct` and `union`:

```haskell
data TwoOrThree = Two Int Int | Three Int Int Int
```

Here we declare a new type `TwoOrThree` which has two `type constructors`: `Two` and `Three`, analogous to the following C code:

```c
#include <stdbool.h>

union _TwoOrThree {
  bool isTwo;
  struct {
    int field1;
    int field2;
  } asTwo;
  struct {
```
typedef union _TwoOrThree TwoOrThree;

TwoOrThree Two(int arg1, int arg2) {
  TwoOrThree toReturn;
  toReturn.isTwo = true;
  toReturn.asTwo.field1 = arg1;
  toReturn.asTwo.field2 = arg2;
  return toReturn;
}

TwoOrThree Three(int arg1, int arg2, int arg3) {
  TwoOrThree toReturn;
  toReturn.isTwo = false;
  toReturn.asThree.field1 = arg1;
  toReturn.asThree.field2 = arg2;
  toReturn.asThree.field3 = arg3;
  return toReturn;
}

Two takes two values of type Int and returns a value of TwoOrThree, e.g.,

:type Two
Two :: Int -> Int -> TwoOrThree

and Three takes three values of type Int, and also returns a value of TwoOrThree, e.g.,

:type Three
Three :: Int -> Int -> Int -> TwoOrThree

Since both Two and Three produce a value of type TwoOrThree, if we are given a value of type TwoOrThree we don’t know how many fields it has: if it was constructed via Two then it has two Int fields, and if it was constructed by Three then it has three. To determine which is the case, we can use pattern matching to take a value of type TwoOrThree and determine which constructor was used:

sumTwoOrThree :: TwoOrThree -> Int
sumTwoOrThree (Two x y) = x + y
sumTwoOrThree (Three x y z) = x + y + z

which is equivalent to the C code

```
int sumTwoOrThree(TwoOrThree arg) {
  if (arg.isTwo) {
    return arg.asTwo.field1 + arg.asTwo.field2;
  } else {
    return arg.asThree.field1 + arg.asThree.field2 + arg.asThree.field3;
  }
}
```

We can also define recursive data types: for example, to define our own list datatype

data MyList = EmptyList | Cons Int MyList

sumMyList :: MyList -> Int
sumMyList EmptyList = 0
sumMyList (Cons myHead myTail) = myHead + sumMyList myTail

sumMyList (Cons 1 (Cons 3 (Cons 2 EmptyList)))
6
In fact, this is exactly how Haskell defines lists, just with `EmptyList` replaced by the symbol `[]` and `Cons` replaced by the symbol `:`, e.g.,

```haskell
myListToHaskellList :: MyList -> [Int]
myListToHaskellList EmptyList = []
myListToHaskellList (Cons myHead myTail) =
    myHead : (myListToHaskellList myTail)
```

```haskell
myListToHaskellList (Cons 1 (Cons 3 (Cons 2 EmptyList)))
[1,3,2]
```

Note that type constructors can have the same name as their result type, e.g.,

```haskell
data IntWrapper = IntWrapper Int
```

### Type Aliases

You can declare an *alias* of a type using the `type` keyword, e.g.,

```haskell
type Temperature = Int
```

Note that `Temperature` is not a wrapper for `Int`, `Temperature` is `Int`, e.g.,

```haskell
myTemperature :: Temperature
myTemperature = 78

myInt :: Int
myInt = 78

myInt == myTemperature
True
```

Thus type aliases should be used to improve readability, not to improve type safety.

### Function Composition & Partial Application

As a functional language, one of the most important operations in Haskell is function composition. In Haskell this is most commonly done by the operator `.` , which is equivalent to the standard \( \circ \) operator in mathematics. Note that function composition here is performed right-to-left, *not* left-to-right, e.g.,

```haskell
f :: String -> String
f s = "f(" ++ s ++ ")"

g :: String -> String
g s = "g(" ++ s ++ ")"

x :: String
x = "x"

(f . g) x
"f(g(x))"
```

This is convenient in that when applied the ordering of variables remains the same (i.e., left-to-right \( f, g, x \)). Haskell performs *partial application* (e.g., *currying*) of functions by default, e.g.,

```haskell
myAdd :: Int -> Int -> Int
myAdd l r = l + r

:type (myAdd 1)
(myAdd 1) :: Int -> Int
```

Here, applying a function that takes two parameters to a single parameter returns a function with one
parameter filled and the other one still waiting to be filled (thus the change from
signaturee $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ to $\text{Int} \rightarrow \text{Int}$). In most other
languages, for example Python, the above would instead result in an error:

```python
def myAdd(l: int, r: int) -> int:
    return l + r
myAdd(1)
```

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: myAdd() missing 1 required positional argument: 'r'

Note that this implies that we can interpret a Haskell function signature
in multiple ways: for example, we can read $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
as “a function that takes two Ints and returns an Int”, but we can also read it as “a
function that takes an Int and returns a function of type $\text{Int} \rightarrow \text{Int}$, as we saw above in the case
of myAdd. Thus, the type signature $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$ is equivalent to $\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})$, indicating that
the $\rightarrow$ operator is right-associative$^3$. Thus, the following type signatures are equivalent:

```haskell
parenExample1 :: Int -> Int -> (((Int -> Int) -> Int) -> Int -> Int)
parenExample1 = undefined

parenExample2 :: Int -> Int -> ((Int -> Int) -> Int) -> Int -> Int
parenExample2 = undefined

parenExample3 :: (Int -> (Int -> (((Int -> Int) -> Int) -> (Int -> Int))))
parenExample3 = undefined
```

The following types are equivalent:

```haskell
:type parenExample1
parenExample1 :: Int -> Int -> ((Int -> Int) -> Int) -> Int -> Int

:type parenExample2
parenExample2 :: Int -> Int -> ((Int -> Int) -> Int) -> Int -> Int

:type parenExample2
parenExample2 :: Int -> Int -> ((Int -> Int) -> Int) -> Int -> Int
```

### Polymorphism

Like the polymorphic lambda calculus covered in Lecture 4, Haskell supports universally quantified types.
For example, in the polymorphic lambda calculus the identity function $\text{id} = \lambda x. x$ has type $\text{id}: \forall x. x \rightarrow x$.
Similarly, in Haskell we can write

```haskell
id :: forall x. x -> x
id v = v
```

In Haskell `forall`s are automatically inserted by the compiler for any undefined type variables in a top-level
type declaration, so we can also write

```haskell
id :: forall x. x -> x
id v = v
```

Haskell also supports polymorphic data types: for example, to define a generic list type,

```haskell
data MyGenericList e = GenericCons e (MyGenericList e)
```

$^3$A binary operation ($\cdot$) is right-associative if $a \cdot b \cdot c = (a \cdot (b \cdot c))$. A common example
is assignment in C: $a = b = c$ can be read as “set $b$’s value to $c$, and then set $a$’s value to the result of $b = c$”, i.e., $a = (b = c)$. In contrast, subtraction is
left-associative: $a - b - c = (a - (b - c))$.

`undefined` is a special value in Haskell that has all types, and so can be substituted for any expression
without violating typechecking. Here we use it as the implementations of `parenExample1`, `parenExample2`, and `parenExample3` are irrelevant
to the example. If encountered at runtime `undefined` will cause the executable to exit with an error message.
myStringList = (GenericCons "hello" (GenericCons "world" (GenericCons "!" GenericEmptyList)))

myBoolList = (GenericCons True (GenericCons False GenericEmptyList))

myGenericHead :: MyGenericList e -> e
myGenericHead (GenericCons h _) = h
myGenericHead GenericEmptyList = error "Cannot take head of empty list"

myGenericHead myStringList
"hello"

myGenericHead myBoolList
True

Note that just as in the polymorphically typed lambda calculus, the type of polymorphism here is **parametric polymorphism**: if a function \( f \) is quantified over a type variable \( t \), then \( f \) must behave exactly the same (equivalently, \( f \)'s implementation may make no assumptions about \( t \)) for any type. Thus, using the features discussed so far it is be impossible to define a join function that works for all element types, e.g.,

genericJoin :: MyGenericList e -> e
genericJoin (GenericCons head tail) = head ++ (genericJoin tail)

<interactive>:295:39: error:
  Couldn't match expected type e with actual type [a0]
  e is a rigid type variable bound by
  the type signature for:
    genericJoin :: forall e. MyGenericList e -> e
  at <interactive>:294:1-35

In the expression: head ++ (genericJoin tail)
In an equation for genericJoin:
  genericJoin (GenericCons head tail) = head ++ (genericJoin tail)

We'll discuss more expressive forms of polymorphism later in the lecture (here and here).

### Laziness & Purity

Unlike most other languages, Haskell uses lazy semantics, a.k.a. normal order, a.k.a. call-by-name, instead of the more commonly used strict semantics, a.k.a. eager evaluation, a.k.a. call-by-value. This means that Haskell only evaluates expressions that are needed to compute the program result, which allows easy computation over infinite data structures, e.g.,

myInfiniteList :: [Int]
myInfiniteList = 0 : myInfiniteList

myHead :: [Int] -> Int

---

5See lecture 3 for the definition of these terms
In both of these cases we are able to evaluate the given function (i.e., `myHead` and `myListProduct`) over the infinite list `myInfiniteList` because they only need to know the head of the list to return the value.\(^6\) In strict languages (e.g., Python, C, Java, and almost every other language you’ve used) the evaluation order would require that the arguments to `myHead` (or `myListProduct`) be evaluated before evaluating `myHead` (or `myListProduct`), and since `myInfiniteList` recurses infinitely an equivalent program in a strict language would never terminate.

While lazy evaluation has the advantage of being “more terminating”\(^7\) than strict evaluation, it complicates the program’s IO behavior. For example, consider the following program in pseudo-Haskell syntax:

\[
\textbf{myfunc} = \lambda x \rightarrow (\text{print } x; x) \\
\textbf{result} = 0 \times \textbf{myfunc} 1
\]

where ; denotes sequential execution\(^8\). Interpreted in strict semantics, we’ll get the following reduction steps:

\[
\begin{align*}
0 \times \textbf{myfunc} 1 & \ [\ \text{stdout: } \texttt{""}] \\
\rightarrow 0 \times 1 & \ [\ \text{stdout: } \texttt{"1"}] \\
\rightarrow 0 & \ [\ \text{stdout: } \texttt{"1"}]
\end{align*}
\]

as expected. However, in lazy semantics something surprising occurs:

\[
\begin{align*}
0 \times \textbf{myfunc} 1 & \ [\ \text{stdout: } \texttt{""}] \\
\rightarrow 0 & \ [\ \text{stdout: } \texttt{""}]
\end{align*}
\]

In lazy semantics, since the result of `myfunc 1` is not needed to determine the value of `result` our `print` statement is never evaluated! Similar issues would occur if instead of `print` we modified a global variable, or wrote to a pointer, or interacted with a file, or any other computation with a side effect. While technically well-defined, this behavior makes it very difficult to write correct programs that perform side effects. To avoid this issue, Haskell chooses the “nuclear option”: it bans side effects altogether (a.k.a., requires that all code is pure).\(^9\)

**Monads**

Clearly banning all side effects is not tenable: while theoretically convenient, a language that is unable to interact with the environment is near-useless. Thus, we’ll have to come up with a way to emulate effectful programs in a pure language. The core of the issue is sequencing: given a set of value computations and a set of side effects, in a lazy language the relation between the computation steps and the ordering of side effects

---

\(^6\)For `myListProduct` of course this only holds if the head of the list is 0: in general `myListProduct` will only terminate on lists where that either (1) have a finite length, or (2) have a 0 element within a prefix list of finite length.

\(^7\)See lecture 3 for the definition of these terms

\(^8\)i.e., \(l; r\) should be read as “execute \(l\), wait for \(l\) to complete, and then execute \(r\)”

\(^9\)This is admittedly an oversimplification. Keep reading for the full story.
is in general unclear. Fortunately, in lecture 9 we covered a mathematical construct that gives us exactly this feature: monads.

**Monad Lecture Recap**

*The following is a brief recap of the relevant parts of lecture 9. For more information, see the lecture 9 slides.*

Recall that a monad is a type polymorphic over a single type variable, combined with two corresponding operations (which are different for each monad): bind (often denoted by >>=) and return. return lifts a standard Haskell value to a monadic value while bind allows us to sequentially compose monads. More precisely, for any monad m, return and bind have the following type signatures:

\[
\text{return} :: a \rightarrow m a \\
\text{bind} :: m a \rightarrow (a \rightarrow m b) \rightarrow m b
\]

bind’s signature provides some useful insight into its behavior: since bind is polymorphic over a and b, at the very least bind must unwrap the first argument (transforming the m a into an a) and then pass the unwrapped a into the second argument (a \rightarrow m b), yielding a value of type m b. This is one of bind’s two main contributions: unwrapping. The second is sequencing: to understand, let’s consider the following:

\[
\text{data M a -- declare a type M but ignore its constructors} \\
\text{return} :: a \rightarrow M a \\
\text{return} = \text{undefined} \\
\text{bind} :: M a \rightarrow (a \rightarrow M b) \rightarrow M b \\
\text{bind} = \text{undefined}
\]

As we can see, using bind and return give us a way to compose two monadic actions together, and since the only way a b can be generated is by running f and then unwrapped using bind, this guarantees that f >>= g will first run f and then run g regardless of what f and g are. Even if f and g don’t use their input arguments, as was the issue in our initial version of print, we have to have evaluated f before we start on g.

Of course, sequential programs have a few other expected semantics, e.g.,

\[
\{ \\
\text{stmt1; } \\
\text{stmt2; } \\
\text{stmt3; }
\}
\]

should be equivalent to both

\[
\{ \\
\text{stmt1; } \\
\{ \\
\text{stmt2; } \\
\text{stmt3; }
\}
\}
\]

and
In other words, sequential composition should be associative. In addition, we would expect

```plaintext
{ stmt;
}
```

to be equivalent to both

```plaintext
{ ;
 stmt;
}
```

and

```plaintext
{ stmt;
 ;
}
```

These conditions cannot be derived from the types of `bind` and `return`, and so to be a valid monad some other *monad laws* must be satisfied:

1. \(((\text{stmt1} \gg\gg \text{stmt2}) \gg\gg \text{stmt3}) \equiv (\text{stmt1} \gg\gg (\text{stmt2} \gg\gg \text{stmt3}))\)
2. \((\text{return} \gg\gg \text{stmt}) \equiv (\text{stmt} \gg\gg \text{return}) \gg\gg \text{stmt}\)

Looking closely, we can see that these correspond to the expected semantics of sequential composition above.

Finally, we expect that any side effects in our sequential program are implicit, e.g., we would do

```plaintext
{ void *x, *y, *z;
  *x = f();
  *y = g();
  *z = h();
}
```

and expect that *x, *y, and *z can be accessed from within f, g, and h even though we don’t explicitly pass them in. More concretely, a sequential program should not require us to explicitly pass in all past results à la

```plaintext
{ void *x, *y, *z;
  *x = f();
  *y = g(x);
  *z = h(x, y);
}
```

This “implicit” dataflow is reflected in the type signature of `bind`:

\[
(\gg\gg) :: \text{m a} \to (\text{a} \to \text{m b}) \to \text{m b}
\]

If we read \text{m a} as “an effectful program producing a value of type \text{a}”, then we can see that the the “step” function (the second argument, typed \text{a} \to \text{m b}) can produce effects (the output type \text{m b}), but sees only the *result* of the previous effectful program (the input type \text{a}) and, critically, *is not explicitly passed any of the previous program’s effects*. Instead, composing the two sets of side effects hidden away in the \text{m a} and the \text{m b}
resulting from the “step” function is the responsibility of \texttt{bind}. Thus, every step in our monadic program

\textbf{Implementing Monads w/ Parameteric Polymorphism}

By now you should have a sense of how monads allow us to emulate the expected semantics of effectful

\begin{verbatim}
data M1 a
data M2 a

bindM1 :: M1 a -> (a -> M1 b) -> M1 b
bindM1 = undefined

returnM1 :: a -> M1 a
returnM1 = undefined

bindM2 :: M2 a -> (a -> M2 b) -> M2 b
bindM2 = undefined

returnM2 :: a -> M2 a
returnM2 = undefined

sequenceM1 :: (a -> M1 b) -> (b -> M1 c) -> (a -> M1 c)
sequenceM1 f g = \initial -> (((returnM1 initial) `bindM1` f) `bindM1` g)

sequenceM2 :: (a -> M2 b) -> (b -> M2 c) -> (a -> M2 c)
sequenceM2 f g = \initial -> (((returnM2 initial) `bindM2` f) `bindM2` g)

-- ... and on and on for each generic monad operation ...
\end{verbatim}

There exist a number of other sequential constructs that we’d like to implement which are generic with

\textbf{Implementing Monads w/ Virtual Tables}

Our ideal implementation of monads would look similar to inheritance in object-oriented languages (in general

\begin{verbatim}
template <template A>
struct Monad {
    virtual void monadReturn(
        A,
        Monad<A> *out
    ) = 0;

    template <typename A, typename B>
    virtual void monadBind(
        Monad<A> *,
        void (*)(A, Monad<B> *),
        Monad<B> *out
    ) = 0;
};

template <typename A>
struct MyMonadType : Monad {
    // ...
\end{verbatim}
virtual void monadReturn(
    A,
    MyMonadType<A> *out
) override {
    // ...
    *out = myResult;
    return;
}

template <typename B>
virtual void monadBind(
    MyMonadType<A> *,
    void (*)(A, MyMonadType<B> *),
    MyMonadType<B> *out
) override {
    // ...
    *out = myResult;
    return;
}

// ...
};

template <typename A, typename B, typename C>
M<C>(*)(A) monadSequence(
    void (*)(A, Monad<B> *),
    void (*)(B, Monad<C> *),
    void (**)(A, Monad<C> *)
) {
    // ...
}

// ...

where Monad is an abstract interface which is implemented by any number of actual monad instances, and then functions can be written to operate over any type that inherits from Monad. The underlying mechanism for this in object oriented languages is called virtual tables, a.k.a. vtables. When we define an abstract interface like Monad, the compiler internally declares a new struct type

template <typename A>
struct MonadVtable {
    void (*monadReturnImpl)(
        A,
        Monad<A> *
    );

    void (*monadBindImpl)(
        Monad<A> *,
        void (*)(A, Monad<B> *),
        Monad<B> *
    );
};

and declare a global instantiation of this MonadVtable struct for each subclass of Monad:

template <typename A>
MonadVtable<A> myMonadTypeVtable = {
    &MyMonadType<A>::monadReturn,
    &MyMonadType<A>::monadBind
};

and then we store a pointer to myMonadTypeVtable in every object of type MyMonadType. Thus, when we want to call monadReturn on some object that inherits from Monad, we follow that object’s pointer to it’s vtable, lookup the pointer to the type’s monadReturn implementation, and then call that function pointer. Essentially, we move the polymorphism from the type system (template in C++, forall in Haskell) to a runtime data value called a vtable.
We can use this same technique to implement ad-hoc polymorphism in Haskell. First we declare a datatype to represent the vtable:

```
data MonadVtable m = MonadVtable
  -- return
  (forall a. a -> m a)
  -- bind, a.k.a. (>>=)
  (forall a b. m a -> (a -> m b) -> m b)
```

```
getReturnFuncFrom :: MonadVtable m -> a -> m a
getReturnFuncFrom (MonadVtable return _) = return

getBindFuncFrom :: MonadVtable m -> m a -> (a -> m b) -> m b
getBindFuncFrom (MonadVtable _ bind) = bind
```

and then we can implement a >>= operation that supports every monad type:

```
monadSequence :: MonadVtable m
  -> (a -> m b)
  -> (b -> m c)
  -> (a -> m c)
monadSequence vtable f g =
  let return = getReturnFuncFrom vtable
      bind = getBindFuncFrom vtable
  in ((\initial -> (((return initial) `bind` f) `bind` g))
```

As an example, let’s implement the vtable approach for a very simple monad: Maybe. The Maybe monad represents a computation that can fail, essentially equivalent to throw/raise in imperative languages. First we have to declare the actual data representation of our monad:

```
data Maybe a
  -- either we've succeeded and returned a value of type a
  = Just a
  -- or an error has occurred
  | Nothing
```

Since a pure value/computation cannot fail, to lift a pure value/computation into a Maybe monad we would just wrap it in Just:

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

Now we can turn to implementing bind, which we’ll do in cases. Recall that the signature of bind is as follows:

```
bindForMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
bindForMaybe init step = undefined
```

First, we’ll consider the case that init is Nothing, i.e., that the previous computation failed. In that case, any computation that comes after it should also fail, so we have

```
bindForMaybe Nothing _ = Nothing
```

If the previous computation succeeded (i.e., init == Just x) we can unwrap the Just to get the value returned by the previous sequential steps. Then we pass that value to step and obtain two further cases: either steps fails (step x == Nothing) or step succeeds (step x = Just y). In the first case, if step fails then the whole computation should fail, so in this case we return Nothing. If step succeeds, we just return the resulting value. We can implement these cases as follows:

```
bindForMaybe (Just x) step = case step x of
  Just y -> Just y
  Nothing -> Nothing
```

Putting it all together, we get

---

10 Technically it’s equivalent to throw where there is no ability to choose an exception type: a function either throws or it doesn’t.
bindForMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
bindForMaybe Nothing _ = Nothing
bindForMaybe (Just x) step = case step x of
    Just y -> Just y
    Nothing -> Nothing

Now that we have defined both return and bind, we can define the MonadVtable instance:

monadVtableForMaybe :: MonadVtable Maybe
monadVtableForMaybe = MonadVtable returnForMaybe bindForMaybe

and then we can use any of our generic monadic operations on Maybe!

myDivideBy :: Float -> Float -> Maybe Float
myDivideBy 0.0 _ = Nothing
myDivideBy denom numer = Just (numer / denom)

myInitialValue :: Float
myInitialValue = 4.0

divideBy2 :: Float -> Maybe Float
divideBy2 = myDivideBy 2.0

divideBy0 :: Float -> Maybe Float
divideBy0 = myDivideBy 0.0

printResult :: Maybe Float -> String
printResult Nothing = "<error>"
-- show is a builtin that converts a value to a String
printResult (Just x) = show x

printResult (Just myInitialValue)
"4.0"

printResult Nothing
"<error>"

shouldSucceed :: Float -> Maybe Float
shouldSucceed =
    let (>>=) = monadSequence monadVtableForMaybe
    in (divideBy2 >>= divideBy2)

printResult (shouldSucceed myInitialValue)
"1.0"

shouldFail1 :: Float -> Maybe Float
shouldFail1 =
    let (>>=) = monadSequence monadVtableForMaybe
    in (divideBy0 >>= divideBy2)

printResult (shouldFail1 myInitialValue)
"<error>"

shouldFail2 :: Float -> Maybe Float
shouldFail2 =
    let (>>=) = monadSequence monadVtableForMaybe
    in (divideBy2 >>= divideBy0)

printResult (shouldFail2 myInitialValue)
"<error>"
Typeclasses

The vtable approach shown above works even for larger programs\(^{11}\) but it has the downside that we need to explicitly pass around our vtables. This is usually safe as the typechecker will catch any bugs where you pass the either the wrong vtable or the vtable for the wrong type, but quickly gets cumbersome as we use more and more vtables. This is exactly why in C++ the vtables are stored as a pointer on each object: passing them around quickly gets annoying! Since most types only implement a single monad instance,\(^{12}\) it seems that the compiler should be able to infer the vtable that should be used and automatically pass it around. In fact, this feature, called *typeclasses* does exist in Haskell! Below we’ll see how to convert `MonadVtable` and our corresponding instance for `Maybe` to an equivalent representation that uses typeclasses. The conversion process is essentially just syntactic: typeclasses are just syntactic sugar for vtables.

First, we’ll need to convert `MonadVtable` to a *typeclass declaration* as follows:

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

which looks very similar to our original vtable code:

```haskell
data MonadVtable m = MonadVtable
  -- return
  (forall a. a -> m a)
  -- bind, a.k.a. (>>=)
  (forall a b. m a -> (a -> m b) -> m b)
```

Then we need to define an instance of `Monad` for `Maybe`:

```haskell
instance Monad Maybe where
  return = returnForMaybe
  (>>=) = bindForMaybe
```

which also looks very similar to the corresponding vtable code

```haskell
monadVtableForMaybe :: MonadVtable Maybe
monadVtableForMaybe = MonadVtable returnForMaybe bindForMaybe
```

Finally, we need to know how to pass a vtable to a function. Instead of explicitly passing the vtable à la

```haskell
monadSequence :: MonadVtable m
  -> (a -> m b)
  -> (b -> m c)
  -> (a -> m c)
monadSequence vtable f g =
  let return = monadReturn vtable
  bind = monadBind vtable
  in  (\initial -> (((return initial) >>= f) >>= g))
```

we now provide a *type constraint*:

```haskell
(>>=) :: (Monad m)
  => (a -> m b)
  -> (b -> m c)
  -> (a -> m c)
(>>=) f g = (\initial -> (((return initial) >>= f) >>= g))
```

Note that now we can just directly use `return` and `>>=` and the compiler will automatically fetch the corresponding vtable instances for us.

And that’s all! The conversion is very simple, and what’s going on at runtime is essentially the same as in the vtable code.

\(^{11}\) In fact, some people have even argued that the vtable approach is often superior to Haskell’s typeclasses

\(^{12}\) In the case that we have multiple instances for a type, we can define wrapper types to disambiguate which instance should be used, e.g., `All` and `Any`
Using Monads

For the rest of this lecture we’ll be using the typeclass interface, but with minor syntactic changes everything below also applies to our vtable implementation. First we’ll walk through a few example monads and write some simple programs with them, and then we’ll examine some convenient syntactic sugar (do-notation) that Haskell provides to make monadic computations look visually more like the sequential programs they’re emulating (but remember it’s really all just monads and functions underneath—there’s no magic here!).

Example Monad Instances

Maybe We’ll start out with the Maybe monad as we’ve already encountered it in the section on implementing monads. Recall that we define the Maybe type as follows:

```
data Maybe a
  -- either we've succeeded and returned a value of type a
  = Just a
  -- or an error has occurred
  | Nothing
```

and that the monad instance is

```
instance Monad Maybe where
  return :: a -> Maybe a
  return x = Just x

  (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  (>>=) Nothing _ = Nothing
  (>>=) (Just pre) step = case step pre of
    Just post -> Just post
    Nothing -> Nothing
```

and with these, we can begin to write monadic programs that emulate the ability of a computation to fail:

```
mySafeHead :: MyGenericList e -> Maybe e
mySafeHead (GenericCons head _) = Just head
mySafeHead GenericEmptyList = Nothing

mySafeTail :: MyGenericList e -> Maybe (MyGenericList e)
mySafeTail (GenericCons _ tail) = Just tail
mySafeTail GenericEmptyList = Nothing

printIntResult :: Maybe Int -> String
printIntResult (Just x) = show x
printIntResult Nothing = "<error>"

printIntList :: MyGenericList Int -> String
printIntList (GenericCons head tail) = show head ++ ":" ++ printIntList tail
printIntList GenericEmptyList = "[]"

printIntListResult :: Maybe (MyGenericList Int) -> String
printIntListResult (Just l) = printIntList l
printIntListResult Nothing = "<error>"

intList0 :: MyGenericList Int
intList0 = GenericEmptyList

intList1 :: MyGenericList Int
intList1 = GenericCons 3 GenericEmptyList

intList2 :: MyGenericList Int
intList2 = GenericCons 5 (GenericCons 2 GenericEmptyList)

intListInfinity :: MyGenericList Int
intListInfinity = let countUpFrom = (\x -> GenericCons x (countUpFrom (x + 1)))
  in countUpFrom 2
```
Using `bind` we can compose computations that can fail, e.g.,

```haskell
myAtIdx1 :: MyGenericList e -> Maybe e
myAtIdx1 l = (mySafeTail l) >>= mySafeHead
```

```
printIntResult (myAtIdx1 intList0)
"<error>"
```

```
printIntResult (myAtIdx1 intList1)
"<error>"
```

```
printIntResult (myAtIdx1 intList2)
"2"
```

```
printIntResult (myAtIdx1 intListInfinity)
"3"
```

Using `bind` for unary functions (i.e., functions that take a single input) is trivial, but it’s not quite as clear how to use `bind` for functions with more inputs. The answer here is to use nested lambdas, e.g.,

```haskell
myPairHead :: MyGenericList e -> MyGenericList e' -> Maybe (e, e')
myPairHead l1 l2 = (mySafeHead l1) >>= (\h1 -> (mySafeHead l2) >>= (\h2 -> return (h1, h2)))
```

```
printPairResult :: Maybe (Int, Int) -> String
printPairResult (Just (l, r)) = "(" ++ show l ++ ", " ++ show r ++ ")"
printPairResult Nothing = "<error>"
```

```
printPairResult (myPairHead intList0 intList0)
"<error>"
```

```
printPairResult (myPairHead intList0 intList1)
"<error>"
```

```
printPairResult (myPairHead intList1 intList0)
"<error>"
```

```
printPairResult (myPairHead intList1 intList1)
"(3, 3)"
```

For many monads it’s also convenient to define some core helper functions that express the core capabilities of the monadic type. For example, in a `Maybe` computation there are two capabilities: we can either successfully compute a value (i.e., `return`) or we can fail (currently denoted by `Nothing`). However, if we do not want to
rely on the underlying `Maybe` data structure we could define the following:

```haskell
fail :: Maybe a
fail = Nothing

didFail :: Maybe a -> Bool
didFail (Just _) = False
didFail Nothing = True

not :: Bool -> Bool
not True = False
not False = True

didSucceed :: Maybe a -> Bool
didSucceed = not . didFail
```

and then could write programs at a higher level, e.g.,

```haskell
myDivide :: Float -> Float -> Maybe Float
myDivide numer denom = if denom == 0.0
  then fail
  else return (numer / denom)
```

which start to look a lot like typical programs in impure sequential languages like C and Python. In the case of `Maybe` this has limited utility, but we’ll see in the next couple sections how this can be quite useful for more complicated monads.

**Reader**  The `Reader` monad implicitly passes an additional read-only “state” through the computation. This is commonly used for explicitly propagating a user-defined application configuration throughout an application. As we’ll be embedding this *read-only stateful computation* into pure Haskell code, we’ll first have to figure out how to express a read-only stateful computation as a Haskell type. Conceptually, a read-only stateful computation takes in an initial state as input, and returns a computed pure value based on that state. Note that we do not return the state as it is unnecessary: our state value is constant throughout the computation. Thus, we’ll implement the `Reader` monad as follows:

```haskell
data Reader s a = Reader (s -> a)

instance Monad Reader where
  -- ...

<interactive>:615:16: error:
  Expecting one more argument to Reader
  Expected kind * -> *, but Reader has kind * -> * -> *
  In the first argument of Monad, namely Reader
  In the instance declaration for Monad Reader
```

but we get a typechecking error! Recall that a monad is required to be a type polymorphic over one variable, but `Reader` is polymorphic over two. Thus, `Reader` itself is not actually a monad. Instead, `Reader` is a family of monads, where for every state type `s` we get a unique monad type `Reader s` whose monad instance we can define as

```haskell
instance Monad (Reader s) where
  return :: a -> Reader s a
  return x = Reader (\_ -> x)

  (>>=) :: Reader s a -> (a -> Reader s b) -> Reader s b
  (>>=) (Reader initialComputation) chooseNextComputation = Reader ((\initialState ->
    let computedByInitial = initialComputation initialState
    in nextComputation computedByInitial)
```

In theory, we’d then declare the monad instance as

```haskell
instance Monad Reader where
  -- ...

<interactive>:615:16: error:
  Expecting one more argument to Reader
  Expected kind * -> *, but Reader has kind * -> * -> *
  In the first argument of Monad, namely Reader
  In the instance declaration for Monad Reader
```

but we get a typechecking error! Recall that a monad is required to be a type polymorphic over one variable, but `Reader` is polymorphic over two. Thus, `Reader` itself is not actually a monad. Instead, `Reader` is a family of monads, where for every state type `s` we get a unique monad type `Reader s` whose monad instance we can define as

```haskell
instance Monad (Reader s) where
  return :: a -> Reader s a
  -- to make this more clear we unwrap Reader, yielding "return :: a -> (s -> a)"
  return x = Reader (\_ -> x)

  (>>=) :: Reader s a -> (a -> Reader s b) -> Reader s b
  -- or unwrapped: "(>>=) :: (s -> a) -> (a -> (s -> b)) -> (s -> b)
  (>>=) (Reader initialComputation) chooseNextComputation = Reader ((\initialState ->
    let computedByInitial = initialComputation initialState
    in nextComputation computedByInitial)
```
As expected, `return` lifts a pure value into a read-only stateful computation that behaves like that pure value, i.e., returns the same pure value regardless of the state. `bind` is also quite simple: given an initial read-only stateful computation `initialComputation` and the next step in our sequential program `nextComputation`, we say that for any initial state `s` that this sequential program is given we first run the initial computation with the given state (i.e., `initialComputation initial\State`) and then run the next computation, also passing it the initial state (i.e., `nextComputation initial\State`).

Now we can use `Reader` to implement a program that, for example, returns a greeting based on a user-configuration:

```haskell

data UserConfig =
  UserConfig
    -- first name
    String
    -- last name
    String
    -- title
    String
    -- age
    Int

firstName :: UserConfig -> String
firstName (UserConfig fstName _ _ _) = fstName

lastName :: UserConfig -> String
lastName (UserConfig _ lstName _ _) = lstName

title :: UserConfig -> String
title (UserConfig _ _ t _) = t

age :: UserConfig -> Int
age (UserConfig _ _ _ a) = a

getFirstName :: Reader UserConfig String
getFirstName = Reader (\s -> firstName s)

getLastName :: Reader UserConfig String
getLastName = Reader (\s -> lastName s)

title :: Reader UserConfig String
getTitle = Reader (\s -> title s)

getAge :: Reader UserConfig Int
getAge = Reader (\s -> age s)

getUserFormalName :: Reader UserConfig String
getUserFormalName = getFirstName >>= (\firstName ->
  getLastName >>= (\lastName ->
    getTitle >>= (\title ->
      return (title ++ " " ++ firstName ++ " " ++ lastName)))))

ageDependentGreeting :: Int -> String
ageDependentGreeting age = if age < 5
  then "Aren't you cute!"
  else
    (if age < 15
      then "Look at you, all grown up!"
      else
        (if age < 80
          then "Another day another dollar, am I right?"
          else
            (if age < 110
              then "How's retirement?"
              else "Wow you're old!"))))

getUserGreeting :: Reader UserConfig String
```

20
getUserGreeting = getUserFormalName >>= (\formalName ->
    getAge >>= (\age ->
      let appropriateGreeting = ageDependentGreeting age
      in return ("Good day to you," ++ formalName ++ "! " ++
        appropriateGreeting)))

To run this we now just need a way to extract the pure function that represents this read-only stateful
computation, i.e.,

extractPureFunction :: Reader s a -> (s -> a)
extractPureFunction (Reader f) = f

and then we can run our program:

myUser :: UserConfig
myUser = UserConfig "Colin" "Unger" "best TA ever" 25

pureGreetUser :: UserConfig -> String
pureGreetUser = extractPureFunction getUserGreeting

pureGreetUser myUser
"Good day to you, best TA ever Colin Unger! Another day another dollar, am I right?"

As with Maybe, we can also define a couple helper functions that capture the essence of what the Reader
monad supports so we don’t have to depend directly on the underlying implementation of the Reader data

getImplicit :: Reader s s
getImplicit = Reader (\s -> s)

Now we can define, for example getLastName without directly using the Reader type constructor:

getLastName :: Reader UserConfig String
getLastName = getImplicit >>= (\s -> return (lastName s))

Writer Writer is the opposite of Reader: instead of a read-only stateful computation it emulates a write-only
(or more accurately append-only) stateful computation. The canonical use for Writer is logging, where we
want our computations to be able to append messages to an implicitly tracked log of messages. Conceptually,
we can model the essence of an append-only stateful computation as a tuple of a computed value a and the
log l produced by that computation, i.e., (a, l). Thus, we define our Writer monad as follows:

data Writer s a = Writer (s, a)

Then we’d turn to defining our monad instance. First we’ll start with return:

instance Monad (Writer s) where
  return :: a -> Reader s a
  -- or unwrapped: "return :: a -> (s, a)"
  return v = Writer (error "What goes here?", v)

  (>>=) = undefined -- ignore bind for now

but it’s unclear what we’d put for the log: a pure computation should not log anything, so we’d expect to
put something like []. However, that would require that s be a list, and we’ve placed no such requirement.
We could add this requirement, but there are things we could log that aren’t just messages, e.g.,

type MemoryUsageLogger x = Writer Int x

13Since for Reader, bind has type Reader s a -> (a -> Reader s a) -> Reader s a, our function to choose the
next computation step (the second parameter to bind, i.e., the one with type a -> Reader s a) is only passed the value type
a and not the state s.
It turns out that the most general structure that provides the expected behavior is a **monoid**, which is defined as: a set of objects \( \text{Obj} \), an associative binary operation \( \text{mappend} \), and an identity element \( \text{mempty} \in \text{Obj} \). In Haskell we can represent this with the typeclass

```haskell
class Monoid t where
  mempty :: t
  mappend :: t -> t -> t

  -- shorthand for mappend
  (<> :: Monoid t => t -> t -> t
  (<> = mappend
```

subject to the following laws:

1. for any \( x \in \text{Obj} \), \( x <> \text{mempty} = \text{mempty} <> x = x \), i.e., \( \text{mempty} \) forms an identity under \( \text{mappend} \).
2. for any \( x, y, z \in \text{Obj} \), \( x <> (y <> z) = (x <> y) <> z \), i.e., \( \text{mappend} \) is associative.

We'll skip a detailed explanation of why monoids provide the necessary structure for the state in \( \text{Writer} \), but a good source of intuition is to notice that the **Monoid** laws look very similar to the **Monad** laws, so we can expect that using **Monoid** will maintain the monadic structure of **Writer**.

Now we can declare the correct monad instance for **Writer**:

```haskell
instance (Monoid s) => Monad (Writer s) where
  return :: a -> Writer s a
  return x = Writer (mempty, x)

  (>>=) :: Writer s a -> (a -> Writer s b) -> Writer s b
  (>>=) (Writer initialComputation) chooseNextStep =
    let (initialLogs, initialReturnValue) = initialComputation
    (Writer (nextLogs, nextReturnValue)) = chooseNextStep initialReturnValue
    in Writer (initialLogs <> nextLogs, nextReturnValue)
```

and then we can implement the two **Writer** usages above. First, appending strings to a log:

```haskell
instance Monoid [a] where
  mempty = []
  mappend l r = l ++ r

type MessageLogger a = Writer [String] a
type NetworkResponse = String

logMessage :: String -> MessageLogger ()
logMessage msg = Writer ([msg], ())

length :: [a] -> Int
length (_:rest) = 1 + length rest
length [] = 0

mockNetworkCommunication :: String -> MessageLogger NetworkResponse
mockNetworkCommunication packet =
  logMessage ("Successfully sent packet of size " ++ show (length packet)) >>= (\_ -> return "hello to you too!")

sendNetworkPacket :: String -> MessageLogger ()
sendNetworkPacket packet = logMessage "Starting network communication" >>= (\_ ->
  mockNetworkCommunication packet >>= (\response ->
    logMessage ("Received response: " ++ response) >>= (\_ ->
      logMessage "Finished network communication!")))

combineEntries :: [String] -> String
combineEntries (h:[]) = h
combineEntries (h:rest) = h ++ "; " ++ combineEntries rest
combineEntries [] = ""

printLogs :: MessageLogger a -> String
printLogs (Writer (log, _)) = combineEntries log
```
printLogs (sendNetworkPacket "hello network!"
"Starting network communication; Successfully sent packet of size 14; Received response: hello to you too!; Finished network communication!

Second, memory usage:

type MemoryUsage = Int

instance Monoid MemoryUsage where
  mempty = 0
  mappend l r = l + r

type MemoryUsageTracker a = Writer MemoryUsage a
data Allocation = Allocation Int

increaseMemoryUsage :: Int -> MemoryUsageTracker ()
increaseMemoryUsage amt = Writer (amt, ()

decreaseMemoryUsage :: Int -> MemoryUsageTracker ()
decreaseMemoryUsage amt = Writer (- amt, ()

trackMalloc :: Int -> MemoryUsageTracker Allocation
trackMalloc size = increaseMemoryUsage size >>= (\_ _> return (Allocation size))

trackFree :: Allocation -> MemoryUsageTracker ()
trackFree (Allocation size) = decreaseMemoryUsage size

exampleProgram :: MemoryUsageTracker ()
exmapleProgram = trackMalloc 12 >>= (\block1 _> trackMalloc 15 >>= (\block2 _> trackMalloc 8 >>= (\block3 _> trackMalloc 9 >>= (\block4 _> trackFree block1 >>= (\_ _> return (Allocation size))

getMemoryUsage :: MemoryUsageTracker a -> MemoryUsage
getMemoryUsage (Writer (s, _)) = s

printMemoryUsage :: MemoryUsageTracker a -> String
printMemoryUsage tracker = "Memory usage: " ++ show (getMemoryUsage tracker)

printMemoryUsage exampleProgram
"Memory usage: 24"

As with Maybe and Reader, we can also define additional functions to avoid depending on the underlying implementation of Writer:

output :: s -> Writer s ()
output v = Writer (v, ()

and thus can rewrite, for example, logMessage as

logMessage :: String -> MessageLogger ()
logMessage msg = output [msg]

State You may notice an issue with the MemoryUsage implementation above: deallocating a block multiple times allows the memory usage to go negative! Since the Writer monad provides no way to access the state from within the computation, to prevent freeing an already freed block we’ll need a monad that combines Reader and Writer, i.e., models a read-write stateful computation. We call this monad State and can implement it as follows:

data State s a = State (s -> (a, s))

Looking at this closely, we can see the similarity to both Reader and Writer: the type of State’s field
(s -> (a, s)) resembles a combination Reader (s -> a) and Writer ((s, a)). We can read the type of State's field as “a function from an initial state to a resulting value and a resulting state”, which suggests the monad instance below:

```haskell
instance Monad (State s) where
  return :: a -> State s a
  -- or unwrapped: "return :: a -> (s -> (a, s))"
  -- we would expect a pure computation to leave the state untouched,
  -- so return creates a function that returns the given value while passing
  -- the state unmodified
  return x = State (\s -> (x, s))

  (>>=) :: State s a -> (a -> State s b) -> State s b
  -- or unwrapped: "(>>=) :: (s -> (a, s)) -> (a -> (s -> (b, s))) -> (s -> (b, s))
  -- given an initial computation and a function to determine the next computation, we
  -- expect (>>=) to first calculate the value and state returned from the initial computation
  -- use the returned value to compute the next computation, and then to pass the returned state
  -- to the next computation.
  -- Note that none of these values actually contain the initial state: the State monad
  -- constructs
  -- a function that is contingent on the initial state passed in rather than constructing the
  -- final
  -- state directly
  (>>=) (State initialComputation) chooseNextStep = State (\initialState ->
    let (intermediateValue, intermediateState) = initialComputation initialState
    (State nextComputation) = chooseNextStep intermediateValue
    in nextComputation intermediateState)
```

Unlike Reader instead of passing on the initial state, State's bind passes on the modified state (intermediateState) that results from the initial computation. Note that we no longer need the type constraint Monoid s that was required for Writer as we allow State not only to append to the state value, but also to overwrite it. This time we'll start by implementing some primitive functions for State, and then will use them in our improved heap implementation:

```haskell
getState :: State s s
getState = State (\s -> (s, s))

setState :: s -> State s ()
setState s = State (\s -> ((), s'))

modifyState :: (s -> s) -> State s ()
modifyState f = getState >>= (\s -> setState (f s))

fromState :: (s -> a) -> State s a
fromState f = getState >>= (\s -> return (f s))

extractPureFunction :: State s a -> (s -> (a, s))
extractPureFunction (State f) = f
```

Now we can turn to implementing a safer version of our MemoryUsage computation: we will store a list of allocations, and whenever we call free we will only decrement the memory usage if the provided block is currently allocated. First we'll need to generalize our example program from the last program, in addition to adding some for the error cases. Since each of our heap implementations will have different behaviors but expose the same interface (malloc and free) we can generalize our example programs over a custom typeclass. The only difficulty is that each of our different interpreters may have a custom Block representation, so we'll inform the compiler of this using the following:

```haskell
class Monad m => HeapLangInterpreter m b | m -> b where
  malloc :: Int -> m b
  free :: b -> m ()
```

Don’t worry too much about understanding this exact syntax—just read it as “declare a new typeclass called HeapLangInterpreter over a type m which requires m to be a Monad and has an additional type b (our

---

14If you're curious about it, see the documentation on functional dependencies here
block type) that is uniquely determined by the identity of type \( m \). With this we can now define an instance for our existing `MemoryUsageTracker`:

```
instance HeapLangInterpreter (Writer MemoryUsage) Allocation where
  malloc = trackMalloc
  free = trackFree
```

Now we can define our heap programs without worrying about how they’ll be interpreted:

```
exampleValidProgram :: HeapLangInterpreter m b => m ()
exampleValidProgram = malloc 12 >>= (\block1 ->
  malloc 15 >>= (\block2 ->
    malloc 8 >>= (\block3 ->
      free block1 >>= (\_ ->
        malloc 9 >>= (\block4 ->
          free block3 >>= (\_ ->
            return ())))))

exampleDoubleFree :: HeapLangInterpreter m b => m ()
exampleDoubleFree = malloc 12 >>= (\block1 ->
  malloc 15 >>= (\block2 ->
    malloc 8 >>= (\block3 ->
      free block1 >>= (\_ ->
        malloc 9 >>= (\block4 ->
          free block1 >>= (\_ ->
            free block1 >>= (\_ ->
              free block1 >>= (\_ ->
                free block1 >>= (\_ ->
                  free block3 >>= (\_ ->
                    free block1 >>= (\_ ->
                      return ())))))))))
```

and since `MemoryUsageTracker` is an instance of `HeapLangInterpreter`, we can confirm that our existing interpreter is insufficient to handle `exampleDoubleFree`:

```
printMemoryUsage exampleDoubleFree
"Memory usage: -36"
```

To fix this, we’ll implement a new interpreter monad using `State` that keeps track of which blocks it has allocated and does not reduce memory usage for any double frees that occur. First, we’ll need a way of tracking which block is which—in our existing implementation we only track the size of allocations, not their identity. To do this, we’ll define a new `Block` type with an additional ID field with type `Int`:

```
data Block = Block Int Allocation

blockSize :: Block -> Int
blockSize (Block _ (Allocation size)) = size

blockID :: Block -> Int
blockID (Block idNum _) = idNum

-- Haskell uses the Eq typeclass to allow types to register
-- custom equality operators. Without an Eq implementation
-- a type is not equality-comparable
instance Eq Block where
  (==) :: Block -> Block -> Bool
  (==) (Block lIdNum (Allocation lSize)) (Block rIdNum (Allocation rSize)) =
    (lIdNum == rIdNum) && (lSize == rSize)
```

and then we can define our representation of our heap’s state along with some getters and setters:

```
data HeapState = HeapState
  -- currently allocated blocks
  [Block]
  -- next free block id
  Int
```
allocated :: HeapState -> [Block]
allocated (HeapState inUse _) = inUse

setAllocated :: [Block] -> HeapState -> HeapState
setAllocated bs (HeapState _ i) = HeapState bs i

getHeapStateID :: HeapState -> Int
getHeapStateID (HeapState _ i) = i

setHeapStateID :: Int -> HeapState -> HeapState
setHeapStateID new (HeapState inuse _) = HeapState inuse new

Since we’ll need to both read from our HeapState (to see which blocks are allocated and which id to use next) and write to it (to update these values on allocation and deallocation), we’ll use a State monad:

```haskell
type WithHeap a = State HeapState a
```

To make it easier to refactor our HeapState later, we’ll implement our core functions so they can only see the part of the State they need using the following helper function:

```haskell
maskState :: (s -> s') -- a getter from the full state to the isolated state
         -> (s' -> s -> s) -- a setter to update the full state with the new isolated state
         -> State s s' b -- our isolated stateful computation
         -> State s b -- our unisolated stateful computation
maskState mask unmask f = let f' = extractPureFunction f
                          in State (
s -> let masked = mask s
                                    (b, s') = f' masked
                                    in (b, unmask s' s))
```

```haskell
class HasNextBlockId s where
    getNextBlockId :: s -> Int
    setNextBlockId :: Int -> s -> s

instance HasNextBlockId HeapState where
    getNextBlockId = getHeapStateID
    setNextBlockId = setHeapStateID

class HasInUseBlockList s where
    getInUse :: s -> [Block]
    setInUse :: [Block] -> s -> s

instance HasInUseBlockList HeapState where
    getInUse = allocated
    setInUse = setAllocated
```

Now we can define masks for each member of HeapState:

```haskell
idMasked :: HasNextBlockId s => State Int a -> State s a
idMasked = maskState getNextBlockId setNextBlockId

inUseMasked :: HasInUseBlockList s => State [Block] a -> State s a
inUseMasked = maskState getInUse setInUse
```

and then we can define some basic stateful actions we’ll need. First we’ll need a way to update our unique ID whenever we allocate a new block:

```haskell
incrementID :: HasNextBlockId s => State s Int
incrementID = idMasked (modifyState (+ 1) >>= (
_) -> getState))
```

Then we’ll need a way to mark blocks as either “in use” (by placing it in our heaps “allocated” list):

```haskell
markBlockInUse :: HasInUseBlockList s => Block -> State s ()
markBlockInUse b = inUseMasked (modifyState ([b] -> b:bs))
```

or “freed” (by removing it from the “allocated” list):
map :: (a -> b) -> [a] -> [b]
map f (head:rest) = (f head) : (map f rest)
map _ [] = []

mconcat :: Monoid t => [t] -> t
mconcat [] = mempty
mconcat (head:rest) = head `mappend` (mconcat rest)

foldMap :: Monoid t => (a -> t) -> [a] -> t
foldMap f = mconcat . map f

removeAll :: Eq a => a -> [a] -> [a]
removeAll x = foldMap (\x' -> if x' == x then [] else [x'])

markBlockFree :: HasInUseBlockList s => Block -> State s ()
markBlockFree b = inUseMasked (modifyState (removeAll b))

We'll also want a way to check if a block is marked as “in use”:
contains :: Eq a => a -> [a] -> Bool
contains x (h:rest) = if h == x then True else contains x rest
contains x [] = False

isInUse :: HasInUseBlockList s => Block -> State s Bool
isInUse = inUseMasked . fromState . contains

With all of that defined, implementing our malloc and free functions is quite simple:
mallocSafe :: (HasInUseBlockList s, HasNextBlockId s) => Int -> State s Block
mallocSafe size = incrementID >>= (\blockId ->
  let block = Block blockId (Allocation size)
      in markBlockInUse block >>\ (\_ -> return block))

-- our HeapLangInterpreter typeclass does not require
-- us to return whether or not a free succeeds, but since
-- we have that information here we return it in case it
-- is still helpful in some other case
freeSafe :: (HasInUseBlockList s, HasNextBlockId s) => Block -> State s Bool
freeSafe b = isInUse b >>= (\c ->
  markBlockFree b >>\ (\_ ->
    return c))

and then we can define our HeapLangInterpreter instance:

instance HeapLangInterpreter (State HeapState) Block where
  malloc = mallocSafe
  free size = (freeSafe size) >>\ (\_ -> return ()),

Now we write a few small helper functions to compute the memory usage for our new heap representation:

memoryUsage :: WithHeap Int
memoryUsage = inUseMasked (fromState (foldMap blockSize))

initialHeapState :: HeapState
initialHeapState = HeapState [] 0

showMemoryUsage :: WithHeap a -> String
showMemoryUsage prog = let f = extractPureFunction (prog >>\ (\_ -> memoryUsage))
  (usage, finalHeap) = f initialHeapState
  in "Memory usage: " ++ show usage

and we can see that now we handle the test programs correctly!

showMemoryUsage exampleValidProgram
"Memory usage: 24"

showMemoryUsage exampleDoubleFree

```
Phew! That was a lot of code.

Now our memory tracking is resilient to double frees, but only by making free fail silently when a double free occurs. Let's now write an additional logging mechanism so we can tell where a double free has occurred! First we'll update our `HeapState` with an additional log:

```haskell
data HeapStateWithLog = HeapStateWithLog
  -- our list of allocated blocks
  [Block]
  -- our next free block id
  Int
  -- a log of heap states where we performed a double free
  -- along with what block we tried to free
  [(HeapState, Block)]

instance HasNextBlockId HeapStateWithLog where
  getNextBlockId (HeapStateWithLog _ idNum _) = idNum
  setNextBlockId idNum (HeapStateWithLog inUse _ log) =
    HeapStateWithLog inUse idNum log

instance HasInUseBlockList HeapStateWithLog where
  getInUse (HeapStateWithLog inUse _ _) = inUse
  setInUse inUse (HeapStateWithLog _ idNum log) =
    HeapStateWithLog inUse idNum log

Now we just need to add a function to let us add new log messages:

```haskell
append :: a -> [a] -> [a]
append x rest = rest ++ [x]

getLog :: HeapStateWithLog -> [(HeapState, Block)]
getLog (HeapStateWithLog _ _ log) = log

setLog :: [(HeapState, Block)] -> HeapStateWithLog -> HeapStateWithLog
setLog log (HeapStateWithLog inUse _ idNum log) =
  HeapStateWithLog inUse _ idNum log

logMasked :: State [(HeapState, Block)] a -> State HeapStateWithLog a
logMasked = maskState getLog setLog

getCurrentHeapState :: State HeapStateWithLog HeapState
getCurrentHeapState = getState >>= (\(HeapStateWithLog inUse _ idNum log) ->
  return (HeapState inUse idNum))

logDoubleFree :: Block -> State HeapStateWithLog ()
logDoubleFree b = getCurrentHeapState >>= (\st -> logMasked (modifyState (append (st, b)))))

and then we can add a logged version of `free`:

```haskell
freeLogged :: Block -> State HeapStateWithLog ()
freeLogged b = freeSafe b >>= (\succeeded ->
  if succeeded then return () else logDoubleFree b)

instance HeapLangInterpreter (State HeapStateWithLog) Block where
  malloc = mallocSafe
  free = freeLogged

and then we can check that we successfully detect the double frees in `exampleDoubleFree`:

```haskell
length :: [a] -> Int
length = foldMap (\_ -> 1)

initialLoggedHeapState :: HeapStateWithLog
initialLoggedHeapState = HeapStateWithLog [] 0 []

getNumDoubleFrees :: State HeapStateWithLog () -> Int
IO  If you’ve been paying close attention, you may have noticed that I haven’t quite delivered what I promised: I told you that monads would let us perform effectful computations, but so far all I’ve done is emulate effectful computations using pure computations. Nothing we’ve covered so far would let us print to the terminal, or talk over the network, or modify the filesystem, or otherwise affect the external world. This is where the IO monad comes in.

At a surface level, the IO monad provides the ability to perform these system interactions. For example, we have

```haskell
putStrLn :: String -> IO ()
```

which lets us print a line to the terminal, e.g.,

```haskell
putStrLn "hello world!"
```

```
hello world!
```

In fact, IO provides a number of functions that let us interact with the outside world:

```haskell
openFile :: FilePath -> IOMode -> IO Handle
hSeek :: Handle -> SeekMode -> Integer -> IO ()
hWaitForInput :: Handle -> Int -> IO Bool
hGetLine :: Handle -> IO String
getLine :: IO String
-- ... and many more
```

and while we can use these functions similarly to any other monad, e.g.,

```haskell
myEcho :: IO ()
-- reads a line from stdin and echos it back out to stdout
getLine >>= putStrLn
```

it’s not clear how, for example, putStrLn would actually be implemented. On a practical implementation level, the answer is that it can’t be: putStrLn is ultimately a piece of C code deep in the Haskell runtime. However, by looking at IO a bit differently we can imagine how the core concept behind IO could be implemented within pure Haskell code.

The key to understanding how to implement IO in pure code is to reimagine what a Haskell program looks like at the top level. To build a Haskell executable, you need a `main` function, e.g.,

```haskell
main :: IO ()
main = putStrLn "Hello World!"
```

Note that `main` has type `IO ()`. Following the same logic as our previous monads, we can read this as “a computation that returns () along with a sequence of interactions with the environment”. But this sounds quite familiar: a “a computation [...] along with sequence of interactions with the environment” is just an impure, sequential program! Depending on the underlying data structures we use, this program would be represented by a string containing the source code for a C program, or a Python AST, or in general a program in any impure language.

Thus, instead of viewing `main` as an impure function that executes when the binary starts, we can instead view `main` as a pure function that returns an impure program which is then executed by something else: for example we could represent this impure program as a C program, pass it to `gcc`, compile it, and run the
resulting executable to perform the actual interaction with the environment. For practical reasons this is not how the underlying implementation works, but it is essentially\textsuperscript{15} equivalent to how the implementation works. Anywhere we can view a Haskell program as being impure, we can also view that program as a pure program that generates an impure program. In other words, Haskell is less a general-purpose language and more a domain-specific language for generating impure programs!

do-notation

As we’ve seen from many of the examples above, while monadic code behaves like sequential code it often fails to look like it. When we’re writing monadic code, we frequently make use of the following two “tricks”.

First, instead of assigning to intermediate variables as is done in sequential languages, in Haskell we use beta-reduction to assign (frequently called “bind”) the results of monadic computations to readable names, e.g.,

\begin{verbatim}
bindingTrickExample :: MemoryUsageTracker ()
bindingTrickExample = malloc 12 >>= (\block1 \rightarrow free block1)
\end{verbatim}

instead of

\begin{verbatim}
{ block1 = malloc 12;
  free block1;
}
\end{verbatim}

Second, we return () and bind to _ in cases where we do not care about the computed value but still want to guarantee sequencing, e.g.,

\begin{verbatim}
discardTrickExample :: Allocation \rightarrow Int \rightarrow MemoryUsageTracker ()
discardTrickExample b1 size = free b1 >>= (\_ \rightarrow malloc size >>= (\_ \rightarrow return ()))
\end{verbatim}

instead of

\begin{verbatim}
{ free b1;
  malloc size;
  return None;
}
\end{verbatim}

With these two tricks in mind, we can mechanically translate our monadic code to the sequential code it emulates, e.g.,

\begin{verbatim}
exampleProgram :: MemoryUsageTracker ()
exampleProgram = malloc 12 >>= (\block1 \rightarrow
  malloc 15 >>= (\block2 \rightarrow
    malloc 8 >>= (\block3 \rightarrow
      free block1 >>= (\_ \rightarrow
        malloc 9 >>= (\block4 \rightarrow
          free block3))))))
\end{verbatim}

is equivalent to

\begin{verbatim}
def exampleProgram() {
  block1 = malloc 12;
  block2 = malloc 15;
  block3 = malloc 8;
  free block1;
  block4 = malloc 9;
  free block3;
}
\end{verbatim}

and similarly, for any sequential code, e.g.,

\textsuperscript{15}There are some exceptions, namely unsafePerformIO which actually does perform impure computation, but for the vast majority of programs this equivalence holds.
def exampleProgram2() {
    block1 = malloc 13;
    malloc 16;
    free block1;
    block2 = malloc 13;
    block3 = malloc 16;
    free block2;
    malloc 12;
}

we can translate it into the equivalent monadic code:

```
exampleProgram2 :: MemoryUsageTracker ()
exampleProgram2 = malloc 13 >>=
    (\block1 -> malloc 16 >>=
        (\_ -> free block1 >>=
            (\_ -> malloc 13 >>=
                (\block2 -> malloc 16 >>=
                    (\block3 -> free block2 >>=
                        (\_ -> malloc 12 >>=
                            (\_ -> return ())))))))
```

Since this translation process is entirely mechanical, it would be convenient if we could construct monadic values using a syntax that looks more like a sequential program, and then the compiler would automatically expand it out to the corresponding monadic code when it compiles our program. This is what do-notation provides: syntactic sugar for constructing monadic values. We can write

```
exampleProgram2WithDo :: MemoryUsageTracker ()
exampleProgram2WithDo = do
    block1 <- malloc 13
    free block1
    _ <- malloc 16
    block2 <- malloc 13
    block3 <- malloc 16
    free block2
    _ <- malloc 12
    return ()
```

and the Haskell compiler will internally translate this to

```
exampleProgram2Real :: MemoryUsageTracker ()
exampleProgram2Real = malloc 13 >>=
    (\block1 -> malloc 16 >>=
        (\_ -> free block1 >>=
            (\_ -> malloc 13 >>=
                (\block2 -> malloc 16 >>=
                    (\block3 -> free block2 >>=
                        (\_ -> malloc 12 >>=
                            (\_ -> return ())))))))
```

In do-notation, for the most part each line must either be an assignment where the left-hand side is a valid Haskell variable name (or pattern match) and the right hand side is value of type `Monad m => m a`, or a value of type `Monad m => m ()` (in which case we just implicitly throw away the resulting `()`). Examining the lines of `exampleProgram2WithDo`, we can see that these conditions hold:

```
:type (malloc 13)
(malloc 13) :: HeapLangInterpreter m b => m b

type (free undefined)
(free undefined) :: HeapLangInterpreter m b => m ()

type (return ())
(return ()) :: Monad m => m ()
```

However, we can also construct more complex-looking programs that also satisfy these rules, e.g.,
exampleProgramComplexDo :: MemoryUsageTracker ()
exampleProgramComplexDo = do block1 <- let increment x = x + 1
 mysize = increment 12
  free myblock
  return myblock
nextsize <- return 13
(block2, block3) <- (malloc nextsize >>= (
  b2 ->
  malloc 16 >>= (
    b3 ->
    return (b2, b3))))
let amIRun = malloc 12345
  free block2
let beforeIReturn = malloc 12
  beforeIReturn >>= (\_ -> Writer (0, ()))

I recommend going through exampleProgramComplexDo and seeing how each line satisfies the rules of do-notation. In fact, you should notice that exampleProgramComplexDo is equivalent to exampleProgram2Real and exampleProgram2WithDo.

Beyond Monads

While monads are arguably the best-known of Haskell’s typeclasses, there exist many other typeclasses, including other fundamental structures of computation, that are used throughout Haskell code. In this section we’ll focus on one set of typeclasses deriving from category theory (Functor, Applicative, and Monad), but there also exist typeclasses from abstract algebra (e.g., Semigroup, Monoid), typeclasses describing collections and iteration (e.g., Foldable, Traversable), as well as a number of category-theoretical constructs that we won’t cover here (e.g., Arrow, Comonad, Category, Divisible, and Predicate).

Functor

Functors distill the concept of a container holding zero or more elements. The Functor typeclass is defined as follows:

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

Intuitively, fmap is responsible for applying a function over the elements of the container. As such, we require it to follow the following laws:

1. fmap id ≡ id ("if we apply id to the elements of the container it should be no different than applying id to the container directly, as none of the elements should have been modified in either case")
2. fmap (g . h) ≡ (fmap g) . (fmap h) ("since our Functor should just act as a container of independent elements, it shouldn’t matter we apply functions over it in one pass, i.e., g . h, or in two passes, i.e., g and then h")

As expected, many common containers are instances of Functor:

map :: (a -> b) -> [a] -> [b]
map f (h:rest) = (f h):(map f rest)
map _ [] = []

instance Functor [] where
  fmap :: (a -> b) -> [a] -> [b]
  fmap = map

instance Functor (Fst t) where
  fmap :: (a -> b) -> Fst t a -> Fst t b
  -- unwrapped: "fmap :: (a -> b) -> (a, t) -> (b, t)

If you’re interested in exploring the wider world of Haskell typeclasses, check out the Typeclassopedia.
Typeclass Constraints

It turns out that not only is \texttt{Reader} a functor, but in fact every \texttt{Monad} is also a \texttt{Functor}: every \texttt{Monad} contains a “return value” which, using \texttt{bind}, we can extract and compute over—just like \texttt{fmap}. Note that while every \texttt{Monad} is a \texttt{Functor}, not all \texttt{Functors} are \texttt{Monads}—this should be unsurprising, as the capability to sequence both value computations and side effects in a \texttt{Monad} is intuitively much broader than the capabilities of a generic container. Because every \texttt{Monad} is a \texttt{Functor}, monadic code often interleaves \texttt{Functor} operations with monadic operations, e.g.,

\begin{verbatim}
  type Dollars = Float
  getPropertyValue :: Monad m => m Dollars
  getPropertyValue = return 100

  calculateTax :: Monad m => Dollars -> m Dollars
  calculateTax d = return (if d > 1000 then (d * 0.10) else (d * 0.05))

  taxEvasion :: (Functor m, Monad m) => m Dollars
  taxEvasion = (fmap \(\lambda\ value \rightarrow value * 0.8\) getPropertyValue) >>= calculateTax
\end{verbatim}

In these cases, requiring the programmer to specify that type \texttt{m} is both a \texttt{Functor} and a \texttt{Monad} is redundant, as if a type \texttt{m} is a valid \texttt{Monad} then mathematically it must also be a valid \texttt{Functor}. To allow the compiler to take advantage of this fact, similar to our \texttt{Monoid} constraint on \texttt{Writer} we can add additional type constraints when declaring typeclasses, e.g.,

\begin{verbatim}
  class Functor m => Monad m where
  -- ...
\end{verbatim}

With this constraint added the compiler will require that every \texttt{Monad} have a corresponding \texttt{Functor} instance and in return we will only need to add the type constraint \texttt{Monad m} instead of both \texttt{Monad m} and \texttt{Functor m} to any code using both monadic and functor operations, e.g.,

\begin{verbatim}
  taxEvasion :: Monad m => m Dollars
  taxEvasion = (fmap \(\lambda\ value \rightarrow value * 0.8\) getPropertyValue) >>= calculateTax
\end{verbatim}

Applicative

Historically \texttt{Functor} and \texttt{Monad} were the dominant typeclasses in \texttt{Haskell}, and the \texttt{Monad} typeclass was subject to the constraint \texttt{Functor m => Monad m}. However, a 2008 paper proposed a useful intermediate abstraction
called Applicative. The Applicative definition is stronger than Functor (i.e., every Applicative is a Functor but not every Functor is an Applicative) but weaker than Monad (i.e., every Monad is an Applicative but not every Applicative is a Monad). Conceptually, where Monad provides the ability to interleave both value computations and effects, Applicative allows us to sequence value computations and effects but not to interleave them.

The canonical example of this difference is ifM vs ifA:

\[
\text{ifM :: Monad m => m Bool \to m a \to m a \to m a} \\
\text{ifA :: Applicative m \to m Bool \to m a \to m a \to m a}
\]

When using the monadic interface, we can evaluate the condition (and its side effects), and if it returns True then we can run the then branch (and its side effects) while ignoring the else branch, and if it returns False then we can run the else branch (and its side effects) while ignoring the then branch:

\[
\text{ifM (Just True) (Just 5) Nothing} \\
\text{Just 5}
\]

In comparison, when using the applicative interface, we can sequence the value computations (i.e., if the condition returns True then return the value of the then branch otherwise return the value of the else branch) and the effects (i.e., first run the effects of the condition, then the then branch, then the false branch), but we can’t use the value returned by the condition to influence which side effects are run, e.g.,

\[
\text{ifA (Just True) (Just 5) Nothing} \\
\text{Nothing}
\]

Formally, the Applicative typeclass is defined as follows:

\[
\text{class Functor m \to Applicative m where} \\
\text{pure :: a \to m a} \\
\text{(<*>) :: m (a \to b) \to m a \to m b}
\]

along with the following laws

1. \text{pure id <*> v = v} \\
2. \text{pure f <*> pure x = pure (f x)} \\
3. \text{u <*> pure y = pure (\f \to f y) <*> u} \\
4. \text{u <*> (v <*> w) = pure (\_) <*> u <*> v <*> w} \\
5. \text{fmap g x = pure g <*> x}

Since every Monad is an Applicative, we’ll also want to add that as a constraint on Monad:

\[
\text{class Applicative m \to Monad m where} \\
\text{return :: a \to m a} \\
\text{return = pure} \\
\text{(>>=) :: m a \to (a \to m b) \to m b}
\]

We can see that while Applicative resembles Monad, there is a difference in the type signatures of <*> and >>=. This difference becomes clearer when we consider \text{flip (<*>)} and >>=:

\[
\text{flip :: (a \to b \to c) \to (b \to a \to c)} \\
\text{flip f = (\langle x y \to f y x \rangle)}
\]

\[
\text{:type (flip (<*>))} \\
\text{(flip (<*>)) :: Applicative m \to m a \to m (a \to b) \to m b}
\]

\[
\text{:type (>=)} \\
\text{(>=) :: Monad m \to m a \to (a \to m b) \to m b}
\]

In >>=, the value computation \(a\) can affect the produced side effects \(m\) through the type of the “step” function \(a \to m b\). However, with <*> the value computation is entirely contained within the side effects.
(m (a -> b)) and so the side effects (m) and the value computation (a, a -> b, b) are independent of each other.

Another way to understand the difference between Applicative and Monad is to examine a case of an Applicative which cannot have a corresponding Monad instance defined. Let’s consider the following applicative instance:

```haskell
data WriterWithNoValue s a = WriterWithNoValue s

instance Functor (WriterWithNoValue s) where
  fmap :: (a -> b) -> WriterWithNoValue s a -> WriterWithNoValue s b
  fmap _ (WriterWithNoValue x) = (WriterWithNoValue x)

instance Monoid s => Applicative (WriterWithNoValue s) where
  pure _ = WriterWithNoValue mempty
  (<*>) :: WriterWithNoValue s (a -> b) -> WriterWithNoValue s a -> WriterWithNoValue s b
  -- unwrapped: "(<*>) :: s -> s -> s"
  (<*>) (WriterWithNoValue fSideEffects) (WriterWithNoValue xSideEffects) =
    WriterWithNoValue (fSideEffects <> xSideEffects)
```

Here we have a modified version of the Writer monad we defined earlier, but here we’ve left out Writer’s value parameter. Since we’re able to define an Applicative instance for WriterWithNoValue, it must hold that the side effects of Applicative can be sequenced without any knowledge of the value computation, since for WriterWithNoValue there is no value computation! Trying to define a Monad instance, however, quickly runs into trouble:

```haskell
instance Monoid s => Monad (WriterWithNoValue s) where
  return _ = WriterWithNoValue mempty
  (>>=) :: WriterWithNoValue s a -> (a -> WriterWithNoValue s b) -> WriterWithNoValue s b
  -- unwrapped: "(>>=) :: s -> (a -> s) -> s"
  (>>=) (WriterWithNoValue lSideEffects) sf =
    let (WriterWithNoValue sfSideEffects) = sf _ -- we don’t have an a to pass to sf!
        in WriterWithNoValue (lSideEffects <> sfSideEffects)
```

It’s easy to see why: the unwrapped type of <*> is s -> s -> s, which is satisfied by mappend, but >>= has unwrapped type s -> (a -> s) -> s and since we cannot assume any properties of a we are unable to create an a to pass to the “step” function! Thus, we can clearly see how the effects of an Applicative are independent of the value computation, but for a Monad they are fundamentally linked.