1. Basics: 4/1/14

This course is taught by David Mazières and Bryan O’Sullivan, who together have done a lot of systems programming and research and Haskell. Meanwhile, the CA, David Terei, is a member of the Haskell language standards committee. . .

The goal of this class is to learn how to use Haskell to program systems with reduced upfront cost. Haskell is typically taught in the context of programming language research, but this course will adopt a systems perspective. It’s a surprisingly flexible and effective language, and since it was created by and for programming language resources, there are lots of interesting things to do, and it’s extremely flexible (if you want to try something new, even if it’s syntactical, this is as easy as using a library, unlike most programming languages).

The first week of this class will cover the basics of Haskell, though having some prior experience or a supplement (e.g. Real World Haskell or Learn You a Haskell) is helpful. After the basics, we will cover more advanced techniques. The class grade will be based on attendance and participation, with scribing one of the lectures, and also three warm-up programming assignments and a large final project and presentation, in groups of one to three people.

Now, let’s talk about Haskell.

In order to use Haskell, one will want to install the Haskell platform or cabal, along with the Haskell compiler, ghc.

The simplest program is

```haskell
main = putStrLn "Hello, world!"
```

Unsurprisingly, this is a “Hello, world!” program. One can compile it, e.g. `ghc -o hello hello.hs`, but also load it into an interpreter `ghci` (in this regard, Haskell is much like Lisp).

The first thing you’ve noticed is the equals sign, which makes a binding, e.g.

```haskell
x = 2  -- Two hyphens introduce a comment.
y = 3  -- Comments go until the end of a line.
main = let z = x + y  -- let introduces local bindings.
       in print z
```
This program will print 5.

Bound names can’t start with uppercase letters, and are separated by semicolons, which are usually automatically inserted as above. Functions can even be bound, e.g.:

```
add x y = x + y    -- defines function add
five = add 2 3    -- invokes function add
```

This is a good way to define functions.

Haskell is a pure functional language. Thus, unlike variables in imperative languages, Haskell bindings are immutable: within a given scope, each symbol can be bound only once. In other words, the following is an error:

```
x = 5
x = 6   -- Error, cannot re-bind x
```

Bindings are thus order-independent.

Another interesting fact is that bindings are lazy: definitions of symbols are only evaluated when they’re needed. For example:

```
safeDiv x y =
    let q = div x y  -- safe as q isn’t evaluated if y == 0
        in if y == 0 then 0 else q
main = print (safeDiv 1 0)    -- prints 0
```

Notice this is completely different from C-like languages!

Another interesting aspect of bindings, which goes hand-in-hand with order-independence, is that bindings are recursive, so each binding is in scope within its own definition.

```
x = 5
-- not used in main!
main = let x = x + 1
        in print x
        -- introduces a new x, defined in terms of itself
        -- loops forever, or stack overflows
```

In C, this would print 6, but here, \( x \) refers to itself! The runtime sometimes is able to detect the loop, however.

This means that writing things in Haskell requires thinking differently! For example, here’s a factorial program in C:

```
long factorial(int n) {
    long result = 1;
    while (n > 1)
        result *= n--;
    return result;
}
```

But in Haskell, one uses recursion.

```
factorial n = if n > 1 then n * factorial (n-1)
              else 1
```

However, the C function requires constant space, but the Haskell version requires \( n \) stack frames! But Haskell supports optimized tail recursion; if a function ends with a call to itself (i.e. is tail-recursive), then it can be optimized into a loop. However, the definition provided above isn’t tail-recursive.

Using an accumulator, the factorial function can be made tail-recursive.

```
factorial n = let loop acc n' = if n' > 1
                  then loop (acc * n') (n' - 1)
                   else acc
                 in loop 1 n
```

This uses an accumulator to keep track of the partial result. It’s a bit clunky, but can be tidied up with Haskell’s incredible concision. For example, one can use guards to shorten function declarations, e.g.

```
factorial n = let loop acc n' | n' > 1 = loop (acc * n') (n' - 1)
                 | otherwise = acc
                         in loop 1 n
```

The guards (pipe symbols) are evaluated from top to bottom; the first one that evaluates to \texttt{True} is followed. \texttt{otherwise} is defined to be \texttt{True}, but it makes the code easier to read. One might also introduce a \texttt{where} clause, which is like \texttt{let} but can support multiple guarded definitions, and is thus convenient for use around guards.
factorial n = loop 1 n
where loop acc n' | n' > 1 = loop (acc * n') (n' - 1)
    | otherwise = acc

You’ll notice that there will be plenty of inner functions, and their arguments are related to that of the outer functions. But it’s easy to confuse \( n \) and \( n' \), so the following code compiles and throws a runtime error!

factorial n = loop 1 n
where loop acc n' | n' > 1 = loop (acc * n) (n' - 1) -- bug, should be n'
    | otherwise = acc

One way to work around that is to use a naming convention in which the outermost variable has the longer name; then, bugs like this are caught at compile time, due to scope.

Haskell is strongly typed, so we have types such as \( \text{Bool} \), which is either \( \text{True} \) or \( \text{False} \); \( \text{Char} \), which is a Unicode code point; \( \text{Int} \), a fixed-size integer; \( \text{Integer} \), an arbitrary-sized integer; \( \text{Double} \), which is like a double in C; and also functions. A function from type \( a \) to type \( b \) is denoted \( a \rightarrow b \). We also have tuples: \((a_1, a_2, a_3)\), including the unit () (a zero tuple, kind of like void of C).

It’s good practice to write the function’s type on top of a function, e.g.

\[
\text{add} :: \text{Integer} \rightarrow (\text{Integer} \rightarrow \text{Integer})
\]
\[
\text{add} \ arg_1 \ arg_2 = \arg_1 + \arg_2
\]

Well, here’s something interesting. The arrow associates to the right, so these parentheses aren’t strictly necessary, but they make an important point: all functions accept only one argument, so the above function takes an integer and returns a function! For example, \( \text{add} \ 2 \ 3 \) is parsed as \((\text{add} \ 2) \ 3\), and \( \text{add} \ 2 \) is a function. Often, this behavior (called currying) is optimized out by the compiler, but can be useful. The compiler can infer types, and in the interpreter this can be queried by :\( \text{t} \).

\[
\text{*Main} \ > \ \text{:t add}
\]
\[
\text{add} :: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer}
\]

The user can also define data types, using the \textbf{data} keyword.

\[
\text{data PointT = PointC Double Double deriving Show}
\]

This declares the type \( \text{PointT} \) with a constructor \( \text{PointC} \) containing two \( \text{Double} \)s. The phrase \textit{deriving Show} means that it can be printed, which is useful in the interpreter. Types and constructors must start with capital names, but live in different namespaces, so they can be given the same name.

Types may have multiple constructors, and said constructors don’t actually need arguments (which makes them look sort of like \textit{enums} in C).

\[
\text{data Point = Cartesian Double Double}
| \text{Polar Double Double deriving Show}
\]

\[
\text{data Color = Red | Green | Blue | Violet deriving (Show, Eq, Enum)}
\]

Now, we can do things like \( \text{myPoint} = \text{Cartesian} \ 1.0 \ 1.0 \) and so on.

One extracts this data using \textbf{case} statements and guards, as in the following example:

\[
\text{getX, getMaxCoord :: Point -> Double}
\]
\[
\text{getX point} = \text{case point of}
\]
\[
\text{Point x y -> x} -- \text{if only the Point x y constructor is around}
\]
\[
\text{getMaxCoord (Point x y) | x > y = x}
| \text{otherwise} = y
\]

\[
\text{isRed :: Color -> Bool}
\]
\[
\text{isRed Red = True} -- \text{Only matches constructor Red}
\]
\[
\text{isRed c = False} -- \text{Lower-case \( c \) just a variable}
\]

The latter notion is called \textit{pattern matching}, which detects which constructor was used to create the object. For another example, consider the following:

\[
\text{whatKind :: Point -> String -- Cartesian or polar constructor as above}
\]
\[
\text{whatKind (Cartesian _ _)} = \"\text{Cartesian}\"
\]
\[
\text{whatKind (Polar _ _)} = \"\text{Polar}\"
\]
This underscore indicates that the value is unused, or something we don’t care about. The compiler can actually infer and optimize based on that. It’s bound, but never used, which is quite helpful, especially given that the compiler warns about unused variables.

Exercise 1.1. Given the following types for a rock-paper-scissors game:

```haskell
data Move = Rock | Paper | Scissors deriving (Eq, Read, Show, Enum, Bounded)
data Outcome = Lose | Tie | Win deriving (Show, Eq, Ord)
```

Define a function `outcome :: Move -> Move -> Outcome`. The first move should be your own, the second your opponent’s, and then the function should indicate whether one won, lost, or tied.

**Solution.**

```haskell
outcome :: Move -> Move -> Outcome
outcome Rock Scissors = Win
outcome Paper Rock = Win
outcome Scissors Paper = Win
outcome us them | us == them = Tie
                 | otherwise = Lose
```

There are plenty of other ways to do this.

Types, much like functions, can accept parameters, but type parameters start with lowercase letters. For example, within the standard Prelude:

```haskell
data Maybe a = Just a | Nothing
data Either a b = Left a | Right b
```

Maybe is used to indicate the presence of an item, or some sort of error, and `Either` can provide more useful error information, etc. In this case, the convention is for `Right` to indicate the normal value, and `Left` some sort of sinister error. The interpreter can reason about these types, too:

```
Prelude> :t Just True
Just True :: Maybe Bool
Prelude> :t Left True
Left True :: Either Bool b
```

Often, one uses the underscore pattern matching mentioned above with these parameterized types to pass exceptions along. For example,

```haskell
addMaybes mx my | Just x <- mx , Just y <- my = Just (x + y)
addMaybes _ _ = Nothing
```

Equivalently (and more simply),

```haskell
addMaybes (Just x) (Just y) = Just (x + y)
adMaybes _ _ = Nothing
```

Lists. Now, we have enough information to define lists, somewhat like the following.

```haskell
data List a = Cons a (List a) | Nil
oneTwoThree = (Cons 1 (Cons 2 (Cons 3 Nil))) :: List Integer
```

But since lists are so common, there are some shortcuts: instead of `List Integer`, one writes `[Integer]`, and the `Cons` function is written `:`, and is infix. The empty list is called `[]`, so instead we could have written `oneTwoThree = 1:2:3:`. Alternatively, there is nicer syntax:

```haskell
oneTwoThree’ = [1, 2, 3] -- comma-separated elements within brackets
oneTwoThree’’ = [1..3] -- define list by a range
```

Strings are just lists of characters.

Here are some useful list functions from the Prelude:
head :: [a] -> a -- first element of a list
head (x:_:xs) = x
head [] = error "head: empty list"

-- all but the first element
tail :: [a] -> [a]
tail (_:xs) = xs
tail [] = error "tail: empty list"

-- infix operator to concatenate lists
(ys ++ [] : xs) ++ ys = ys
(x ++ ys) = x ++ ys

-- This code is from language specification.
length :: [a] -> Int
length [] = 0
length (_:l) = 1 + length l

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

Note the function error :: String -> a, which reports assertion failures. filter is a higher-order function, i.e. one of its arguments is also a function. For example, one might define a function isOdd :: Integer -> Bool and then filter a list of Integers as filter isOdd listName.

In addition to deriving Show, one has deriving Read, allowing one to parse a value from a string at runtime. But parsing is tricky: there could be multiple possible values, or ambiguity. For example, suppose we have the following declaration:

```haskell
data Point = Point Double Double deriving (Show, Read)
```

Then, the following would happen in the interpreter.

```haskell
*Main> reads "invalid Point 1 2" :: [(Point, String)]
[]
*Main> reads "Point 1 2" :: [(Point, String)]
[(Point 1.0 2.0,"")]
*Main> reads "Point 1 2 and some extra stuff" :: [(Point, String)]
[(Point 1.0 2.0," and some extra stuff")]
*Main> reads "(Point 1 2)" :: [(Point, String)] -- note parens OK
[(Point 1.0 2.0,"")]
```

Notice that reads returns a list of possibilities, along with the rest of the string. This is asymmetrical from show.

This isn't a language property, but the best way to search for functions (and their source code!) is Hoogle, at [http://www.haskell.org/hoogle/](http://www.haskell.org/hoogle/) This is a good way to look up functions, their type signatures, and so on. Looking at the source is a great way to learn the good ways to do things in Haskell; in particular, notice just how short the functions are. Haskell has a steep learning curve, but it's pretty easy to understand what code is doing.

For another example of functional thinking, here's a function to count the number of lowercase letters in a string.

```haskell
import Data.Char
-- brings function isLower into scope

countLowerCase :: String -> Int
-- String is just [Char]
countLowerCase xs = length (filter isLower xs)
```

This looks absurd in C, but since Haskell is lazily evaluated, it doesn’t actually copy the string; in some sense, values and function pointers are the same under lazy evaluation. But, of course, this can be written more concisely:

```haskell
countLowerCase :: String -> Int
countLowerCase = length . filter isLower
```

Here, $f \ . \ g$ means $f \circ g$ mathematically: this is function composition: $(f \ . \ g) \ x = f \ (g \ x)$. But now, the argument isn’t necessary. This can be used like a concise, right-to-left analogue to Unix pipelines. This is known as point-free programming.

Conversely, if one wants the arguments but not the function, you can just use lambda expressions. The notion is $\backslash \ variables -> body$. For example:
countLowercaseAndDigits :: String -> Int
countLowercaseAndDigits =
    length . filter (\c -> isLower c || isDigit c)
This is useful for small functions that don’t need to be used in more than one place.
Another neat syntactic trick is that any function can be made prefix or infix. For functions that start with a letter, underscore, digit, or apostrophe, prefix is the default, but they can be made infix with backticks, e.g. 1 ‘add‘ 2. For anything else, infix is default, adding parentheses makes them prefix: (+) 1 2 and so on. Infix functions can also be partially applied:

stripPunctuation :: String -> String
stripPunctuation = filter ('notElem ' "!#$%&*+./<>?@\^\|\-\:")
-- Note above string the SECOND argument to notElem

2. Basics 2: 4/3/14
Fixity. Most operators are just library functions in Haskell; only a very few are reserved by the language syntax (specifically, ... ; :: ; + \ l , <- , -> , @ , ~ , and \ ). So there’s nothing to stop someone from defining their own operators. One can specify operators’ associativity and “fixity” with commands such as infixl, infix, infixr, and so on, which determine the associativity (and precedence is given by a number).

Here are some infixr 0 operators.

• ($$) :: (a -> b) -> a -> b
  f $ x = f x
This is just function application with the lowest precedence, so it’s useful for things like

putStrLn $ "the value of " ++ key ++ " is " ++ show value

• seq :: a -> b -> b returns the second argument, but forces the first to evaluate, which actually has to be built into the compiler.

• $! combines $ and seq. This is useful: in the recursive factorial programs implemented last lecture, the accumulator could contain $O(n)$ thunks, thanks to lazy evaluation. Using the above can help with that:

factorial n0 = loop 1 n0
where loop acc n | n > 1 = (loop $! acc * n) (n - 1)
                | otherwise = acc

factorial n0 = loop 1 n0
where loop acc n | n > 1 = acc "seq" loop (acc * n) (n - 1)
                | otherwise = acc

Cabal and Hackage. Hackage is a large collection of Haskell packages, which will probably be useful for the project later in this course. Cabal is a tool for browsing Hackage and installing packages; run cabal update to create a file $HOME/.cabal, and within the config file, we’re recommended to set documentation: True and library-profiling: True.

Modules and importing files. Haskell groups top-level bindings into modules. The default module name is Main, which contains the main function where programs start. All other modules M must reside in the file M.hs. At the top of a source file for module ModuleName, one can write module ModuleName where or module ModuleName (...) where, where the list of exported symbols is in the parentheses. Then, one can import ModuleName to access its functions, or use qualified or hiding to export only some symbols (or suppress some symbols) to prevent namespace conflicts.

With this, here’s a complete program:
Notice that the do notation is necessary because pure functions can’t print or read text from a terminal... but real programs find that somewhat useful. Then, this do block allows one to sequence I/O actions, such as

- pattern <- action binds pattern (a variable or constructor pattern) to the result of executing action.
- let pattern = value (no in necessary) binds pattern to value, which should be a pure value.
- Just calling an action executes it, and discards the result (or returns it, at the end of a block).

GHCI input is like a do block, in that one can use the above syntax.

Note that do, let, and case don’t parse after a prefix function unless one uses parentheses, e.g. func (do...). Thus, it’s convenient to use func $ do instead, to keep track of fewer parentheses.

What’s the type of an I/O action?

I/O is a parameterized type; IO a indicates an I/O action that produces a type a if executed. Unlike Maybe, we don’t actually see the constructor, and it’s just a little bit magical.

It seems reasonable in some languages to write something like main = putStrLn stdout (getLine stdin), because putStrLn expects a type String but gets type IO String. So how do we work around that? We can’t use case, because there’s no visible constructor.

Well, there’s another way to look at IO. hGetLine and hPutStrLn return actions that can change the world. In some sense, the world is passed along as an argument, and the result is a modified world, world'! Then, the do block builds a compound action from these other actions; when executed, it applies IO a actions to the world, and then extracts values of type a. In some sense, the whole program is like this, for the main function.

Short note on compiling: using ghc, one can create an executable like in C. But GHCI can read object code, so one can type ghci ./greet.hs, and it reads the object file. But then, it doesn’t have access to the internal symbols and functions, so the command prompt lacks an asterisk; use an asterisk in the :load *filename command to override this.

Now, what if we want I/O actions to return things? Every line in a do block must have type IO a, so we need a trivial IO action. This is the return :: a -> IO a function, which is badly named, because it’s a function, not a control-flow statement. It doesn’t terminate a do block, even though it is often used to return values from do blocks.

There’s also point-free I/O composition, using >>= (pronounced “bind”) rather than .. This composes right-to-left, and can be used to get rid of name in the above code:

Interestingly, do is just syntactic sugar for the bind operator; the above is processed as

The value is thrown away if it’s not used for anything, which is what the underscore means.

Here’s another example, which obtains a MDove type for playing rock, paper, scissors as in last lecture:

Here are some more polymorphic functions from the Prelude:
There's a function \( \text{print} \ a = \text{putStrLn} \ (\text{show} \ a) \), but we don't know how to express its type right now.

There are two kinds of polymorphism: the first, \textit{parametric polymorphism}, does the same thing for all types, e.g. \( \text{id} \). These work for every possible type. But there's also \textit{ad hoc polymorphism}, which does different things to different types. For example, \( 1 + 1 \) and \( 1.0 + 1.0 \) compute different things. \( \text{show} \) is another example of an ad hoc polymorphic function. These ad hoc polymorphic functions are called methods, and declared within classes.

```
class MyShow a where
  myShow :: a -> String
```

Then, for each type for which this is defined, one provides an instance declaration:

```
data Point = Point Double Double
instance MyShow Point where
  myShow (Point x y) = "(\text{\texttt{\text{show}}} \ x \ \text{\texttt{\text{show}}} \ y)"
```

All right, what's the type of \( \text{myShow} \)?

```
myPrint x = putStrLn $ myShow x
```

```
myPrint :: MyShow a => a -> IO ()
```

This is what is known as a context. This seems a bit laborious, and there's no way to write \( \text{deriving myShow} \) as with the standard typeclasses.

The context is in some sense hiding some implicit dictionary arguments, of how exactly to call the functions.

The Dreaded Monomorphism Restriction (DMR). Suppose \( \text{superExpensive} :: \text{Num} \ a \Rightarrow \text{Int} \rightarrow \ a \) is some expensive function whose result we want to cache. Then, we might call \( \text{cachedResult} = \text{superExpensive} \ 5 \), which will be a thunk that is executed once, and then done. Then, \( \text{cachedResult} \) has type \( \text{Integer} \), so it's a good thing that \( \text{cachedResult} \) doesn't have type \( \text{Num} \ a \Rightarrow a \); were this the case, it would require the context to be provided each time, and thus recompute everything. Note that the type of \( \text{0} \) is \( \text{Num} \ a \Rightarrow a \), because it can be any numeric type, so this is a good thing to know.

In this case, the compiler can't always infer everything: if something looks like a function, a statement will infer ad hoc and parametric types, but if it looks like anything else, there will only be parametric inference unless it's explicitly declared. To look like a function, something should bind to a single symbol rather than a patter (e.g. \( \text{f} \ x = \ldots \)). Moreover, it should have an explicit argument, i.e. \( \text{f} \ x = \ldots \). Types are first inferred from use elsewhere (such as type signatures), then some educated guessing. But if that doesn't work, compilation may fail.

In practice, this boils down to the following:
But all of these will work if you declare explicit type signatures for these functions! There’s a moral here.

Notice that this turns a runtime error (which would be really slow for the caching example) into a compiler error.

**Superclasses and Contexts.** One class can require classes to be members of another. For example, Ord instances don’t make sense without an Eq instance, so Ord declares Eq as a superclass, using a context.

```haskell
class Eq a => Ord a where
  (<), (>=), (>), (<=) :: a -> a -> Bool
  a <= b = a == b || a < b -- default methods can use superclasses
  -- and so on
```

One might also want instances to have contexts:

```haskell
instance (MyShow a) => MyShow [a] where
  myShow [] = "["]
  myShow (x:xs) = myShow x ++ ":" ++ myShow xs
```

Superclasses and contexts are used to define classes of parameterized types. For example, Functor is a class of parameterized types on which you can map functions (e.g. lists, where the function is mapped onto every element; Maybe, where it’s mapped onto the Just).

Here’s the declaration:

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
instance Functor Maybe where
  fmap _ Nothing = Nothing
  fmap f (Just a) = Just (f a)
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
instance Functor [] where
  fmap = map
instance Functor IO where
  fmap f io = io >>= return . f
  -- equivalent to: do val <- io; return (f val)
```

Now, one can use `fmap` in any of these contexts, e.g.

```haskell
greet h = do
  hPutStrLn h "What is your name?"
  fmap ("Hi, " ++ (hGetLine h)) >>= hPutStrLn h
```

So what happens if one tries to make Int into a functor? Then, `fmap :: (a -> b) -> Int a -> Int b`, which produces a compilation error: Int isn’t a parameterized type. Thus, we can classify types into those that require zero parameters (e.g. Int, Double, ()), those that require one parameter (Maybe, IO, []), those that require two constructors (Either, ()), and so on. Parameterized types (i.e. at least one parameter) are called type constructors.

One can reason about these with some new notation. * is a kind of type constructor with no parameters. * -> * is the kind of type with one parameter, and * -> * -> * is a type constructor with two arguments of kind *. Thus, Functor must accept kind * -> *.

**Monads.** The moment you’ve all been waiting for... the entire first two lectures have been working towards this point. `return` and `>>=` are actually methods of a class called Monad.

```haskell
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
  fail :: String -> m a -- called when pattern binding fails
  fail s = error s -- default is to throw exception
  (>>) :: m a -> m b -> m b
```

9
This has pretty far-reaching consequences; for example, one could use do blocks for non-I/O purposes, and a lot of library code works with the general Monad class, rather than just I/O.

Well, why would you want to do that? It turns out monads appear in a bunch of places. For example, Maybe is a monad:

```haskell
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing >>= _ = Nothing
  return = Just
  fail _ = Nothing
```

This is pretty useful, because multiple Maybe statements can be combined with do notation, as in the following example, where one processes a form.

```haskell
extractA :: String -> Maybe Int
extractB :: String -> Maybe String
parseForm :: String -> Maybe Form
parseForm raw = do
  a <- extractA raw
  b <- extractB raw
  return (Form a b ...)
```

This is a good way to do exception handling! Moreover, because Haskell is lazy, expensive computations after an error will be skipped: computation stops at the first Nothing.

Another example of monads in the real world is a security library which implemented a restricted I/O monad. Then, the entire standard library works well with this.

**Algebraic Data Types.** Some data types have a lot of fields:

```haskell
-- Argument to createProcess function
data CreateProcess = CreateProcess CmdSpec (Maybe FilePath)
  (Maybe [(String, String)]) StdStream StdStream StdStream Bool
```

This is pretty tangled; instead, one can use an algebraic data type, that allows one to label fields, akin to a struct in C.

```haskell
data CreateProcess = CreateProcess {
  cmdspec :: CmdSpec,
  cwd :: Maybe FilePath,
  env :: Maybe [(String, String)],
  std_in :: StdStream,
  std_out :: StdStream,
  std_err :: StdStream,
  close_fds :: Bool
}
```

### 3. Testing and Quickcheck: 4/8/14

Today’s lecture is by Bryan O’Sullivan, who has been involved in Haskell for twenty years, and has written one of the major standard references, *Real World Haskell*. There are several “states of the art” for testing software:

- An Excel spreadsheet full of different tests to be done by hand. Really.
- Unit testing, where

```java
public class TestAdder {
  public void testSum() {
    Adder adder = new AdderImpl();
    assert(adder.add(2, 2) == 4)
    assert(adder.add(5, 6) == 11)
    // and so on
  }
}
```

- Integration testing checks that various components of a program work together.
Fuzz testing feeds random input to a program. This might have been used to discover the recent major security bug in OpenSSL.

Unit tests are only useful up to a point: people get bored quickly, and also tend to miss important corner cases. That's why people often turn to computers, which are more patient than people. For example, UTF-8, locked up in a castle such that nobody has to think about it. It's a variable-length encoding and relatively obscure, so there have been several security exploits based on incorrect assumptions people make about it. Basically, it's real-world, obscure, yet important, so good unit testing is particularly crucial.

In Haskell, Char represents a Unicode point, so encodeChar should have type Char -> [Word16] (after importing Data.Char) (or Char -> Either (Word16, (Word16, Word16)), if you want to be fancier). Then, we can build up some unit tests.

```haskell
import Data.Char (ord) -- it's good practice to explicitly specify what you import
import Data.Bits ((.&.), shiftR) -- low-level bit-hacking functions

ord :: Char -> Int
fromIntegral :: (Integral a, Num b) => a -> b
encodeChar :: Char -> [Word16]
encodeChar x
  | w < 0x10000 = [fromIntegral w]
  | otherwise = [fromIntegral a, fromIntegral b]
    where w = ord x
    -- low-level bit operations go here
```

Now, we can do unit testing with Test.HUnit. How do you pronounce HUnit? Bryan O'Sullivan doesn’t know. This is basically a port of JUnit, the granddaddy of all automatic unit testing frameworks.

Here's an example of a bad test.

```haskell
import Test.HUnit
badTest = do
  assertEqual "sestertius encode as one code unit" 1 (length (encodeChar \x10198'))
```

This, of course, fails if you run it, as it should; this symbol has two words.

But now we can parameterize and generalize the test:

```haskell
testOne char = do
  assertEqual "ASCII values have one code point" 1 (length (encodeChar char))
goodTest = do mapM_ ['\x0'..'\x7f']
```

But explicitly parameterizing these takes a while, and is somewhat inefficient. Thus, Test.QuickCheck exists. The idea is that, especially for combinatorial things for which it would be very hard and time-consuming to explicitly test everything, it can automate randomized testing. For most types, QuickCheck starts with small test cases (which is nice, so that bugs found are more likely to be minimal), and then builds up to bigger ones. But only 100 tests isn't really enough to find a character in the 2-word range, so QuickCheck isn't telling the whole story yet.

QuickCheck generates random values using a typeclass called Arbitrary. This is a multipurpose abstraction useful in many ways in Haskell.

```haskell
-- Generator type
data Gen a

-- Used to generate values, abstracted away from where we can see ot
class Arbitrary a where
  arbitrary :: Gen a

-- Generate a random value within a range
-- Random is a typeclass, which means it's meaningful to choose a random value.
choose :: Random a => (a, a) -> Gen a
  -- For example, choosing a random Bool
instance Arbitrary Bool where
  arbitrary = choose (False, True)
instance Arbitrary Char
  -- all of the chars go here...
```
Now, we can denote a class `Testable` if propositions about it can be tested:

```haskell
-- Protection for a Gen
data Property = MkProperty (Gen a)

-- The set of types that can be tested
class Testable prop

instance Testable Bool -- the body code isn't interesting magical. What is:
instance (Arbitrary a, Show a, Testable prop) => Testable (a -> prop)
```

In other words, any function that can accept an `Arbitrary` function and return a proposition is really testable! Thus, something is testable if it's a terminal return type (a `Bool`) or something that returns a `Testable`. Thus, we can create functions of arbitrarily many arguments `Testable`. This is pretty deep magic.

Now, we can say more about our testing code:

```haskell
prop_encodeOne :: Char -> Bool
prop_encodeOne c = length (encodeChar c) == 1
```

Now we can just call `quickCheck prop_encodeOne`, which runs 100 (or more or less, if you specify) random tests. There's also `verboseCheck`, which provides more information about the passed tests.

The test is broken, even if one increases the number of tests. Let's look at the source.

```haskell
module Test.QuickCheck.Arbitrary where

instance Arbitrary Char where
  arbitrary = chr 'fmap' oneof (choose (0,127), choose (0,255))

Well, it's no longer 1985, so we want to have more diverse Chars around. We want a type that is almost exactly like Char, except with a different `Arbitrary` instance. This is what the `newtype` keyword does.

```haskell
newtype BigChar = Big Char deriving (Eq, Show)
```

Now, we want to write a new `Arbitrary` instance which chooses values between 0 and `0x10ffff`. So we want to use `choose`, which means making BigChar an instance of `Random`:

```haskell
import Control.Arrow (first) -- a function that maps a function to the first element of a tuple
import System.Random

instance Random BigChar where
  random = first Big 'fmap' random
  -- and so on. A little bit of yak shaving, so to speak. Somewhat boilerplate
```

But there's a GHC language extension that works around this! GHC usually only can derive standard typeclasses from `newtype`, but this allows it to infer things about non-standard typeclasses. It almost always works.

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
newtype BigChar = Big Char deriving (Eq, Show, Random) -- just works. Splendid!
```

Now we can write more comprehensive tests:

```haskell
instance Arbitrary BigChar where
  arbitrary = chose (Big '0', Big '\x10FFFF')

prop_encodeOne3 (Big c) = length (encodeChar c) == 1
```

Now, this fails almost instantly, and tells us what counterexample it failed on.

This changes the unit test policy of hand-coding variations on a theme to the QuickCheck way of expecting a property to universally hold true and randomly checking inputs. This is almost entirely focused on random values, but there are other libraries (e.g. SmallCheck) that can be used to find bugs that appear with extremely low probability.

One problem is that random values tend to be large; are there smaller inputs that could cause the bug to happen again? This makes the bug easier to understand; finding the smallest input is known as shrinking.

Writing a function called `shrink` takes an `Arbitrary` value and generates a list of smaller `Arbitrary` values. For example:
import System.Random
import Test.QuickCheck
import Data.Char

newtype BigChar = Big Char
  deriving (Eq, Show, Random)

instance Arbitrary BigChar where
  arbitrary = choose (Big '0', Big '\x10FFFF')
  shrink (Big c) = map Big (shrinkChar c)
  shrinkChar c = map chr $ shrink $ ord c

prop_encodeOne2 = do
  c <- choose ('\0', '\xFFFF')
  return $ length (encodeChar c) == 1

Now that this setup all exists, we can look at ways of choosing test values. One could generate them directly:


At this point, we've seen the `Functor` class, with `fmap` used to lift a pure function into a functor. In general, the idea of “lifting” is to take a concept and transform it to work in a different, more general, setting. For instance, one can lift with monads:

```
liftM :: (Monad M) -> (a -> b) -> m a -> m b
liftM f action = do
  b <- action
  return (f b)
```

This `liftM` is a cousin of `fmap`: they do very similar things. Basically, `liftM` executes some action, and applies `f` to it, and `fmap` does something that is effectively the same. Notice their type signatures:

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

It turns out that lifting pure functions into monads is extremely common, so `Control.Monad` provides lots of combinator functions `liftM2` to `liftM5` which lift functions of 2 through 5 arguments. This allows us to tighten `Arbitrary` instances from

```
data Point a = Point a a -- as in last week's lectures

instance (Arbitrary a) => Arbitrary (Point a) where
  arbitrary = do
    x <- arbitrary
    y <- arbitrary
    return (Point x y)
```

to

```
import Control.Monad (liftM2)
instance (Arbitrary a) => Arbitrary (Point a) where
  arbitrary = liftM2 Point arbitrary arbitrary
```

It just works.

QuickCheck also allows one to shrink tuples, using a predefined function that allows one to shrink pairs of data points. More interestingly, one could generate arbitrary recursive data structures:

```
data Tree a = Node (Tree a) (Tree a)
  | leaf a
deriving (Show)

instance (Arbitrary a) => Arbitrary (Tree a) where
  arbitrary = choose (liftM Leaf arbitrary, liftM2 Node arbitrary arbitrary)
```
4. Exception Handling and Concurrency: 4/10/14

In Haskell, concurrency and exceptions are related, because of Haskell’s concept of asynchronous exceptions. These are a lot prettier than the corresponding concepts in, say, C++.

We have so far seen a few functions that return any type, e.g. `undefined :: a` (used to build code up and test individual functions, suppressing type errors from other code) and `error :: String -> a`. These are language-level constructions; to use exceptions directly, call `import Control.Exception`. (In older versions of ghc, one need to hide an earlier `catch` function, so `import Prelude hiding (catch); this is probably not necessary anymore).

Thus, we have an `Exception` typeclass, and the following functions:

```haskell
class (Typeable e, Show e) => Exception e where ...
  throw :: Exception e => e -> a
  throwIO :: Exception e => e -> IO a
  catch :: Exception e => IO a -> (e -> IO a) -> IO a
```

The `Typeable` typeclass requires the `DerivingDataTypable` language extension, as in the following code; this will be discussed more in depth in a future lecture. The following code creates an error catching function, which invokes a handler when an error is received by `catch`.

```haskell
{-# LANGUAGE DeriveDataTypeable #-}
import Prelude hiding (catch) -- if necessary, but still good practice.
import Control.Exception
import Data.Typeable
data MyError = MyError String deriving (Show, Typeable)
instance Exception MyError

catcher :: IO a -> IO (Maybe a)
catcher action = fmap Just action 'catch' handler
  where handler (MyError msg) = do putStrLn msg; return Nothing
```

Note that the type of `handler` can’t be inferred, so it will be necessary to create a constructor or use a type signature. The constructor pattern `e@(SomeException _)` matches all exceptions, rather than just `MyError`.

The `Exception` typeclass has two methods, `toException :: e -> SomeException` and `fromException :: SomeException -> e` (which determines whether it’s the correct exception we are looking for).

Haskell’s pure functionality makes this more interesting: exceptions can be thrown in pure code, but must be caught within `IO`. This is because the world state may change in exception handling, and also because there’s ambiguity in the order of evaluation, e.g. in `(error "one") + (error "two")`. Nondeterminism is allowed, but within the `IO` monad. Then, `throw` produces a thunk that throws an exception, but is lazily evaluated; `throwIO` is strict, so in the following code, only the first exception is reached.

```haskell
do x <- throwIO (MyError "one")  -- this exception thrown
   y <- throwIO (MyError "two")  -- this code not reached
return (x + y)
```

In general, `throwIO` is better, but there are some exceptions:

```haskell
pureCatcher :: a -> IO (Maybe a)
pureCatcher a = (a 'seq' return (Just a))
  'catch' \(SomeException _) -> return Nothing

pureCatcher $ 1 + 1
Just 2
*Main> pureCatcher $ 1 'div' 0
Nothing
*Main> pureCatcher (undefined :: String)
Nothing
*Main> pureCatcher (undefined,undefined :: String)
Just "*** Exception: Prelude.undefined"
```

1 No pun intended.
This is because evaluating a constructor of a list doesn’t evaluate the head or tail, so \( \text{seq}(\text{undefined}:\text{undefined})() \) just returns () again, since it just evaluates the constructor. To force evaluation of every element when constructing, use \( \text{deepseq} \) in the library of the same name (for many common data types), or match on the constructor ([] or \( \langle x:x:s \rangle \)) to actually look at everything.

```haskell
seqList :: [a] -> b -> b
seqList [] b = b
seqList (a:as) b = seq a $ seqList as b
```

Some more useful functions: try returns Right \( a \) if it works normally, and Left \( e \) if an exception occurred. finally and onException run clean-up actions within IO; finally in all cases, and onException only when there’s an exception.

```haskell
try :: Exception e => IO a -> IO (Either e a)
finally :: IO a -> IO b -> IO a
      -- cleanup always
onException :: IO a -> IO b -> IO a
      -- after exception
      -- In both cases, result b of cleanup action is discarded.
```

Though these are all nice, language-level exceptions are cumbersome outside of the IO monad; in other monads, it might be better to use the fail method that is part of the Monad typeclass. For example, in Maybe, \( \text{fail} \_ = \text{Nothing} \), while in other monads, this might throw an exception.

**Threads.** Haskell’s threads are lightweight and user-level, and actually are more useful than threads in other languages! They live in Control.Concurrent. A new thread is created with \( \text{forkIO} :: \text{IO} () \rightarrow \text{IO} \text{ThreadId} \), which creates a new thread and returns its ID. Here are some other useful functions:

```haskell
throwTo :: Exception e => ThreadId -> e -> IO ()
killThread :: ThreadId -> IO ()
      -- = flip throwTo ThreadKilled
threadDelay :: Int -> IO ()
      -- sleeps for # of sec
myThreadId :: IO ThreadId
```

As an example of some code, here’s a timeout function, which evaluates an action or aborts after a certain number of microseconds. There’s a builtin in the library System.Timeout.

```haskell
data TimedOut = TimedOut UTCTime deriving (Eq, Show, Typeable)
instance Exception TimedOut

timeout :: Int -> IO a -> IO (Maybe a)
timeout usec action = do
  -- Create unique exception val (for nested timeouts)
  -- the unique value is just the current time
  expired <- fmap TimedOut getCurrentTime
  ptid <- myThreadId
  -- Idea: child thread tracks the time, and kills the thread
  -- if time runs out.
  let child = do threadDelay usec
    throwTo ptid expired
    parent = do ctid <- forkIO child
      result <- action
      killThread ctid
      return $ Just result
      -- uses catchJust in case the parent has to deal with other exceptions
    catchJust (\e -> if e == expired then Just e else Nothing)
      parent
      (\_ -> return Nothing)

An MVar is a mutable variable that is either full or empty (something in it, or not), and allows threads to communicate via shared variables.

```haskell
newEmptyMVar :: IO (MVar a)
newMVar :: a -> IO (MVar a)
      -- create empty MVar
      -- create full MVar given val

-- These block a thread until the MVar becomes available.
takeMVar :: MVar a -> IO a
putMVar :: MVar a -> a -> IO ()
      -- non-blocking versions
tryTakeMVar :: MVar a -> IO (Maybe a)
      -- Nothing if empty
```
If an `MVar` is empty, `putMVar` fills it with a value, and if it's full, `takeMVar` empties it. If several try to access these, they're blocked to accessing them one at a time. These are akin to channels in Go, and may be less intuitive than mutexes and condition variables, but are a lot less error-prone.

In use, here's a benchmarking program.

```haskell
import Control.Concurrent
import Control.Exception
import Control.Monad

pingpong :: Bool -> Int -> IO ()
pingpong v n = do
  mvc <- newEmptyMVar  -- MVar read by child
  mvp <- newEmptyMVar  -- MVar read by parent
  let parent k = do
      when v $ putStr $ " " ++ show k
      putMVar mvc k
      takeMVar mvp >>= parent
    | otherwise = return ()
  child = do
    k <- takeMVar mvc
    putMVar mvp (k - 1)
    child
  tid <- forkIO child
  -- no matter what happens, clean up the child thread
  parent n 'finally' killThread tid
  when v $ putStrLn ""

-- *Main*> pingpong True 10
-- 10 9 8 7 6 5 4 3 2 1
```

A library called `criterion` for Haskell, written by Bryan O'Sullivan, is a great way to do benchmarking; it's a really good library. It's used as follows:

```haskell
import Criterion.Main

main :: IO ()
main = defaultMain [bench "thread switch test" mybench]
where mybench = pingpong False 10000

-- -O for optimization
$ ghc -O pingpong.hs
[1 of 1] Compiling Main             ( pingpong.hs, pingpong.o )
Linking pingpong ... 
$ ./pingpong
benchmarking thread switch test
mean: 3.774590 ms, lb 3.739223 ms, ub 3.808865 ms, ci 0.950
```

It turns out that there are two versions of the Haskell runtime: by calling `ghc -threaded`, one can take advantage of multiprocessing. Specifically, this allows one to create OS threads (as with `pthreads`). Then, the initial thread is an OS thread, and with `forkOS`, a new OS thread is created, akin to `forkIO`.

So you'd think adding `-threaded` makes things faster, but this means some threads are unbound, and therefore migrate between heavier-weight OS threads. The initial thread was bound, so this was basically a benchmark of Linux threads! Thus, if one can keep communicating threads in the same OS thread, context switches will be more like function calls. There's a library call `runInUnboundThread` for this, which is implemented somewhat like this:

```haskell
wrap :: IO a -> IO a
wrap action = do
  mv <- newEmptyMVar
  -- Exception transferred to parent from child, by laziness
  _ <- forkIO $ (action >>= putMVar mv) "catch"
    \e@(SomeException _) -> putMVar mv (throw e)
  takeMVar mv
```
But bound threads are useful: if an unbound thread blocks, it could block the whole program, so it’s nice to keep that within its own thread. This is also relevant to the Foreign Function Interface for Haskell calls into C code: with -threaded, safe C code calls are given their own OS thread. Similarly, there are C libraries where the code expects to be called from the same thread. Finally, one might want to run on a specific CPU (e.g. forkOn).

Another useful collection of functions allow one to modify an MVar without forgetting to release it:

```hs
modifyMVar :: MVar a -> (a -> IO (a, b)) -> IO b
modifyMVar_ :: MVar a -> (a -> IO a) -> IO ()
```

For example, to increment an MVar Int, one could write

```
modifyMVar x (
  n -> return (n+1, n))
```

The underscored version always returns unit; the other one returns the modified value.

Here’s what an initial implementation might look like:

```hs
modifyMVar :: MVar a -> (a -> IO (a,b)) -> IO b
modifyMVar m action = do
  v0 <- takeMVar m
  (v, r) <- action v0 'onException' putMVar m v0
  putMVar m v
  return r
```

But what if another thread kills this thread using killThread in the middle? Then, the MVar is stuck. This is where we might want support for masking exceptions:

```hs
mask :: ((forall a. IO a -> IO a) -> IO b) -> IO b
```

(The forall comes from a language extension called RankNTypes, and can be ignored for now.) Then, mask $ \f \to b$ runs action b with asynchronous exceptions ignored, and the function f allows exceptions to be unmasked again. Exceptions are also unmasked while a thread sleeps (e.g. takeMVar).

Here’s a masked version of the above code:

```hs
modifyMVar :: MVar a -> (a -> IO (a,b)) -> IO b
modifyMVar m action = mask $ \ unmask -> do
  v0 <- takeMVar m
  -- automatically unmasked while waiting
  -- here, we do want to allow exceptions, but not before or after
  (v, r) <- unmask (action v0) 'onException' putMVar m v0
  putMVar m v
  return r
```

The wrap function given above is also not good, and can be fixed as follows:

```hs
wrap :: IO a -> IO a
wrap action = do
  mv <- newEmptyMVar
  -- first we disable interrupts
  mask $ \ unmask -> do
    -- but if we get an exception here, we stick it into the MVar
    tid <- forkIO $ (unmask action >>= putMVar mv) 'catch' \e@(SomeException _) -> putMVar mv (throw e)
    -- here, takeMVar re-enables exceptions, and passes them off to the child thread
    -- eventually, takeMVar succeeds, and we return its value.
    let loop = takeMVar mv 'catch' \e@(SomeException _) ->
      throwTo tid e >> loop
    loop
```

mask is powerful but tricky, so let’s introduce some higher-level constructs. bracket accepts an opening action, a cleanup action, and then a main action:

```hs
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
-- Init Cleanup Main
```

Now, we can fix the parent function from the time-out example:

```hs
parent = bracket (forkIO child) killThread $ \_ -> fmap Just action
```
MVars are fairly versatile: if you need or want a mutex, an MVar will do fine.

```haskell
type Mutex = MVar ThreadId

mutex_create :: IO Mutex
mutex_create = newEmptyMVar

mutex_lock, mutex_unlock :: Mutex -> IO ()
mutex_lock mv = myThreadId >>= putMVar mv
mutex_unlock mv = do mytid <- myThreadId
  lockTid <- tryTakeMVar mv
  unless (lockTid == Just mytid) $
    error "mutex_unlock"

As for condition variables, they tend not to work well with asynchronous exceptions, but one could make a condition variable an MVar which is a list of MVar units, each of which belongs to a thread that’s waiting. Here’s the full implementation:

```haskell
data Cond = Cond ( MVar [ MVar () ])

cond_create :: IO Cond
cond_create = liftM Cond $ newMVar []
  -- liftM is fmap for Monads (i.e., no required Functor instance):
  -- liftM f m1 = do x <- m1; return (f m1)

cond_wait :: Mutex -> Cond -> IO ()
cond_wait m ( Cond waiters ) = do
  me <- newEmptyMVar
  modifyMVar_ waiters $ \others -> return $ others ++ [me]
  mutex_unlock m
  -- note we don’t care if preempted after this
  takeMVar me 'finally' mutex_lock m
  -- wake up, and mutex is unlocked
  {- In general, it’s bad to not atomically release the mutex and then go
  to sleep, but here, this “inside” MVar is local to a single thread -}

cond_signal, cond_broadcast :: Cond -> IO ()
cond_signal ( Cond waiters ) = modifyMVar_ waiters wakeone
  where wakeone [] = return []
       wakeone (w:ws) = putMVar w () >> return ws

cond_broadcast ( Cond waiters ) = modifyMVar_ waiters wakeall
  where wakeall ws = do mapM_ ( flip putMVar () ) ws
                      return []

The point is that putting MVars inside each other can be very powerful.

Control.Concurrent.Chan provides unbounded channels, also like Go.

```haskell
data Item a = Item a ( Stream a )

```

This can actually be implemented, in a simplified way, with MVars.

```haskell
data Item a = Item a ( Stream a )

```

```haskell
data Chan a = Chan ( MVar ( Stream a ) ) ( MVar ( Stream a ) )

```

```haskell
newChan :: IO ( Chan a )
newChan = do
  empty <- newEmptyMVar
  liftM2 Chan ( newMVar empty ) ( newMVar empty )
  -- liftM2 is like liftM for functions of two arguments:
  -- liftM2 f m1 m2 = do x1 <- m1; x2 <- m2; return (f x1 x2)

writeChan :: Chan a -> a -> IO ()
writeChan ( Chan _ w ) a = do
  empty <- newEmptyMVar
  modifyMVar_ w $ \oldEmpty -> do
```
putMVar oldEmpty (Item a empty)
return empty

readChan :: Chan a -> IO a
readChan (Chan r _) =
modifyMVar r $ \
full -> do
(Item a newFull) <- takeMVar full
return (newFull, a)

This threading also allows one to do networking! In the Network package, there's support for high-level networking, specifically TCP and Unix-domain sockets.

connectTo :: HostName -> PortID -> IO Handle
listenOn :: PortID -> IO Socket
accept :: Socket -> IO (Handle, HostName, PortNumber)
sClose :: Socket -> IO ()
hClose :: Handle -> IO ()

For example, one could take last week’s rock-paper-scissors program and make it work over a network:

import Network
-- accepts a single connection, plays a game, and then exits.
withClient :: PortID -> (Handle -> IO a) -> IO a
withClient listenPort fn =
bracket (listenOn listenPort) sClose $ \s -> do
bracket (accept s) ($\(h, _, _) -> hClose h) $
\(h, host, port) -> do
putStrLn $ "Connection from host " ++ host
++ " port " ++ show port
fn h

Use of bracket prevents leaking of file descriptors.

5. API Design: 4/15/14

“Dwarf Fortress... why was I reading about Dwarf Fortress?”

Today we’ll talk about API design and phantoms, a nontraditional apprach. Patterns are something we’ve seen before., albeit implicitly. For example, addition: we have a 0 such that 0 + n = n + 0 = n, and addition associates. Similarly, we have multiplication: there’s a 1 that is the identity, and multiplication associates. Similarly, lists and concatenation have a similar structure, with the empty list functioning as the identity, and concatenation ++ associating. The same is true with booleans and &&, with True the identity and associativity.

This is something called a monoid, which comes to us via abstract algebra.
Definition. A monoid is a set $M$ with an associative function $\oplus : M \times M \to M$ and an identity element $0 \in M$ such that $0 \oplus m = m \oplus 0 = m$ for all $m \in M$.

In Haskell, these appear in the Data.Monoid class: the identity is called `mempty` and the associative operator is called `mappend` (which is bad naming, but such is life). Then, they are required to satisfy these axioms delineated above. But how does Haskell enforce these axioms? It doesn’t. Haskell is a programming language, not a theorem prover (e.g. Agda or Coq), so be careful.

Here’s an example:

```haskell
instance Monoid [a] where
  mempty = []
  xs `mappend` ys = xs ++ ys
```

There are several other definitions which are type-correct, syntactically correct, and obey the monoid laws. For example, one could do

```haskell
instance Monoid [a] where
  mempty = []
  mappend _ _ = []
```

This one is useless, but the former one is the one that is actually useful, and we’ll use it. Numbers are interesting because they are monoids in two “good” ways, addition and multiplication. There’s a lesson here: one should use a typeclass where there’s exactly one canonical behavior one would expect for a given type; there’s one canonical choice for lists, so this can be done. For numbers, we can hide the machinery:

```haskell
newtype Product a = Product { getProduct :: a }
deriving (Eq, Ord, Read, Show, Bounded)
instance Num a => Monoid (Product a) where
  mempty = 1
  Product x `mappend` Product y = Product (x * y)

newtype Sum a = Sum { getSum :: a }
deriving (Eq, Ord, Read, Show, Bounded)
instance Num a => Monoid (Sum a) where
  mempty = 0
  Sum x `mappend` Sum y = Sum (x + y)
```

Now we take advantage of the Either type, which is usually used for error handling. For example, one might define type `Result a = Error String a`. One might use it to construct a Monoid instance that gives the first success from a chain of Either statements.

```haskell
instance Monoid Either String a) where
  mempty = Left "fnord"
  xs `mappend` ys
    | xs == Right b = b
    | otherwise = ys
```

The issue is, there doesn’t seem to be a good canonical value for `mempty`. In Haskell, type variables are quantified, i.e. the live in some domain. If there’s no typeclass mentioned, a type variable is implicitly universally quantified (which can be explicitly specified with a `forall` construction in a language extension), so, for example, `length` must accept a list of any class. But there’s no `mempty` for a general type, so this doesn’t work! So maybe we should restrict this to `Either String a` (so the left side is always strings). Thus, the following typechecks with the FlexibleInstances language extension. (Not all language extensions are safe or a good idea, but this one is pretty OK. One bad one is UndecidableInstances, which means the compiler doesn’t have to terminate.)

```haskell
{- this is a normal comment -}
{-# this is a special comment, a pragma for the compiler #-}
instance Monoid Either String a) where
  mempty = Left "fnord"
```

This is useful for two reasons: you might be a researcher, for whom infinite loops lead to papers, or because one might know a program is type-safe but the type-checker can’t verify it.

2This is useful for two reasons: you might be a researcher, for whom infinite loops lead to papers, or because one might know a program is type-safe but the type-checker can’t verify it.
So this is a valid monoid. But is it canonical? Not really. This is important because any time one declares an instance of any typeclass, it is automatically available to any module that imports the code, which pollutes your library users’ code. The way to get around this is to use a newtype to hide the special instance.

```haskell
{-# LANGUAGE FlexibleInstances #-}
import Data.Monoid
newtype FirstRight a b = FirstRight {
    getFirstRight :: Either a b
}
instance Monoid (FirstRight String a) where
    mempty = FirstRight (Left "suxx0rz")
    a@(FirstRight (Right )) ‘mappend‘ _ = a
    _ ‘mappend‘ b = b
```

Because monoids have this really predictable behavior, it’s now extremely obvious how one ought to use it, which is useful for organizing types and data. Basically, there’s a way to glue two of them together and obtain another one.

Let’s talk about HTTP POST requests. An ancient form protocol involves posting multipart data, some of which is mandatory, and some of which is optional.

```hs
data Part = Part {
    name :: String -- mandatory
    pathname :: Maybe FilePath -- optional
    -- et cetera
}
```

We can make an algebraic data type to represent a POST request body. Here are some types.

```hs
type Param = (String, String)
type ContenType = String
data Payload = NoPayload
    | Raw ContenType String
    | Params [Param]
    | FormData [Part]
    deriving Show
```

Can we make a nice Monoid out of them? NoPayload is the clear choice for mempty, but then after some cases it’s not clear how to proceed.

```hs
instance Monoid (Maybe Payload) where
    mempty = Nothing
    mappend Nothing b = b
    mappend a Nothing = a
    mappend (Just (Params a)) (Just (Params b)) = Just (Params (a ++ b))
    mappend _ _ = Nothing
```

This is technically a monoid, but it’s not a very satisfying one. There are plenty of APIs like this in the real world, but you can do better than a bunch of cases and pattern-matching.

But wait! If you feed this to GHCi, there’s an eight-line error message! This is scary; eight lines is far too long[3]. The issue here is: FlexibleInstances allows both instances of Maybe Payload and Maybe a (the latter in Data.Monoid) as Monoids. The language extension means there’s a compile-time error, though it only is discovered at a call to mappend, which is kind of annoying.

The solution is... another language extension. This is called OverlappingInstances, which dictates that conflicts such as this resolve to the most specific type. This is very convenient! And very dangerous. Here’s what can happen:

- Overlapping instances are big doors for semantic meaning of a program to escape entirely.
- One can get very confusing error messages.

A new instance can be defined invisibly somewhere else, which can completely change the actions of the program. On the other hand, this lead to interesting research.

One solution is something called a phantom type:

```haskell
data Payload a = NoPayload
  | Raw ContentType String
  | Params [Param]
  | FormData [Part]
  deriving Show
```

The type `a` appears exactly once, to prevent this overlapping instance conflict. Now, we can write an API.

```haskell
param :: String -> String -> Payload Param
param name value = Params [(name, value)]

filePart :: String -> FilePath -> IO (Payload [Part])
filePart name path = do
  body <- readFile name
  return (FormData [Part name (Just path) Nothing [body]])
```

These have the same runtime representation, but have different types, so the compiler can’t mix them. Now, we can write the following.

```haskell
addParams (Params as) (Params bs) = Params (as ++ bs)

instance Monoid (Payload [Param]) where
  mempty = NoPayload
  mappend = addParams
```

Now, we have the functions `Param`, `filePart`, and `fileString`, but how does one enforce the fact that this is it? This can be done by exporting the name of the type `Part`, but not its constructors (though instances, as mentioned above, always get exported). For example, the following `(..)` notation exports a type and all of its constructors:

```haskell
module PayloadPhantom
  (
    Part (..)
    -- and so on
  ) where
```

But instead, we can export the `Payload` type, without all of the constructors:

```haskell
module PayloadPhantom
  ( Part (..)
    Payload, -- meaning no constructors!
    param,
    filePart,
    fileString
  ) where
```

Another cool piece of syntax is that `<>` is shorthand for `mappend`. This is used all over the place.

So, why monoids? They tend to work well for APIs, because they force one to address important design questions early on. There’s also the semigroups package on Hackage, which gives monoids without an identity (just an associative operation); these aren’t related to monoids in the language, but ought to be.

There’s another interesting ata type for concurrent operations called `Data.IORef`, which is non-blocking. This lives in `Data.IORef`, and allows anyone who has access to it to read, write, or modify the contents. But what if one wanted specific permissions? Some people’s access should be read-only, and others should have read and write access. Here phantom types come to the rescue again!

```haskell
import Data.IORef

newtype Ref t a = Ref (IORef a)
```

Notice that `Ref` only exists and compile time, and the runtime performance is the same.
-- anybody can read a Ref
readRef :: Ref t a -> IO a
readRef (Ref ref) = readIORef ref

-- All of the checking is done at compile time!
-- No cost upon runtime
writeRef :: Ref ReadWrite a -> a -> IO ()
writeRef (Ref ref) v = writeIORef ref v

readOnly :: Ref t a -> Ref ReadOnly a
readOnly (Ref ref) = Ref ref

The type system allows for some expressive and surprising applications. For good reads on monoids, check out "Data Analysis with Monoids" and R. Lämmel's paper "Google's MapReduce Programming Model — Revisited" which flavors MapReduce with all sorts of interesting algebra.

6. STM: 4/17/14

How might one write a program to make a transaction? This entails sending money from Alice to Bob, but in a way that is thread-safe and checks if Alice has enough money to send.

import Control.Concurrent
import Control.Monad

type Account = MVar Double

printMV :: (Show a) => MVar a -> IO ()
printMV mv = withMVar mv print

{-# This is a bad solution: what if you get an asynchronous exception? Needs to do some exception handling. -}

It can also deadlock, e.g. transacting Alice -> Bob and Bob -> Alice
transfer account from to =
    modifyMVar_ from $ \bf -> do
        when (bf < amount) $ fail "not enough money"
        modifyMVar_ to $ \bt -> return $! bt + amount
        return $! bf - amount

transfer :: Double -> Account -> Account -> IO ()
transfer amount from to =
    let tryTransfer = modifyMVar_ From $ \bf -> do
        when (bf < amount) $ fail "not enough money"
        mbt <- tryTakeMVar to
        case mbt of
            Just bt -> do putMVar to $! bt + amount
            Nothing -> return (bf, False)
        ok <- tryTransfer
        unless ok $ safetransfer (- amount) to from
        -- Still needs to be fixed against asynchronous exceptions

main :: IO ()
main = do
    ac1 <- newMVar 10
    ac2 <- newMVar 0
    transfer 1 ac1 ac2
    printMV ac1
    printMV ac2
    return ()

This is pretty gross... instead, a model called software transactional memory uses database-like transactions to do things like this (e.g. writing to memory). This involves writing to a long, and then committing everything in the end. Notice that this would be difficult in procedural languages, such as C or Java, as something could write to the file system while this is going on. However, here, the IO type can control side effects.
In Control.Concurrent.TVar, there's a type called TVar which makes this work. Here are some methods.

```haskell
newTVarIO :: a -> IO (TVar a)  
readTVarIO :: TVar a -> IO a

readTVar :: TVar a -> STM a
writeTVar :: TVar a -> a -> STM ()
modifyTVar :: TVar a -> (a -> a) -> STM () -- lazy
modifyTVar' :: TVar a -> (a -> a) -> STM () -- strict

atomically :: STM a -> IO a
```

The last function is very interesting: the STM monad gives one global lock — but it acts otherwise like a more fine-grained lock. This comes at the price of lacking external IO actions. The transaction example from above now looks like this:

```haskell
type Account = TVar Double

transfer :: Double -> Account -> Account -> STM ()
transfer amount from to = do
  -- Quirk of Haskell: a - b can't be prefixed, since this conflicts with unary negation
  -- so subtract a b = b - a
  modifyTVar' from (subtract amount)
  modifyTVar' to (+ amount)

main :: IO ()
main = do
  ac1 <- newTVarIO 10
  ac2 <- newTVarIO 0
  atomically $ transfer 1 ac1 ac2
```

Two more useful functions are `retry :: STM a` and `orElse :: STM a -> STM a -> STM a`.

`retry` aborts a transaction, and then retries. But it’s "smart" in that it knows what TVar's have been read, and sleeps until at least one of them changes.

```haskell
orElse takes two transactions, and executes the first; then, if this fails, then it executes the second (if both fail, it sleeps). This is useful for nesting transactions.
```

```haskell
transfer2 :: Double -> Account -> Account -> Account -> STM ()
transfer2 amount from1 from2 to =
  atomically $ transferSTM amount from1 to
  'orElse' transferSTM amount from2 to
```

Another useful function is `alwaysSucceeds :: STM a -> STM ()`. This adds an invariant to check after every transaction, which either throws the exception, or is ignored. Here's an example in use:

```haskell
newAccount :: Double -> STM Account
newAccount balance = do
  tv <- newTVar balance
  alwaysSucceeds $ do balance <- readTVar tv
                      when (balance < 0) $ fail "negative balance"
  return tv
```

One can absolutely do this with weirder constraints. It's a really nice concurrency abstraction; there's no such thing as idiot-proof concurrent programming, but this makes many things a lot clearer.

Let's switch gears somewhat. How does the compiler represent data in memory? A value has a constructor and arguments, so it seems reasonable that we have a value's constructor at runtime (but not the type, because this is just type-checked at compile time).

```c
struct Val {
  unsigned long constrno; /* constructor # */
  struct Val *args[]; /* flexible array */
```
There may be more than one constructor, e.g. for list types, which is what constrno is useful for (it’s different for each constructor). However, this approach has neither room for exceptions nor thunks, which are very important. The garbage collector also can’t really follow this, and having lots of pointers for simple types like Int makes the whole program less efficient.

Here’s another idea; add a level of indirection to describe values.

```c
typedef struct Val {
    const struct ValInfo *info;
    struct Val *args[];
} Val;

/* Statically allocated at compile time. Only one per
 * constructor (or closure-creating expression, etc.) */
struct ValInfo {
    struct GCInfo gcInfo; /* for garbage collector */
    enum { CONSTRNO, FUNC, THUNK, IND } tag;
    union {
        unsigned int constrno;
        Val *(*func) (const Val *closure, const Val *arg);
        Exception *(*thunk) (Val *closure);
    };
};
```

Here, this struct can contain either a constructor (the tag is CONSTRNO, so the code from the previous block can still be used), a function, an exception, or a thunk. Note that the type is not present at runtime, as the program has already been checked at compile-time. Then, gcInfo indicates how many Val*s are in the args array, and where they are, so the garbage collector can do its thing. Notice that if tag == IND, then args[0] is an indirect pointer to another Val. This is useful in case args needs to grow in size.

Looking at functions, we can assume that all functions take only one argument (thanks to currying; this is somewhat untrue due to compiler optimization, but is reasonable enough for a first look). To apply f to argument a, one calls f->info->func (f, a); the closure argument is useful so that the function can be reused; it holds f itself. The closure isn’t always necessary (e.g. top-level bindings), but consider the following:

```c
add :: Int -> (Int -> Int)
add n = \m -> addn m
    where addn m = n + m
```

It seems that there are n different addn structures, even though they’ll have the same ValInfo value. Thus, the closure passed in is the same, and args[0] holds the value of n.

What about thunks? Then, there’s another function pointer, evaluated as v->info->thunk (v); Then, the thunk is updated in-place in memory (and isn’t a thunk anymore); v is passed in as a closure again, but this time, unlike functions, thunks update themselves, and they can also throw exceptions (functions can too, but then they can just be taken to return a thunk that throws an exception).

Turning a thunk into a non-thunk is known as “forcing.” One issue is that if the thunk’s return value doesn’t fit in its arguments, one has to allocate a new Val. This is why IND is a possible tag: the former thunk should be made with the IND tag, with a forwarding pointer to the new Val. Here’s what it might look like.

```c
Exception *force (Val **vp)
{
    for (;;) {
        if (((*vp)->info)->tag == IND)
            *vp = ((*vp)->arg)[0];
        else if (((*vp)->info)->tag == THUNK) {
            Exception *e = ((*vp)->info)->thunk (*vp);
            if (e)
                return e;
        }
        else // no exception
            return NULL;
    }
}
```

As an example of what this does, here’s how it handles currying. Start with the following function.
Then, the curried function `const3 a` returns a function `const3_1(const3, a)`, which is a function pointer to `const3_2`, and so on.

```
Val *const3_1 (Val *ignored, Val *a)
{
    v = (Val *) gc_malloc (offsetof (Val , args[1]));
    v->info = &const3_2_info; /* func = const3_2 */
    v->args[0] = a;
    return v;
}

Val *const3_2 (Val *closure, Val *b)
{
    v = (Val *) gc_malloc (offsetof (Val , args[2]));
    v->info = &const3_3_info; /* func = const3_3 */
    v->args[0] = b;
    v->args[1] = closure;
    return v;
}

Val *const3_3 (Val *closure, Val *v)
{
    return v->args[1]->args[0];
}
```

This solves some problems elegantly, but introduces lots of overhead, e.g. each Int needs a distinct ValInfo structure, and we don’t know which ones the program will need ahead of time. Thus, there are unboxed types that don’t need a Val struct to be used. Here’s a way of doing it.

```
union Arg {
    struct Val * boxed;  /* most values are boxed */
    unsigned long unboxed; /* "primitive" values */
};

typedef struct Val {
    const struct ValInfo * info;
    union Arg args[];  /* args can be boxed or unboxed */
} Val;
```

Unboxed types have no constructors, and can’t be thunks. But this requires updating the garbage collector slightly.

A GHC extension, via the {-XMagicHash} option, allows one to get unboxed types with the sharp character #; for example, we have the type Int#, and the operation +# (which is a single machine instruction). See also: GHC.Prim or :browse GHC.Prim in the interpreter. This looks like a module, but has an awful lot of magic compiler builtins in it. There are also unboxed constants, e.g. 2#, 2## (unsigned), ‘a’#, and 2.0##.

Now we know what Int actually is. It’s a single-constructor data type with exactly one unboxed argument. This allows it to contain thunks, but avoids all of the pointer traversing (e.g. unboxed Int#s can be stored in registers). Thus, while calculating 1 + 1 = 2 might be as complicated as case 1 of I# u -> I# (u +# 2#), it’s extremely useful in inner loops.

However, there are some restrictions on these unboxed types. They’re different kinds of types: an Int is of type *, and we also had * -> *, etc.), but unboxed types are of kind #, which is different. This is done because they aren’t polymorphic in the same way as boxed types. But applying the constructor allows one to box the type and define new datatypes.

```
{-# LANGUAGE MagicHash #-}
import GHC.Prim

-- These unboxed types are a challenge to syntax highlighters
data FastPoint = FastPoint Double# Double# -- ok
fp = FastPoint 2.0## 2.0## -- ok

-- Error: can’t pass unboxed type to polymorphic function
fp’ = FastPoint 2.0## (id 2.0##)
```
The next thing to look at is `seq`, which forces its first argument, and then forces the second argument and returns it. This ends up being tricky:

```haskell
infiniteLoop = infiniteLoop :: Char -- loops forever
seqTest1 = infiniteLoop 'seq' "Hello" -- loops forever
seqTest2 = str 'seq' length str -- returns 6
where str = infiniteLoop "Hello"
```

`seq` forces a `Val`, but not its arguments, so it forces a string's constructor, but not its head or tail. This is a concept known as weak-head normal form (WHNF). Here's an example implementation.

```haskell
const struct ValInfo seq_info = {
    some_gcinfo, THUNK, .thunk = &seq_thunk
};

Val * seq_2 (Val * closure, Val *b)
{
    /* assume seq_1 put first arg of (seq a b) in closure */
    c = (Val *) gc_malloc (offsetof (Val , args[2]));
    c->info = &seq_info;
    c->args[0] = closure->args[0];
    c->args[1] = b;
    return c;
}

Exception * seq_thunk (Void *c)
{
    Exception *e = force (&c->args[0]);
    if (!e)
    {
        c->info = &ind_info; /* ValInfo with tag = IND */
        c->args[0] = c->args[1]; /* forward to b */
    }
    return e;
}

With this behind-the-scenes look at Haskell, it's possible to understand some more of the higher-level magic. For example, one can declare strict wrapper types, e.g.

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

This means that the `Val`'s struct can't have the thunk or indirect types, so accessing a strict `Int` requires only one cache line.

Strictness is primarily used for optimization, in order to avoid building up long chains of thunks, or save overhead on checking whether a thunk has evaluated. But this comes at a loss of semantic data: a non-strict `Int` isn't just a number. There's also the value `_⊥`, corresponding to looping forever or throwing an exception. Types including `_⊥` are called lifted. Thus, statements such as

```haskell
Maybe !Int
```

incur an error.

The `case` statement can also force thunks; with the exception of irrefutable patterns (those that always evaluate, e.g. `_` or just a single variable), any non-irrefutable pattern forces evaluation of its argument, in top-to-bottom, then left-to-right order. Even function pattern matching is desugared into `case` statements, e.g.

```haskell
f ('a':'b':rest) = rest
f _ = "ok"
```

```haskell
test1 = f (undefined:[]) -- error
test2 = f ('a':undefined) -- error
test3 = f ('x':undefined) -- "ok" (didn't force tail)
```

A minor, but useful syntactic trick is that `_` makes a statement irrefutable. For example, `three = (\ h:t -> 3) undefined` evaluates to 3. This is useful for weird fixed-point constructions defined in terms of themselves. Finally, `newtype`, unlike `data`, creates an alias for a type (so no runtime code is different). Thus, a `case` statement on a `newtype` compiles to nothing.

```haskell
newtype NTInt = NTInt Int deriving (Show)
uNTInt = NTInt undefined
-- fine, because _ is irrefutable
testNT = case uNTInt of NTInt _ -> True -- returns True
```

However, forcing a value with its constructor leads to something different.
Thus, newtype is almost always better, when it applies. A pragma called UNPACK allows one to unbox all strict fields, via the compiler flag -funbox-strict-fields.

7. LANGUAGE EXTENSIONS: 4/22/14

GHC implements many language extensions to Haskell, enabled by placing {-# LANGUAGE extensionName #-} at the top of a file, or compiling with -XextensionName (or :set with the same flag in the interpreter). Some of these are very safe, integrated in Haskell’s or GHC’s libraries, or are easy to desugar, and others make type-checking nondeterministic or other less safe consequences.

Monad transformers are type constructors that build monads parameterized by other monads.

```haskell
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }

instance (Monad m) => Monad (StateT s m) where
    return a = StateT $ \s -> return (a, s)
    m >>= k = StateT $ \s0 -> do
        -- in monad m
        ~(a, s1) <- runStateT m s0
        runStateT (k a) s1

instance MonadTrans (StateT s) where
    lift ma = StateT $ \s -> do
        -- in monad m
        a <- ma
        return (a, s)
```

Then, there are helper functions get and put:

```haskell
get :: (Monad m) => StateT s m s
put :: (Monad m) => s -> StateT s m ()
```

For example, one can call the equivalent of x++ using state computations by getting x and putting back an incremented variable.

The `MonadIO` class is useful if one wants to execute an I/O action regardless of the current monad (however many lifts are necessary to get to the I/O monad).

```haskell
class (Monad m) => MonadIO m where
    liftIO :: IO a -> m a

instance MonadIO IO where
    liftIO . id

instance (MonadIO m) => MonadIO (StateT s m) where
    liftIO = lift . liftIO
```

Now, we can write functions that perform I/O, but work in many monads.

```haskell
myprint :: (Show a, MonadIO m) -> a -> m ()
myprint a = liftIO (print . show) a
```

Another background example is that bindings in Haskell are allowed to be recursive, e.g.

```haskell
oneTwo :: (Int, Int)
oneTwo = (fst y, snd x)
    where x = (1, snd y) -- mutual recursion
        y = (fst x, 2)

-- more reasonable example: memoizes the Fibonacci sequence
nthFib :: Int -> Integer
nthFib n = fibList !! n
```
This recursion can be implemented using something called a fixed-point combinator (which is a function that is a solution to a certain fixed-point equation). Instead, here we’ll use recursive bindings to implement a fixed-point combinator.

What this means is that if one takes a function that isn’t recursive, \(\text{fix}\) simulates recursion by computing a fixed point. However, monadic bindings are not recursive: the thing you’re binding isn’t in scope in the binding.

However, monads in the \texttt{MonadFix} class have a fixed-point combinator \(\text{mfix}\). Thus:

This is useful for simulating circuits with monads, which is actually pretty elegant, but requires recursion whenever a circuit has a loop.

Hence, the \texttt{RecursiveDo} language extension, which allows the new \texttt{rec} keyword to be desugared into \(\text{mfix}\). Thus, it’s useful for structuring one’s thinking, even if it’s easy to fix later. For example, it’s easy to create recursive data structures.

It can also be used to call non-strict methods of classes. So, how would one actually implement \(\text{mfix}\)? For a warm-up, here’s the \texttt{Identity} monad, which doesn’t actually do much.

Then, \(\text{mfix}\) is essentially \(\text{fix}\).

For a less trivial example, lazy I/O is accomplished using a magic \texttt{unsafeInterleaveIO} :: IO \(a \rightarrow IO \ A\), which looks like an \(IO\) identity function, but defers the \(I/O\) until the output is forced. But don’t use this at home, as it’s not Haskell-like or even functional.

For \(IO\), \(\text{mfix}\) is just \(\text{fixIO}\):
This is surprisingly similar to how the compiler implements the ordinary fix. In particular, by the time the thunk is forced, it contains the correct value.

One might want to generalize mfix to all monads, using something like mfix’ f = fix (>>= f) (turns it into a pure value), or mfix’ f = mfix’ f >>= f. But this doesn’t work, because >>= is strict in the first argument for many monads, so it recurses infinitely and causes a stack overflow. Thus, mfix needs to take the fixed point over a value, not a monadic action, which ends up being monad-specific and doesn’t work for all of them (e.g. not ContT or ListT).

One way to make this more concrete is to look at the mfix syntax for StateT.

```haskell
instance MonadFix m => MonadFix (StateT s m) where
  mfix f = StateT $ \s0 -> do
    ~(a, s1) <- runStateT (f a) s0
    return (a, s1)
```

Another way to think of this is fix f is a fixed-point of f, i.e. fix f = f (fix f), but it’s better to think of it as let x = f x in x. This latter definition is preferred because it won’t cause a stack overflow. This is because a let statement corresponds to allocating a thunk, but the first definition doesn’t allocate a thunk, so it keeps calling fix recursively, allocating memory for each iteration. This is a good rule to remember.

For a quick review of type classes, a Haskell 2010 type class declaration is of the form

```haskell
class ClassName var where
  methodName :: Typ {- and so on -}
```

Here, var need not have kind *. Then, one may have superclasses, etc, but the type of each method must mention var, and the ClassName var is implicitly carried around.

Right now, each typeclass can take only a single parameter, but a language extension works around this.

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
class Convert a b where convert :: a -> b
instance Convert Int Bool where convert = (== 0)
instance Convert Int Integer where convert = toInteger
instance (Convert a b) => Convert [a] [b] where convert = map convert
```

This extension is relatively safe in itself, but is a gateway drug to all sorts of depravity. It encourages other extensions, because each method’s type must mention every parameter: we can’t just write myDefault :: a. Also, all types must be fully determined, so we can’t just call convert 0 :: Bool, since 0 has type (Num a) => a.

Okay, so let’s enable flexible instances.

```haskell
{-# LANGUAGE FlexibleInstances #-}
instance Convert Int Char where convert = show
instance Convert a a where convert a = a
```

This works sometimes, and then throws an overlapping instances error for converting a list of Ints to itself. So we need another language extension! Clearly, OverlappingInstances is used, but widely frowned upon; it chooses the more specific instance, but doesn’t consider the context. Import this where the overlapping extensions are defined, not where those instances are used. An instance h1 is more specific than h2 when an instance for h1 can be created by substuting the values for h2, but not vice versa.

```haskell
-- requires OverlappingInstances, but works fine.
class MyShow a where MyShow :: a -> String
instance MyShow Char where myShow = show
instance MyShow Int where myShow = show
instance MyShow [Char] where myShow = id
instance (MyShow a) => MyShow a where myShow a = "[" ++ (join (myShow a)) ++ "]"
```

But this doesn’t solve our problem with Convert! For Convert [Int] [Int], one now has to add a third instance Convert [a] [a] to break the tie, as it’s more specific than both.

```haskell
module Help where
  class MyShow a where
    myShow :: a -> String
  instance MyShow a where
    myShow a = MyShow [a] where
```
myShow xs = concatMap myShow xs

showHelp :: MyShow a => [a] -> String
showHelp xs = myShow xs  -- doesn't see overlapping instance

module Main where
import Help

data T = MkT
instance MyShow T where
    myShow x = "Used generic instance"
instance MyShow [T] where
    myShow xs = "Used more specific instance"

main = do
    myShow [MkT]  -- generic
    showHelp [MkT]  -- specific...? Oops.

One can add an extra helper method showList to handle this tie, but it's still a good argument against the OverlappingInstances extension. But FlexibleContexts is a relatively safe: MultiParamTypeClasses leads to imexpressible types, with some otherwise illegal contexts. Basically, we can define functions, but not write down their types without the extension.

The upshot is that we can make StateT work from lots of methods.

{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleContexts #-}

class (Monad m) => MonadState s m where
    get :: m s
    -- these have the same implementations as before.
    put :: s -> m ()

But this causes problems to the following code.

xplusplus = do n <- get; put (n+1); return n

We can't infer that get has the type we need, which means it has to be explicitly specified (which is ugly, both in print and from a programming perspective) — what if there's some similar function out there for StateT with a Double?

Hence the FunctionalDependencies extension, which is widely used, but somewhat frowned upon. This lets a class define some parameters as functions of others, e.g.:

class (Monad m) => MonadState s m | m -> s
    -- and then get and put, and so on

Each of these steps seemed pretty reasonable, but can create instances where the type-checker can do arbitrary computation (!), and in particular might not terminate. In order to avoid this, there are two conditions. Write an instance as instance [context =>] head [where body]. Here, the context is zero or more comma-separated assertions (e.g. it's numeric).

There are two conditions, called the Paterson and Coverage conditions, imposed by the compiler, which reduce things to decidability.

crash = f5 ()
    where f0 x = (x,x)
    f1 x = f0 (f0 x)
    f2 x = f1 (f1 x)
    f3 x = f3 (f3 x)
    f4 x = f4 (f4 x)
    f5 x = f4 (f4 x)

This is perfectly legal and decidable, but the type size of $f_n$ grows exponentially, and causes the compiler to crash!

So now (gasp) we have the UndecidableInstances extension, which lifts the conditions and instead just imposes a maximum stack size on ghc (which can be increased or decreased with pragmas and/or compiler flags). This allows one instance to cover all monadic transformers! Somehow, this is less frowned upon than OverlappingInstances.

This allows one to do some pretty gnarly stuff.

data HFalse = HFalse deriving Show
data HTrue = HTrue deriving Show

-- then, functions such as hNot and hAnd hOr, by casework.
Can we compute when two types are equal?

```haskell
class TypeEq a b c | a b -> c where typeEq :: a -> b -> c
instance TypeEq a a HTrue where typeEq _ _ = HTrue
instance TypeEq a b HFalse where typeEq _ _ = HFalse
```

But this is defeated by the overlapping instances: neither is more specific than the other. With a little bit of trickery, this can be elided... and used to program recursively at the type level by distinguishing base and recursive cases. This relies fundamentally on OverlappingInstances, though. For one example, this can be used to make MonadState to recursively lift (base case, if this type is equal to that type, do the base case; otherwise, recurse). This can save you a lot of typing when designing general things.

8. **Generic Programming:** 4/24/14

“*The bottom is a singular value that inhabits every type. When evaluated the semantics of Haskell longer yield a meaningful value. It's usually written as the symbol ⊥ (i.e. the compiler flipping you off).”*  
– Stephen Diehl

The origin of this is that typing can be annoying in some contexts. For example, it’s easy to convert pairs (tuples) to lists of Strings, but it’s not possible to generalize, because show is a method (a collection of functions), rather than a function itself. One solution is to use language extensions to allow methods to be passed as arguments to functions.

```haskell
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleInstances #-}

class Function f a b | f a -> b where
  funcall :: f -> a -> b
instance Function (a -> b) a b where
  funcall = id

pairToList :: (Function f a c, Function f b c) => f -> (a, b) -> [c]
pairToList f (a, b) = [ funcall f a, funcall f b]
```

Now, we need placeholder singleton types to represent methods.

```haskell
data ShowF = ShowF  -- some unit type
instance (Show a) => Function ShowF a a [Char] where
  funcall _ = show

data FromEnumF = FromEnumF  
instance (Enum a) => Function FromEnumF a Int where
  funcall _ = fromEnum
```

Now we can implement pairToList as intended to convert pairs of tuples to lists of tuples. But can it be generalized to arbitrary n-tuples? There’s a way to auto-expand these definitions, e.g.

```haskell
class TupleFoldr f z t r | f z t -> r where
  tupleFoldr :: f -> z -> t -> r
```

This causes the compiler to run out of memory at about $n = 10$, and this is sort of it for compile-time tricks. More insights will come later.

**DeriveDataTypeable.** Haskell allows six classes to be automatically derived: Show, Read, Eq, Ord, Bounded, and Enum. But the DeriveDataTypeable adds two more, Typeable and Data.

Typeable, which lives in Data.Typeable, allows one to compare types for equality.

```haskell
class Typeable a where
  typeOf :: a -> TypeRep -- never evaluates argument

data TypeRep -- not exposed to user, but instance of Eq, Ord, Show, and Typeable
```

Now, we can do things like checking the type at runtime.

```haskell
rtTypeEq :: (Typeable a, Typeable b) => a -> b -> Bool
rtTypeEq a b = typeOf a == typeOf b
```

4Pun intended?
Last time, we did this at compile time, but this required additional language extensions. And this is generally different anyways.

This also has an interesting application: GHC has a function called `unsafeCoerce`, which can coerce any type into any other type... and can do lots of other scary things, such as segfault (e.g. converting `()` into a string). But using `Typeable`, one can make a safe case function.

```haskell
cast :: (Typeable a, Typeable b) => a -> Maybe b
cast a = fix $ \ ~(Just b) if typeOf a == typeOf b then Just unsafeCoerce a b else Nothing
```

This can be generalized to monadic computations using `gcast`.

```haskell
import Data.Maybe (fromJust)
-- fromJust :: Maybe a -> a
-- sends Just a to a, and is an exception on Nothing.
-- here, c is anything of kind * -> *
-- which is figured out automatically discovered by the compiler.
gcast :: (Typeable a, Typeable b) => c a -> Maybe (c b)
gcast c a = mcr
  where mcr = if typeOf (unc ca) == typeOf (unc $ fromJust mcr)
          then Just $ unsafeCoerce ca
               else Nothing

unc :: c x -> x
unc = undefined
```

Notice in this routine the undefined function `unc`. This is a perfectly fine idiom, and uses the fact that `typeOf` is not strict. One way to think about this is that `Typeable b =>` is like an implicit argument, bringing in the dictionary of functions for a method.

One way to do this (which is a little bit of boilerplate) is a function called `mkT`, which behaves like `id` except on one type:

```haskell
mkT :: (Typeable a, Typeable b) => (b -> b) -> a -> a
-- Example
newtype Salary = Salary Double deriving (Show, Data, Typeable)
raiseSalary :: (Typeable a) => a -> a
raiseSalary = mkT $ \(Salary s) -> Salary (s * 1.04)
-- Goal: raiseSalary () --> (), but raiseSalary 7 --> 7.28
-- Example implementation of mkT: uses that (->) is Typeable!
mkT f a = case cast f of Just g -> g a
               Nothing -> a
```

Notice the Haskell type magic: `g` is applied to `a` and then returned, so `g` has type `a -> a`, so `cast f` has type `Maybe (a -> a)`, and the compiler must know to use the `Typeable` dictionary of `(b -> b)` for the argument, and `(a -> a)` for the return type of `cast`. Here’s an alternate implementation.

```haskell
-- Standard function: default value, conversion, possible value (to be converted)
maybe :: b -> (a -> b) -> Maybe a -> b
maybe b _ Nothing = b
maybe _ f (Just a) = f a

-- super duper succinct: if we get nothing, just do nothing (id).
mkT f = maybe id id $ cast f
```

Relatedly, there’s a function called `mkQ` which computes over one type or returns a default value.

```haskell
-- Example
salaryVal :: Typeable a => a -> Double
salaryVal = mkQ 0 $\(Salary s) -> s -- 0 is the default value.

mkQ :: (Typeable a, Typeable b) => r -> (b -> r) -> a -> r
mkQ defaultVal fn a = case cast fn of Just g -> g a
               Nothing -> defaultVal

-- Example: salaryVal () --> 0.0, salaryVal (Salary 7) --> 7.0
```
Awesome. But what if one wants to do something for a few types, and something else for another type? They can be chained.

```haskell
extQ :: (Typeable a, Typeable b) => (a -> r) -> (b -> r) -> a -> r
extQ q f a = case cast a of
  Just b -> f b
  Nothing -> q a
```

Now, these can be chained, e.g. to create customized Show functions that operate on several types.

Though it feels like playing Calvinball with new rules, there’s yet another useful language extension, called **ExistentialQuantification**. This allows one to add type variables on the right side of a data declaration.

```haskell
{-# LANGUAGE ExistentialQuantification #-}
data Step s a = Done | Skip !s | Yield !a !s
data Stream a = forall s. Stream (s -> Step s a) !s
```

This says that, given a value `Stream a`, there exists a type `s` such that (and so on). Awkwardly, the same `forall` keyword is used to avoid adding a new keyword, so don’t mix it with the `forall` from `Rank2Types`. This is a very safe language extension; `Control.Exception` relies on it.

One application is Dynamic, an opaque type living in `Data.Dynamic` that can hold anything typeable. Then, we also have these functions.

```haskell
toDyn :: Typeable a => a -> Dynamic
fromDynamic :: Typeable a => Dynamic -> Maybe a
```

The implementation is pretty ugly (e.g. using `unsafeCoerce`), but can be implemented with existential quantification.

Another example is how GHC uses primitive, unsafe extensions, `raise#` and `catch#` (in `GHC.Prim`). This causes an unsafe coercion when `b` isn’t always the same type. This is problematic, so `SomeException` is used to implement safe, hierarchical exceptions, which all derive from `Typeable`.

```haskell
class (Typeable e, Show e) => Exception e where
toException :: e -> SomeException
toException = SomeException -- default, from SomeException
fromException :: SomeException -> Maybe e

throw :: Exception e => e -> a
throw e = raise # (toException e)

-- catch is a little more complicated (invokes the IO constructor, which is pretty deep)
catchX :: IO a -> (b -> IO a) -> IO a
catchX (IO a) handler = IO (catch# a (unIO . handler))
catch :: (Exception e) => IO a -> (e -> IO a) -> IO a
catch action handler = catchX action handler’
  where handler’ se = -- does the cast that we need, avoiding the low-level stuff
```

It’s now easy to create one’s own exception types, by making it an instance of `Exception` (getting the default methods for free). But types can also be created in a hierarchy, making it straightforward to catch some exceptions but not others.

`DerivedDataTypeable` also allows one to derive the `Data` class.

```haskell
data T a b = C1 a b | C2 deriving (Typeable, Data)
gfoldl k s (C1 a b) = z C1 ‘k’ a ‘k’ b
gfoldl k z C2 = z C2
toConstr (C1 _ _) = ... --encodes constructor number
toConstr C_2 = ...
gunfold k z c = case constrIndex x of
  1 -> k (k z c)
  2 -> z C2
```

This is convenient and builds impressively on a small amount of unsafe code, but runtime type checking is slow. Thus, one might wish to push generic programming to compile time. One interesting application of this is as follows.

```haskell
{-# LANGUAGE MultiParamTypeClasses, FunctionalDependencies, FlexibleInstances, TypeOperators, DefaultSignatures #-}
-- DefaultSignatures allows default signatures to not work for
-- all instances. It works wherever it works, which is pretty nice
```
-- for generic programming.

data Data tag contents = Data contents
data Cons tag contents = Cons contents

class Tag tag where showTag :: m tag c -> String

instance (Tag tag, Show c) => Show (Data tag c) where
  showsPrec d m@(Data c) = showParen (d > 10) $ \
    "Data\[" ++ showTag m ++ "]" ++ showsPrec 11 c rest

instance (Tag tag, Show c) => Show (Cons tag c) where
  showsPrec d m@(Cons c) = showParen (d > 10) $ \
    "Cons\[" ++ showTag m ++ "]" ++ showsPrec 11 c rest

-- Representing types like () with no constructor arguments
data Unit = Unit deriving Show

-- Representing constructor arguments
-- Requires TypeOperators language extension
-- makes a type that acts like ->, i.e. is an infix operator
-- basically, glues together two different types.
newtype Arg t = Arg t deriving Show
data (:*:) a b = a :*: b deriving Show
infixr 6 :*:

-- This isn't exactly what GHC does. The idea is that the compiler can generate
-- an instance for it.
class MetaData d m | d -> m, m -> d where
  fromData :: d -> m
  toData :: m -> d

data T1 = C1 deriving Show

data T1 _tag
instance Tag T1 _tag where showTag _ = "T1"

data C1_tag
instance Tag C1_tag where showTag _ = "C1"
type T1_meta = Data T1_tag (Cons C1_tag Unit)

instance MetaData T1 T1_meta where
  fromData ~C1 = Data (Cons Unit)
toData _ = C1

data T2 = C2 String Bool deriving Show

data T2_tag
instance Tag T2_tag where showTag _ = "T2"

instance MetaData T2 T2_tag (Arg String :*: Arg Bool))
instance MetaData T2 T2_tag where
  fromData ~(C2 s b) = Data (Cons (Arg s :*: Arg b))
toData ~(Data (Cons (Arg s :*: Arg b))) = C2 s b

-- We want to automatically derive instances of a Show-like class.
-- Or rather, we want the compiler to do this for us.
class MyShow a where
  myShow :: a -> String
default myShow :: (MetaData a m, MetaMyShow m) => a -> String
myShow = genericMyShow

instance MyShow String where myShow = show
Though this looks extremely intimidating, it allows instances of classes and types to work, which has all sorts of applications.

A DeriveGeneric extension allows the compiler to support a single Generic class that converts any datatype to a Rep class that can be computed over generically.

A lot of this lives in a module called GHC.Generics, which offers generic types U1 for units (constructors without arguments), M1 for meta-information (constructor names, etc.), and D, C, and S, for datatypes, constructors, and selectors, respectively. Selectors allow one to pick out record names from types. This module also uses :*: to glue multiple constructor arguments together, with the same fixity. Then, there are L1 and R1 for types with multiple constructors and K1 for concrete types. This means one needs OverlappingInstances, which is somewhat unfortunate.


“This might be some sort of emoji that looks like a TIE fighter. In fact, it’s probably Darth Vader’s TIE fighter.”

Today, we’ll try to develop some more intuition about abstract things in Haskell. Some of it is easy, e.g. \( \text{fmap} \ (\text{+}1) \ [1,2,3] \) returns \([2,3,4]\). But what is this actually doing to the list? Its length doesn’t change, so the function affects the elements, but not the container.

This is the same idea as for lists as functors: \((\text{my})\text{fmap}\) applies the function within the container. One could certainly make it return Nothing in both cases, but this would be naughty. As for efficiency, this is desugared into the same thing in Haskell’s intermediate representation (a language called Core) as the equivalent case statements.
newtype isn’t really a container; it exists only at compile time. Thus, Functor can be used on things that aren’t just containers. For example, we have the “tupling operator” (,) :: a -> b -> (a,b). But this is actually an operator, so one can do things like

```ghci
:type (,) 'X'
(,) 'X' :: b -> (Char , b)
```

Thus, the type signatures (,) Char b is identical to (Char , b), e.g. binky b = (,) 'X’ b. But this means we can write a MyFunctor instance for them.

```haskell
instance MyFunctor ((,) , a) where
  myfmap f (a, b) = (a, f b)
```

The idea is that we could pick either argument, but not both, and the reason we pick the second and not the first is purely a matter of convention. This is clearly a container, but newtype isn’t strictly a container, but behaves enough like one at compile time, which is pretty interesting.

Well, it turns out that -> is a type constructor: (->) a b is equivalent to a -> b, so we ought to be able to make functions instances of MyFunctor (or regular Functors), which is where the notion of functors acting on containers starts to go away.

```haskell
instance MyFunctor ((-> a) where
  myfmap f g = f . g
```

These are definitely not containers, so what makes a Functor a Functor? There are two laws that a “good” functor should implement (though this isn’t checked by the compiler or runtime): that fmap id == id, and (fmap f) . (fmap g) == fmap (f . g). But since types are curried, the type of fmap is (a -> b) -> (f a -> f b), so it takes a function and lifts it into the new context of the functor, e.g. a list, a Maybe, or a function.

But why functors? You’ll come across the damn things everywhere in Haskell, and they’re relatively simple (next to Monoid), and the fact that (->) is a Functor, but not a container, then we are forced to think at this higher level of abstraction of what a Functor actually “is.” In some sense, the container is a metaphor, but isn’t useful any more. Instead, we’ll call the f in instance Functor f a context. Maybe it’s a state, but in any case, it’s information we can use when doing our regular computation. For example, in lists, the context is a list, and for functions with the first argument supplied, the context is the function, and for I/O, the context is stuff that can have side effects.

A step up in power is applicative functors, defined in Control.Applicative.

```haskell
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b -- "Darth Vader’s TIE Fighter"
```

Notice how similar (<*>) is to fmap and $; they’re all just ways of doing function application. There four laws which applicative functors need to follow; the rules can be found at the Typeclassopedia. For a good example of Applicative, here’s Maybe:

```haskell
-- already provided in Control.Applicative
instance Applicative Maybe where
  pure = Just
  Just f <*> Just x = Just (f x)
  -- at least one is Nothing, so return Nothing
  _ <*> _ = Nothing
```

Exercise 9.1. If you want to gain a greater understanding of Applicative, write instances of it for lists, Identity, and (->) a. If newtype MyConst a b MyConst a, then write Functor and Applicative instances for it.

Now, we know enough to write a parser (!), using a well-known library for parsing, Parsec.

```haskell
import Control.Applicative
import Data.Char (chr)
import Numeric (readHex)
import Text.Parsec (char, hexDigit)
import Text.Parsec.String (Parser)

hexChar :: Parser Char
-- *> is defined in the Applicative module
hexChar = char '%' *> (combo <$> hexDigit <*> hexDigit)
   -- <$> is infix for fmap
where combo a b = case readHex [a,b] -- and so on
```
Now, we can parse an entire URL-encoded string, but this uses a thingy called Alternative. This is a subclass of Applicative, through the lens of monoids.

```
class Applicative f -> Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a
```

empty is useful for a fail state, and <|> makes for “try something, and if that fails, try something else instead.” This is useful for making combinators to separate a string by a separator.

```
pair :: Parser (String, Maybe String)
pair = (,) <$> many 1 urlChar
  <$> optional (char '=' *> many urlChar
```

```
urlChar = oneOf urlBaseChar
  <|> hexChar
  <|> ' '<$ char '+'
```

Every Monad is an Applicative: return is pure, but what happens for >>=? There’s a reverse bind function too.

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

...which means that <=< does the same thing as fmap and its friends: it injects a function into a context. But how does it differ?

Functor and Applicative are only able to look at the elements of a container (or context, or whatever): they are oblivious to the enclosing structure; both $\text{fmap}$ and $<$<$ cannot affect the context. Notice that with monads, we can; the function type is $a \to m b$. It is possible to see the context, to pattern-match on it, and so on, and return based on that. This is the crucial difference between monads and applicative functors — monads are more powerful, but harder to think about.

However, in the Haskell implementation, Monad is independent of these, so it’s not yet explicitly an instance of Applicative. This will be fixed in GHC 7.10, which ought to be out later this year.

Now, there are lots of subtly different things to use, so which should you use when? Here are some principles; in general, these boil down to simplicity.

- Use Applicative in preference to Monad, if possible.
- Similarly, the principle of least surprise means one should prefer Functor to Applicative whenever possible.

This also makes it harder for your own users to perform foot-shooting.

Perhaps it will be a surprise that these show up in practice in Facebook. One generic pattern is some database and server, with a service API. Specifically in the context of spam detection, a PHP request is sent to a server that detects whether something is spam or not. But since spam is always innovating, then there’s a whole team looking for emergent threats in the spam landscape. People write rules that tell these machine-learning algorithms what to do.

Of course, Facebook’s databases are distributed all over the place: there’s MySQL, memcache, and so on; and one request can cause others to propagate along the way. For example, querying about spam from a newly registered user could check the number of friends (Since many spammers don’t have friends on average), or even a series of fetches to get sophisticated data (age of friends’ accounts), and so forth. But we want to make this code sane, modular, abstract, and so forth.

It’s possible to use a monad for this, but this gets blocked at the first data fetch:

```
numCommonFriends x y = do
  fx <- friendsOf x -- blocked
  fy <- friendsOf y
  return $ length $ intersect fx fy
```

The monad is notationally convenient, but since the context of >>= includes the result of the previous computation, it gets blocked, so an applicative functor is better.

```
numCommonFriends x y = length <$> (intersect <$> friendsOf x <$> friendsOf y)
```

Since they don’t depend on each other, these two requests can be performed in parallel.

This system is called Haxl. A common thread to these kinds of applications of applicative functors and monads is that you get concise code that can be written quickly, and don’t depend on hacks in the way other languages’ parsing libraries sometimes do. For example, there’s a very good command-line parsing library called OptParse.
Today’s lecture was a guest lecture by Gabriel Gonzalez.

There are several functions that are akin to mapping over monads, e.g., `mapM`, `sequence`, and `replicateM`. These do all of their computation before returning the result, which is rather inefficient, both time-wise and memory-wise, and this doesn’t work on infinite lists. One solution, specified to the `IO` monad, is lazy I/O, but only works well for sources of information, not sinks or transformations, and makes it a lot harder to reason about programs: specifically, it invalidates the equational reasoning of a Haskell program. Finally, it seems like admitting defeat to the monadic paradigm.

Hence, pipes, which, like generators, are based on coroutines. The idea is to separate the generation of and the consumption of values, making things modular and easier to test and use.

```haskell
import Pipes
import System.IO (isEOF)

-- Producer is a generator
stdinLn :: Producer String IO ()
stdinLn = do
  eof <- lift isEOF
  if eof
  then return ()
  else do
    str <- lift getLine
    yield str
    stdinLn

useString :: String -> Effect IO ()
useString str = lift (putStrLn str)

-- What happens: until EOF, print the lines that the user enters
echo :: Effect IO ()
echo = for stdinLn useString
```

Cool. So how would we build this? The `Producer` type can be thought of as a list with effects inside, but also as a syntax tree of `Yield` values and a `nil` value:

```haskell
import Control.Monad.Trans.Class (MonadTrans(lift))

data Producer a m r
  = Yield a (Producer a m r)
  | M (m (Producer a m r))
  | Return r

yield :: a -> Producer a m ()
yield a = Yield a (Return ())

instance Monad m => Monad (Producer a m) where
  -- return :: Monad m
  return :: r -> Producer a m r
  return r = Return r

  (Yield a p) >>= return' = Yield a (p >>= return')
  (M m) >>= return' = M (m >>= \p -> return (p >>= return'))

  -- similarly for Return

  -- then, make it an instance of MonadTransformer with liftM

for :: Monad m
  => Producer a m ()
  -> (a -> Producer b m ())
  -> Producer b m ()
for (Yield a p) yield' = yield' a >>= for p yield'
for (M m) yield' = M (m >>= \p return (for p yield))

-- and so on for return
```

Another useful function is `runEffect`, which is a `Producer` that has no yield constructors (since they’re defined to return `Void`, a type without constructors). The idea is that it compiles down to Haskell code that you could write if you really
needed to, and ought to have similar semantics to lazy I/O, but invoking pipes makes writing the code a lot easier. The Pipes library also works well for exceptions, but that's a bit beyond the scope of the talk.

Behind this lies the theory of the library, which could be a useful model for how one should write good Haskell code, and even why Haskell is so well-liked in general. Haskell is unique in the ways it applies category theory to programming to reduce complexity.

In general, a piece of software might have lots and lots of components that get connected together in some ways. But typically, the more components, the more complex the system. Thus, we want a way to append things together such that we still get the same type in the end. This is exactly what a monoid does!

```haskell
class Monoid m where
  mappend :: m -> m -> m
  mempty :: m
  (<> :: Monoid m => m -> m -> m
  (<>) = mappend
```

This is familiar, and we also want <> to satisfy associativity, and with mempty its identity element.

So this works nicely with Yield: returning unit is akin to returning zero things, and yielding several things (e.g. with do or >>) is akin to adding things. This is an aspect of the fact that (>>>) and return () form a monoid within any monad, and satisfy the associativity and identity laws.

But monoids are somewhat of a simpler interface; we can generalize!

```haskell
class Category cat where
  (.) :: cat b c -> cat a b -> cat a c -- akin to mappend
  id :: cat a a
  -- akin to mempty
  (>>>) :: Category cat => cat a b -> cat b c -> cat a c
  (>>>) = flip (.)
  -- so we can read types left to right
instance Category (->) where
  (g . f) x = g (f x)
  id x = x
```

We require (.) to be associative, and id to be the identity with regards to (.).

It turns out that the "fish" operator >>= and return form a category (the former is defined as (f >>= g) x = f x >>= g), with the latter the identity.

Let's define (f ~> g) x = for (f x g) ("into"), so that (>>) and yield form a category, and obeys the laws. This simplifies the notation, and makes it more powerful.

Associativity in this category is a bit harder. We want the following to hold.

```haskell
for (for p g) h = for p (\x -> for (g x) h)
```

You could convince yourself with examples, or just thinking about what gets printed where.

The goal of the Pipes library is to use this to simplify the coroutines into one final syntax tree, so that it stays about as simple for all sizes.

Here's some information about the API. A Consumer is a sink that can change over time, e.g. prefixing a line with an increasing number (e.g. for echoing line numbers).

```haskell
import Pipes
import Pipes.Prelude (stdinLn)

numbered :: Int -> Consumer String IO r
numbered n = do
  str <- await -- we don't know where it'll come from just yet
  let str' = show n ++ ": " ++ str
  lift (putStrLn str')
  numbered (n + 1)

giveString :: Effect IO String
giveString = lift getLine

nl :: Effect IO ()
  -- >> is also called the feed operator.
  nl = giveString >> numbered 0
```
The Consumer type is much like the producer type, but instead of emitting the elements, it accepts them, so there's a `Await` primitive rather than a `Yield`. In much the same way as a Producer, these can be connected with `do`, and have some structure. We have a feed operator again:

\[
\text{\texttt{(\rightarrow)}} :: \text{\texttt{Monad \ m \ \Rightarrow \ Consumer \ a \ m \ \Rightarrow \ Consumer \ b \ m \ \Rightarrow \ Consumer \ a \ m \ c}}
\]

This is, once again, exactly the same as `<<<` in a category. Then, `await` is the identity, because `await \ \rightarrow \ f = f` and `f \ \rightarrow \ await = f.`

Now, it's helpful to connect Producers and Consumers, using the pipe operator `->>`. This connects the `await` of one to the yield of another.

\[
\text{\texttt{(\textbf{->>)}}} :: \text{\texttt{Producer \ a \ IO \ r \ \rightarrow \ Consumer \ a \ IO \ r \ \rightarrow \ Effect \ IO \ r}}
\]

```haskell
main :: IO ()
main = runEffect ( stdinLn >-> numbered )
```

The above program reads lines in and then prints them out with their line number. Thus, we have a Pipe type which can yield and `await`! For example, one could create a pipe which accepts two inputs and then terminates on the third (which is really hard to do in lazy I/O), so one could call it `take n`, and pipe `runEffect (stdinLn >-> take 2 >-> numbered)` and so on. Because pipes are monads, we can change them over time using `take` and so on.

Notice that the types have been unclear or somewhat overloaded. This is because there are multiple such operators with the same name, e.g. `>` can go between producers, consumers, and pipes. This is because a producer is just a pipe with the input sealed off (at least in theory; the implementation uses a better trick), so type `Consumer \ a = Pipe \ a \ Void`. This is parametric polymorphism (top-down, implemented from a big superclass).

Thus, `->>` and `cat` (exactly the same as the Unix command) form another category! The API is category-inspired, with `->>>` and `return`, `(\rightarrow)` and `yield`, `(\rightarrow)` and `await`, and `->>>` and `cat`. But this is just the beginning: `(->>>)` and `(\rightarrow)` distribute, in the sense of multiplication and addition, and `return \ \rightarrow \ h = return` (akin to \(0 \cdot a = 0\)).

Another advantage of this is that the category laws provide unit tests, in some sense, for the library. This framework keeps complexity simple, and though it seems intimidating, a small amount of theory goes a long way.

**Exercise 10.1.** Implement `takeWhile`, which passes lines on until a predicate is satisfied, and then terminates.

**Solution.**

```haskell
import Pipes
import Pipes.Prelude (stdinLn, stdoutLn)
import Prelude hiding (takeWhile)

takeWhile :: Monad m => (a -> Bool) -> Pipe a a m ()
takeWhile keep = do
  input <- await
  if keep input
    then do
      yield input
      takeWhile keep
    else return ()

main = runEffect $ stdinLn >-> takeWhile (/= "quit") >-> stdoutLn
```

**Exercise 10.2.** Implement `map`, which accepts a function and applies that function on everything piped through it.

**Solution.**

```haskell
import Pipes
import Pipes.Prelude (stdinLn, stdoutLn)
import Prelude hiding (map)

map :: Monad m => (a -> b) -> Pipe a b m ()
map f = do
  input <- await
  yield $ f input
map f

-- Alternate implementation
map f = for cat (yield . f)
```
import Control.Monad (replicateM_)
import Pipes

mystery :: Monad m => Int -> Pipe a a m r
mystery n = do
  replicateM_ n await
cat

Solution. This is akin to the \texttt{drop} function from the standard library: it ignores the first \(n\) inputs and then passes on all of the rest.

Exercise 10.4. What about this one?

import Pipes

yes :: Monad m => producer String m r
yes = return "y" >~ cat

Solution. This is the Unix \texttt{yes} utility, equivalent to \texttt{forever $ yield y}.

Finally, here’s a simplified \texttt{grep} utility, which only passes on strings that contain a given string somewhere in them.

import Pipes
import qualified Pipes.Prelude as Pipes

grep :: Monad m => String -> Pipe String String m r
grep str = Pipes.filter (isInfixOf str)

main = runEffect $ Pipes.stdinLn >>- grep "import" >>- Pipes.stdoutLn

11. Parsing and Continuations: 5/6/14

“Parking here is not my favorite activity.”

Today we’re going to talk about parsing, which will be a vehicle for talking about API and program design.

In times of yore, people wrote all of their parsers by hand. There’s lots of ugly C++ pointer arithmetic or Java boilerplate. These are extremely hard to debug, and it’s not clear from the code what they’re supposed to be doing. These do tend to be very fast, though. Alternatively, one can use a domain-specific language (e.g. Ragel), which makes a parser easier to understand and much more compact, but learning a new language (not to mention gluing it to other code) is difficult, and the generated code is hard to follow. But these make debugging much easier.

Let’s use Haskell! The simplest parser we can think of accepts a string to match against, then another string as input, and returns whether they’re equal. This ought to have type \texttt{String \rightarrow String \rightarrow Bool}. The problem is, this is not how parsers work: they return the result of input conversion (e.g. an integer parser returns the integer parsed), along with the rest of the input, so that parsers can be chained. Thus, a better type signature is \texttt{Type Parser s a = s \rightarrow Maybe (a, s)}. But parsers can fail, so it better is \texttt{Parser s a = s \rightarrow Maybe (a, s)}. Here’s a less crappy API for the equality tester string.

import Data.Char
import Data.List
import Data.Maybe

type Parser a s = s -> Maybe a s

string :: String -> Parser String String
string pat input =
  case stripPrefix pat input of
    Nothing -> Nothing
    Just rest -> Just (pat, rest)

number :: Parser String Int
number input = let (h, t) = span isDigit input
  in case reads h of
Now, one can chain these together: the following function parses HTTP version headers.

```haskell
-- version "HTTP/1.1/\r\n" == Just ((1,1), "\r\n")
-- version "HTTP/wibblesticks" == Nothing
version :: Parser String (Int, Int)
version input = do
  (_, input ') <- string " HTTP /" input
  (a, input ') <- number input '
  (_, input 3) <- string "."
  (b, rest ) <- number input 4
  return ((a, b), rest )
```

One naïve way to write this is to make a big staircase of case expressions. This boilerplate can be simplified with a function that encapsulates the case statement and returns the result (or nothing). But then, need to return the minimum and maximum, which involves calling the function \input -> Just (a, input). This is type-correct, but unintuitive, and hard to read.

But there's a better solution, involving some magic trickery: the function used to hide the case statements has the same type as the monadic bind operator, and the returning operator looks like monadic return. But this isn't quite a monad. it would be bad for Parser a s to be a monad, because then every function a -> (a, s) is infected with this instance. Instead, use newtype, calling Peter Parker "Spiderman," so to speak. Then, some constructors have to be passed around, but this isn't hard.

```haskell
-- Here, "oldParser" is the non-monadic type and wrapper function we used before
andThen :: OldParser s a -> (a -> OldParser s b) -> OldParser s b
bind :: Parser s a -> (a -> Parser s b) -> Parser s b
```

Then, one can make this an instance of Monad and Functor without too much code or fuss. Now, monadic notation makes parsing much easier. But then, since it's a functor, it can be turned into Applicative, making the parser much smaller.

```haskell
import Control.Applicative
import Control.Monad (ap)
instance Applicative (Parser s) where
  pure = return
  ( <<< ) = ap
```

Using various combinators, it can be reduced to just a few lines of code.

Another interesting option is to use Alternative to make a parser that can choose another parse if the first one fails.

```haskell
-- can be done since Parser s is already Applicative
instance Alternative (Parser s) where
  -- empty :: f a
  empty = P \input -> Nothing
  -- (<<>) :: f a -> f a -> f a
  f <!> g = -- some case statements follow
```

Parsing was one of the major reasons applicative functors were first introduced, even though they have found many more applications\[^5\].

The best-known parser for Haskell is called Parsec, but Bryan O'Sullivan has written a much faster version called Attoparsec. It's more specialized, e.g. to ByteString and Text, but not String. Attoparsec also doesn't give readable error messages (assumes it's parsing something automated such as network packets). Use either depending on these constraints.

Here's one thing people should be careful about. Suppose one receives a fragmented or incomplete TCP segment off of the network. The packet doesn't hold its message information (also true of JSON blobs), so how can we tell whether the object contain enough data. Also, can this be elegantly incorporated into a parser?

One solution is monad transformers, which allows one to stack things on top of other things. The composition of two monads is not in general a monad, but monad transformers allow one to stack “transformers” on top of a monad such that the result is still a monad. These are easy to work with, but not to implement. Parsec does this by stacking a transformer type ParsecT on top of IO. A function called Partial can handle partial application/parsing.

\[^5\]Pun intended?
Another way this works is with continuations. It’s not a great idea for your mental state to do this for too long. The idea is, if a parse succeeds or fails, one calls a function.

```
{-# LANGUAGE RankNTypes #-}

type ErrMSg = String
newtype ContP a = ContP {
  runP :: forall r.
    String -> (ErrMsg -> Either ErrMsg r)
    -> (String -> a -> Either ErrMsg r)
    -> Either ErrMsg r
}
```

But what is this language extension? For polymorphic functions such as `id`, there’s an implicit type quantifier not written, i.e. `id :: forall a. a -> a`. This universal quantifier, which is unwritten, is a type-level lambda, akin to the anonymous function declarer (one sometimes sees `Λ a → a`). In some sense, the function asks for a type, and applies it everywhere it sees it. This is a rank-1 type because the universal quantifier is present at the outermost rank. However, in the listing above, the `forall` is hidden. Thus, the type parameter `r` is controled by the calle (the library); the caller (the user) cannot access it. This is a rank-2 type, and higher nesting exists (but is rare). Working with continuations is tough! If you want to be convinced, try writing a `Functor` instance for them.

Let’s define types to track input that we’ve seen and what’s left to see, `Input` to `Accept`. Then, one can use these to allow the parser to backtrack if one branch parses unsuccessfully. Then, there are types

```
-- parse fails: error message, stack of context information
type Failure t r = Input t -> Added t -> More -> [String] -> IResult t r
type Success t a r = Input t -> Added t -> More -> a -> IResult t r

-- partial result: could succeed or fail.
data IResult t r = Fail t [String] String
  | Partial (t -> Iresult r)
  | Success t String r
```

As is usually the case with monads, the user-visible commands are relatively simple.

```
parse :: Monoid t => parser t a -> t -> IResult t a
parse m s = runParser m (I s) (A mempty) Incomplete failK successK
```

The last functions are “terminal continuations,” i.e. functions that don’t chain up another continuation, but return to normal computation:

```
failK :: Failure t a
failK i0 _a0 _m0 stack msg = Fail (unI i0) stack msg

successK :: Success t a
successK i- _a0 _m0 a = Done (unI i0) a
```

One interesting resource for information is #haskell on irc.freenode.net; there are lots of people willing to help with difficult Haskell questions. This has been true for years and years. For example, it’s hard to cajole the above kind of parser to rewind and recover thrown away information, so Bryan asked about this, and one well-known user suggested passing around integers as state instead. This is vastly easier, and allows one to write a function that takes a parser of any type to the number of bytes it’s read.

Nontheless, most of these don’t work when the grammar is ambiguous. For example, `string "foo" <|> string "foobar"` will not do what one expects on input on “foobarbaz”. But these can be detected by analyzing the abstract syntax tree form of the parser, which is a great use of applicative functors (not monads; they mess stuff up)!


These subjects are what got Professor Mazzières into Haskell in the first place.

Suppose one wants to incorporate untrusted code into a Haskell application, e.g. one wants to translate a webpage into Pig Latin. On Hackage, there might be some code with the function `toPigLatin :: L.ByteString -> L.ByteString`. This sounds good, but what if it contains `unsafePerformIO` calls that can do whatever they want?

Starting with GHC 7.2, the `-Xsafe` flag enables Safe Haskell, which disallows the import of any unsafe modules (e.g. `System.IO.Unsafe`, so there’s no `unsafePerformIO`); this is courtesy of the course TA, David Terei! There are also safe imports, via `import safe PigLatin (toPigLatin)`, which is enabled by `-XUnsafe`, and should guarantee
that `toPigLatin` doesn’t contain unsafe functions. But it’s turtles all the way down, so to speak: `toPigLatin` uses `ByteString`, which deep down in the call chain uses `unsafePerformIO` and other unsafe functions.

There’s another notion of safety: `-XSafe` means the compiler promises that it is safe, but `-XTrustworthy` is a flag that means that the author asserts that the module is safe (even if it contains unsafe code) — then, a module such as `Data.ByteString`, compiled with this flag, puts its unsafe operations in an inner module `Data.ByteString.Unsafe`. This bounces back to whether we trust the author of the package, but one can specify trustworthiness on a per-package basis, e.g. with `-fpackage-trust`, `-trust`, `-distrust`, and `-distrust-all-packages`. But if you don’t trust the author and the compiler verifies it, then that’s fine.

But what if the untrusted code needs to do actual IO? For example, actual translation needs to at least read files and (for example) also query Google Translate. The idea here is to use a restricted IO monad, the `RIO` monad. This creates type signatures such as `googleTranslate :: Language -> L.ByteString -> RIO L.ByteString`; then, we can make `RIO` functions with the same names as the IO monad which make this work. Here’s a hypothetical example.

```haskell
{-# LANGUAGE Trustworthy #-}
module RIO ( RIO () , runRIO , RIO . readFile ) where

-- Notice that UnsafeRIO isn't exported from the module.
-- Moreover, this is a compile-time construct -- there is no
-- incurred runtime cost!
newtype RIO a = UnsafeRIO (IO a)
runRIO :: RIO a -> IO a
runRIO ( unsafeRIO io) = io
-- Exposing RunRIO is also pretty bad, as you might imagine
-- this is really just IO, but with the right constructors.
-- unsafeRIO is the lift IO a -> RIO a
instance Monad RIO where
  return = UnsafeRIO . return
  m >>= k = UnsafeRIO $ runRIO m >>= runRIO k
  fail = unsafeIO . fail
-- Returns true iff access is allowed to the file name
pathOK :: FilePath -> IO Bool
-- e.g. only allow files in /tmp
readFile :: FilePath -> RIO String
readFile file = UnsafeRIO $ do
  ok <- pathOK File
  if ok then Prelude . readFile file else return ""
```

Using this sort of monad, one can force a program to only read and write files in some sandbox subdirectory, forbid execution of other programs, only allow connections to certain ports or servers, disallow access to devices, etc. There are similar policies enforced in java and JavaScript, albeit less clearly.

But this isn’t quite enough: one could make a `googleTranslate` function that is allowed to talk to Google Translate to translate email, and thus can store the logged email data in a temp file — and then email that temp file to an attacker. Thus, it’s important to keep track of which information is sensitive and where it goes.

The solution is something called Decentralized Information Flow Control (DIFC). Information Flow Control originated with military applications and classified data: every piece of data and thread (or process) has a label, and there’s a partial order `⊆` (“can flow to”) where `L_E ⊆ L_F` if `L_F` can access `L_E`. But `⊆` is transitive, so to prevent a file from being released to the Internet, one could set its label `L_E ⊆ L_e`, and buggy software that can access the file can’t push it to the Internet, because that would violate the transitivity. Sometimes, `L_∅` denotes public data, and if `L_A` and `L_B` are the things accessible by two users `A` and `B`, then `L_{AB} = L_A ∪ L_B` is the least upper bound in the lattice given by `⊆`: the things accessible by both users `A` and `B`.

```
  L_A
  ⊆

  ⊆

  ⊆

  L_B

  L_{AB}

  L_∅
```

The decentralization means that every process has a set `p` of privileges, and exervising privilege `p` changes the label requirements. There’s a more permissive partial ordering `⊆_p`. Thus, one can collapse the lattice above for two users `A` and `B` by reasoning about their combined privileges.

45
One way to do this in Haskell (not the only way) is to encode a lattice in Haskell's type system. There will be a type \( H \) for secret data and \( L \) for public data.

```haskell
-- define types H and L, and MkSec to wrap things in them

class Flows sl sh where
Flows H H
Flows H L
Flows L H
-- but not Flows H L!

Then, we introduce a `Sec` monad for each data type, where anyone can label a value, but privilege is needed to unlabel.

```haskell
newtype Sec s a = MSec a

instance Monad (Sec s) where
  return x = MkSec x
  MkSec a >>= k = k a

label :: a -> Sec s a
label x = MkSec x

unlabel :: Sec s a -> s -> a
unlabel (MkSec a) s = s 'seq' a -- (s is either H or L, and acts as a key)

-- can define relabel similarly
```

Now, untrusted code gets access to sensitive data only within `Sec` monads. For example, one might allow `Sec H` to only be sent to Google, but `Sec L` be sent over the network in general. Then, the implementation is provided by specific functions, using privilege and `unlabel`. But this is a bit too overpowered: the data itself shouldn’t be necessary to make a decision about its security.

Thus, one combines this with RIO to create a `SecIO` monad that has restricted access and the privilege system. First, we need some utility functions:

```haskell
-- allows a Sec value to be accessed within the Sec monad
value :: Sec s a -> SecIO s a
value sa = MkSecIO (return sa)

-- can return high values from SecIO L by wrapping in Sec
plug :: Less sl sh => SecIO sh a -> SecIO sl (Sec sh a)

-- includes security type
type File s = SecFilePath String

-- checks that the reading and writing is OK
readFileSecIO :: File s -> SecIO s (Sec s String)
writeFileSecIO :: File s -> String -> SecIO String
```

This is good, and now the untrusted function has type `untrustedTranslate :: Sec H L.ByteString -> SecIO H L.ByteString`, which has the correct restrictions. But this is static: we might want to update the lattice dynamically each time a user is added, but since it doesn’t exist at runtime, this needs to be done differently.

Thus, let’s do it dynamically, at runtime, using a `LIO` monad. This associates a `LIOState` to every thread, and also tracks the maximum label (clearance).

```haskell
{-# LANGUAGE Unsafe #-)

data LIOState l = LIOState { liolabel, lioclearance :: !l}
newtype LIO l a = LIOTCB (IORef (LIOState l) -> IO a)

instance Monad (LIO l) where
  -- usual implementation

  -- Now, we want some functions that execute trustworthy code.
  -- akin to liftIO, but safer
  ioTCB :: IO a -> LIO l a
  ioTCB = LIOTCB . const

  -- also some functions for getting and setting state
```

46
getLIOStateTCB :: LIO l (LIOState l)
getLIOStateTCB = LIOTCB readIORef -- and so on

Now, implementing labels is not hard:

module LIO.Label

class (Eq l, Show l, Read l, Typeable l) => Label l where
  lub :: l -> l -> l
  glb :: l -> l -> l
  infixl 5 'lub' 'glb'
  canFlowTo :: l -> l -> Bool
  infix 4 'canFlowTo'

For privileges, we want to know if one subsumes another:

class (Typeable p, Show p) => SpeaksFor p where
  speaksFor :: p -> p -> Bool

Then, we can implement CanFlowTo and so on. For example, suppose we have a hierarchy with the following definitions and privileges.

data Level = Public | Secret | TopSecret -- it turns out this is deriving Ord
data Compartment = Nuclear | Crypto
data MilLabel = MilLabel { level :: Level
  , compartments :: Set Compartment
  }

instance Label MilLabel where
  lub a b = MilLabel (max (level a) (level b))
    (Set.union (compartments a) (compartments b))
  glb a b = MilLabel (min (level a) (level b))
    (Set.intersection (compartments a) (compartments b))
  canFlowTo a b = level a <= level b
    && compartments a 'Set . isSubsetOf ' compartments b

Now, before reading and writing, one should check and possibly adjust the LIOState, checking if the data can flow to the user in question (and also vice versa for writing). Though this isn’t super easy, it’s much, much easier than the corresponding code on the OS level.

For privileges, we want to be able to talk about them in any context, so we’ll use them by wrapping them in a newtype.

class (Label l, SpeaksFor p) => PrivDesc l p where -- ... can downgrade data if has the right clearance, etc.

newtype Priv a = PrivTCB a deriving (Show, Eq, Typeable)

instance Monoid p => Monoid (Priv p) where
  mempty = PrivTCB mempty
  mappend (PrivTCB m1) (PrivTCB m2) = PrivTCB (m1 'mappend' m2)

privDesc :: priv a -> a
privDesc (PRivTCB a) = a
-- we can go from a privilege to a privDesc, but not vice versa.

But how should they be created in the first place? main runs unrestricted IO, so it can initialize privileges.

Most LIO functions have variants requiring privilege that take a P suffix, which then call internal privilege-checking functions. But privileges can also be delegated, etc:

delegate :: SpeaksFor p -> Priv p -> p -> Priv p

newtype Gate p a = GateTCB (p -> a) deriving Typeable

gate :: (p -> a) -> Gate p a

gate = GateTCB -- not exported

callGate :: Gate p a -> Priv p -> a
callGate (GateTCB g) = g . privDesc
Here, a gate is a wrapper around a function that allows things to work even in the case of mutual distrust.

Now, there are lots of functions we might want to use that need to be rewritten just slightly, which can be simplified via a blessTCB helper, through the magic of functional dependencies.

```
data LObj label object = LObjTCB !label !object deriving Typeable
-- actually in LIO.TCB.LObj
```

One of the main applications is a web framework called Hails, which works via multiple applications that might distrust each other. Communications are via policy models that make this security work. Another example is GitStar, which can host potentially private git repositories, and then, for example, call functions like a syntax-highlighting library without giving it access to copy the sources.

13. Zippers and Lenses: 5/13/14

“Sounding very clever is obviously one of the most important parts of learning Haskell.”

Today we will talk about Haskell programs that don’t go wrong for a bit: no infinite loops and crashes, there are exactly two values of type Bool, and three different values of type Ordering, LT, EQ, and GT. In Haskell 2010, there are even types with no constructors (for phantom types or type-level programming).

Consider the type `data NextType = A Bool | B Ordering`. This can have five possible values, and Either a b does the same thing (the sum of the number of values for a and b); thus, types such as this, with an |, are called sum types. Similarly, tuples are called product types, because one obtains the product of the individual numbers of values of the types. These are called algebraic data types, in that they are really doing algebra.

But they’re also useful for applications; for example, the criterion package for profiling has many nested types upon nested types, and then the record accession types interact nicely with function composition to make data access easy. But updating is ugly, since each explicit nest must be dealt with. Thus, we want to access fields within records, both for accessing and editing, such that both accesses and updates are composable. Haskell’s record syntax already handles 2 of them, but that’s not enough.

Let’s start from fairly simple building blocks, the tuple (a,b). To edit its second or first elements, one can write functions like these.

```
editSnd :: (b -> c) -> (a,b) -> (a,c)
editSnd f (a,b) = (a, f b)
-- editFst is the same
editFst :: (a -> c) -> (a,b) -> (c,b)
editFst f (a,b) = (f a, b)
```

In this idea, we focus on either the first or the second element.

We can count the number of possible values, representing dropping a field from a product type:

```
data Hole 3 a b c = AHole b c
  | BHole a c
  | CHole a b
```

Substitute in (x,x,x), which has $x^3$ possible values; then, the type `Hole3 x x x` has $3x^2$ different values... which is a derivative. This is not a coincidence; it also happens with other parameterized types.

Returning to 2-tuples, here’s a hole type.

```
data PairHole a b = HoleFst b
  | HoleSnd a
```

```
data PairZipper a b c = PZ c (PairHole a b)
```

```
-- Now, we have shinier implementations of focusFst and focusSnd
-- given the types, these are the only implementations that could possibly work.
focusFst :: (a,b) -> PairZipper a b a
focusFst (a,b) = PZ a (HoleFst b)
focusSnd :: (a,b) -> PairZipper a b b
focusSnd (a,b) = PZ a (HoleSnd a)
```

These types of functions are called zippers. The fact that the polymorphism forces there to be only one implementation is really nice.

Obviously, we need to be able to go the other way.
unfocusFst :: PairZipper a b a -> (a,b)
unFocusFst PZ a (HoleFst b) = (a,b)

-- now, getting the value we focused on
view :: PairZipper a b c -> c
view PZ c _ = c

-- this is the more fun part!
over :: (c -> c) -> PairZipper a b c -> PairZipper a b c
over f (PZ c l) = PZ (f c) l

Now, editing a record is as natural as unfocusSnd . over (+1) . focusSnd. This is pretty cool, but right now it only works on functions that return the same type as their arguments. This can be remedied by fiddling with the type signatures, but this quickly becomes ugly.

Let’s instead wrap everything up in one type:

data Focused t a b = Focused {
    focused :: a,
    rebuild :: b -> t
} \end{ lstlisting }

This is a pair consisting of a focused element and a function that knows how to reconstitute the original value.  

\begin{ lstlisting}
type Focuser s t a b = s -> Focused t a b
\end{ lstlisting }

Typically, the author of a library writes the function, and its user only needs to understand its type signature. Here, s and t might be the same, as might a and b, depending on whether editing changes the type. A Focuser is a function that takes a value and returns a Focused.

Now, some more machinery for working with these types.

unfocus :: Focused s a a -> s
unfocus ( Focused focused rebuild ) = rebuild focused

view :: Focuser s t a b -> s -> a
view l s = focused (l s)

over :: Focuser s t a b -> (a -> b) -> s -> t
over l f s = let Focused focused rebuild = l s
             in rebuild (f focused)

-- how we recast focusFst and focusSnd, with simpler names
_1 :: Focuser (a,b) (c,b) a c
_1 (a,b) = Focused a (\c -> (c,b))

_2 :: Focuser (a,b) (a,c) b c
_2 (a,b) = Focused b (\c -> (a,c))

Notice that if a and b are the same, then so are s and t.

Now, one can implement a relatively concrete example.

focusHead :: Focuser [a] [a] a a
focusHead (a:as) = Focused a (\a -> a)

The (:as) syntax defines a function \c -> (c:as), which is nice.

over and view have very similar types and implementations. One could abstract they via types.

wat :: Focuser s t a b -> (a -> f b) -> s -> f t

This assumes the function f is a type-level function; if the type-level identity function is passed in, then the type of over appears; if one uses const a instead, the type of view is obtained.

It turns out the type-level identity function does exist, and is defined in Data.Functor.Identity. There’s also a type-level constant function in Control.Applicative.

newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity a where -- and so on

newtype Const a b = Const { getConst :: a }

49
instance Functor Const a b where
  fmap _ (Const v) = Const v

Now we can introduce the abstracted type, called the `lens` type.

```haskell
{-# LANGUAGE RankNTypes #-}

type Lens s t a b = forall f. Functor f =>
  (a -> f b) -> s -> f t

over :: lens s t a b -> (a -> b) -> s -> t
over l f s = runIdentity (l (Identity . f) s)

view :: lens s t a b -> s -> a
view l s = getConst (l Const s)
```

This says that, given an `s`, this will focus on elements of type `a`. Using `over` to edit changes these types to `b`, and once editing is done, one gets back `t`, which might just be `s`.

Notice how powerful the type-checker is: it doesn’t just prevent errors, but defines how the program ought to work. And since `newtype` is only a compile-time construct, then there’s no incurred cost.

A language extension called `{# LANGUAGE TupleSections #-}` makes the function `(a,)` equivalent to \( \lambda b \rightarrow (a,b) \), which is notationally convenient, as in the following examples.

```haskell
{-# LANGUAGE TupleSections #-}

_1 :: Lens (a,b) (c,b) a c
_1 f (a,b) = (,b) <$> f a

_2 :: Lens (a,b) (a,c) b c
_2 f (a,b) = (a ,) <$> f b
```

-- Same idea works with _head, for the head of a list.

The applicative syntax means that we’re using a functor, but we don’t know which it is (the identity or the const functor).

A lens takes an action on a part of a structure into an action on the whole structure. Thus, it’s much easier to compose them together, and `view` and `over` correspond to getters and setters. Thus, this is somewhat like function composition, but very different than naïve getter and setter code.

Lenses are the first and most useful approach to composing editing of deeply nested structures (also first-class labels, semantic editor combinators, etc., which are subsumed by lenses). Thus, they show up more and more in real-world code, to make the API easy to work with and easy to use. It looks and is abstract, but is very useful.

There are a few different packages: `lens-family-core` is simpler and easier to use, and `lens` is incredibly comprehensive, abstract, powerful, and confusing. It contains anything you might ever want to use, and many, many things you won’t. `lens` is a bit controversial because it’s so big and hard to understand.

These libraries use some operators with different notation (specifically, they remind me of Perl). The getter or `view` is called `&`, and `over`, the editor, is `%~`, called “wiggly fish thing.” Then, another fish, is a setter: it accepts a value instead of a function. Finally, & is just \$_ with the arguments flipped. Thus, things can be composed with `foo & someField %~ ('a':) & otherField .~ 'b'`. These take some getting used to, but after that it’s all fine.

---

14. Web Programming: 5/15/14

"The methodology for this is: I found the graphic that made my point the best."

Today’s lecture was given by Amit Levy, a Stanford CS Ph.D. student. The goal is to think about web development and how and why one would do it in Haskell.

Web programming is obviously pretty useful these days, and even if one starts out by building something that doesn’t seem like a web-based application, it often might end up on the web anyways. And even discounting this, HTTP is becoming a general-purpose protocol for APIs, available in most languages and good server-side support (frameworks, SSL, virtual domains, ease of convincing the boss or whatever that it should be used).

There are lots and lots of frameworks for web applications. It used to be that Java was king, and Sun indeed pioneered the idea of web apps. But a lot of people didn’t like it; it was in many ways a bad language. It’s still in use, but not as trendy, falling out of favor towards Ruby on Rails (or Sinatra), Django with Python, server-side Javascript, PHP, etc.; in fact, PHP is far the most common language in use today.

These newer languages are all dynamic. Why is this? First, they tend to be less verbose (e.g. this Ruby code):

```
6 There is a bitwise and operator, but it’s surrounded by dots.
```
Compare to the equivalent (and much longer) Java code:

```java
def incr(y)
    y + 1
end
```

```java
protected static int x = 123;

public static int incr(int y) {
    return y + 1;
}
```

There are often other nice features such as closures (though Java 8 has them now!), or fast development and prototyping, and so on. People also believe that dynamic languages are really good for dynamic content (whereas something such as Scala is better for the more static backend). This is... pretty handwavy, but the idea is that typechecking lots of dynamic content is a pain.

A lot of these arguments are just arguments against Java. For example, Haskell has strong types but optional type declarations (in many cases), and has closures.

In general, a web application does three things:

1. parses a request from the client,
2. performs some side-effect actions (e.g. reading from a database), and
3. generates a response for the client.

For example, given a request (which in Haskell is modeled as some struct type) and a response type, then an application is just of type `Application = Request -> IO Response`. This is the essence of the WAI (Web Application Interface) package. This is a common interface between servers and applications, with the goal of being able to mix and match, such as the several Haskell-based servers and frameworks (e.g. Yesod, Hails, Scotty).

The actual base types are a little but complicated. They just correspond to all of the fields in an HTTP request or an HTTP response.

```haskell
-- status is the HTTP status, akin to the return value of a Unix process.
data Response
  = ResponseFile Status ResponseHeaders FilePath (Maybe FilePart) -- sends a static file
  | ResponseBuilder Status ResponseHeaders Builder -- builds a dynamic file
  | ResponseSource Status ResponseHeaders (forall b. WithSource IO (C.Flush Builder) b)
  | ResponseRaw (forall b. WithRawApp b) Response

type Application = Request -> IO Response
```

For example, here's a very basic app.

```haskell
-- Note: need to call 'cabal install wai warp' to make this work
module Main where

-- Can be simplified by using the OverloadedStrings language extension... but same end result.
import qualified Data.ByteString.Lazy.Char8 as L8
```
import Network.HTTP.Types
import Network.Wai
import Network.Wai.Handler.Warp (run)

app :: Application
-- responseLBS uses the response constructor to make a byte string
-- arguments: status, any arguments, data
app req = return $ responseLBS status 200 [] $ L8.pack "Hello, World"

main :: IO ()
main = run 3000 app

This is pretty concise compared to other web apps. Even Rails requires lots and lots of files to make things work. Simple is a web framework with a single Controller type, which looks like the Application type from above. In fact, it's a small wrapper.

newtype Controller s a = Controller {
  -- the Either means to either return and send the response to the client, or continue running
  -- similar to monad transformers StateT and ReaderT
  runController :: s -> Request -> IO (Either Response a, s)
}

-- These mean it's easier to refer to the controller in different contexts
instance Monad Controller
instance Applicative Controller
instance MonadIO Controller

Then, there are a bunch of combinators to make life easier.

-- Stop computing and respond to a request.
respond :: Response -> Controller s a
okHtml :: ByteString -> Response
notFound :: Response

-- e.g. respond (okHtml "Hello world")

-- Get the request state and app state
request :: Controller s Request
controllerState :: Controller s s

-- Parse query and form parameters. Return strings or stringlike things (e.g. ByteString)
queryParam :: Parseable p => Controller s p
parseForm :: Controller s ([Param], (ByteString, FileInfo ByteString))

Then, there are also a bunch of routing combinators, which make it easier to do common tasks. Routing is how different types of functions or pieces of code respond to different kinds of code. Thus, there are a bunch of functions to match on patterns and then respond accordingly. Each takes a sub-controller as an argument, which is what gets run if the match succeeds; thus, it's possible to nest these combinators to make more complicated routers.

-- Match on next dir in path
routeName :: Text -> Controller s () -> Controller s ()
routeName "articles" $ ...

-- Treat first dir in path as query param
routeVar :: Text -> Controller s () -> Controller s ()
routeName "articles" $ routeVar "name" $ ...

-- Match whole pattern of path
routePattern :: Text -> Controller s () -> Controller s ()
routePattern "/articles/:name" $ ...

-- Match if no path left
routeTop :: Controller s () -> Controller s ()

-- Match on request method
routeMethod :: Method -> Controller s () -> Controller s ()
-- example of nesting
routeMethod GET (routePattern "/articles/:name")
routeHost :: ByteString -> Controller s () -> Controller s ()

And of course, there are things built on top of those.

get :: Text -> Controller s () -> Controller s ()
get ptrn ctrl = routeMethod GET $ routePattern ptrn ctrl

post :: Text -> Controller s () -> Controller s ()
post ptrn ctrl = routeMethod POST $ routePattern ptrn ctrl

-- Example of small application.
-- base directory responds with "Hello World"
-- /foo responds with "bar"
myapp :: Controller s ()
myapp = do
    get "/" $ respond $ okHtml "Hello World"
    get "/foo" $ respond $ okHtml "bar"

The next thing one might want to use is an object relational mapper (ORM). There’s a PostgreSQL ORM (for a specific open-source variety of SQL), which correspond to types with some kind of key, a DBKey type and some data. For example:

data Article = Article
    { articleId :: DBKey
    , articleTitle :: Text
    , articleBody :: Text
    , articleShortName :: Text }

class Model a where
    modelInfo :: ModelInfo a
    modelRead :: RowParser a
    modelWrite :: a -> [ Action ]

data DBKey = DBKey ! Int 64 | NullKey

data ModelInfo a = ModelInfo { modelTable :: ByteString
                                , modelColumns :: [ ByteString ]
                                , modelPrimaryColumn :: Int
                                , modelGetPrimaryKey :: a -> DBKey }

One way to reduce the amount of boilerplate is to make a model derive from Generic.

{-# LANGUAGE DeriveGeneric #-}
import GHC.Generics

data Article = Article
    { articleId :: DBKey
    , articleTitle :: Text
    , articleBody :: Text
    , articleShortName :: Text } deriving ( Show , Generic )

instance Model Article

This provides access to three functions, which save, search, and retrieve entries in the database.

-- save a row to the database
save :: Model a -> Connection -> a -> IO ()
-- find all rows matching the constraint
-- not lazy! There is a lazy findAll function in the library, though
findAll :: Model a -> Connection -> IO [ a ]
-- retrieve the values for a given key
findRow :: Model a -> Connection -> DBRef a -> IO ( Maybe a )

Because Haskell is strongly typed, some common edge cases are already irrelevant: fields can’t be null, unless they have a Maybe, and fields have their known, strict types. This makes a lot of validation redundant.

One can initialize an app via the command-line command `smpl create --templates --postgresql my_cms` (which initializes a database, etc.). Then, the main function is in one file, and the run function for the application lives in another.
A content management system (CMS) is a method of, for example, generating pages with a form, and having a central homepage which lists all of the other pages. This can be done by initializing a database within Haskell with the `create_table` call. This all looks pretty cool, but involves installing software that doesn’t work on many kinds of computers (e.g. my own, running Mac OS 10.6.8), so some amount of confusion has to be overcome before any of this can happen. So, uh, here’s some more code, I guess.

```haskell
{-# LANGUAGE OverloadedStrings #-}
module Application where
import Control.Monad.IO.Class -- for liftIO
import MyCms.Common
import Web.Simple
import Web.Simple.Templates
-- for putting articles in a database
data Article = Article {
  articleID :: DBKey,
  articleTitle :: String,
  articleBody :: String
} deriving (Generic)
instance Model Article where
  -- separates words by underscores to get some string for it.
  -- some complications for backwards compatibility
  modelInfo = underscoreModelInfo "article"
app :: (Application -> IO ()) -> IO ()
app runner = do
  settings <- neeAppSettings
  runner $ controllerApp settings $ do
    routeTop $ do
      articles <- withConnection $ \conn -> liftIO (findAll conn :: IO [Article])
      render "index.html" ()
    -- Then: routes go here.
```

15. Performance: 5/20/14

“When I got started in Haskell a while ago, it was a miracle when a program worked at all.”

Nowadays, though, we can make our code go very fast, and it feels sad when it can’t. There are two perennial questions: why does my program take so long, and why does it take up so much memory? This is particularly tricky in Haskell because of lazy evaluation.

The standard benchmarking library for Haskell is `criterion`, written by Bryan O’Sullivan, as follows.

```haskell
import Criterion.Main -- via 'cabal install criterion'
len :: [a] -> Int
len (x:xs) = 1 + len xs
len _ = 0
main = defaultMain [bench "len" (whnf len [1..100000])] 
```

The naming might be nonintuitive, but the idea is that the benchmark is named "len", and is of function `len` of the input `[1..100000]`.

The thing being benchmarked is the `whnf`, which indicates the time taken to reduce the function and argument to weak head normal form (WHNF) (there’s also `nf`, which reduces it to the related normal form). The idea here is that, since Haskell is lazily evaluated, a function such as `len` can return a thunk rather than the result, so the normal form is a way of specifying that it should be strictly evaluated. For example, the internal representation after using `seq` forces evaluation. This is what allows things like infinite lists to work: the representation never fully evaluates (or else...). Importing `Control.DeepSeq` allows for full evaluation (don’t do this on infinite lists), and this corresponds to normal form (rather than weak-head normal form). One can look into the internal representation and see “black holes” which forward things to the final object, and are cleaned up by the garbage collector. Try installing `ghc-heap-view` to see this (which requires some magic). Here, we use weak-head normal form, because there’s only one constructor for `Int`. Note that weak-head normal form won’t go into infinite loops, but `nf` can.
Anyways, the criterion library provides both kinds of forms, and provides lots of nice information. In addition to the times taken to run the function, it also estimates the clock resolution, error, and time cost of a clock call in order to estimate the variance and the number of times it needs to run a benchmark in order to get a reasonable answer. It also estimates variance due to noise (which can come from a huge number of sources, from background programs to Dwarf Fortress to strange decisions made on Macs, such as the desktop updating live previews every time an item is changed). Thus, the library reports how much of the standard deviation and variance are due to noise. This can be reduced, but not eliminated.

Another interesting way to do this is to dump a report as an HTML file, with lots of sleek pictures and Javascript!

There are two other ways to run a benchmark, \texttt{whnfIO} and \texttt{nfIO}, which accept an IO action, run it, and then return the result.

Now, let’s talk about space complexity. This requires using the \texttt{ghc} flag \texttt{-rtsopts}, which is off by default because it can cause security leaks. This allows \texttt{criterion} to provide information about space complexity (total memory used, peak used, stats about the garbage collector).

One interesting fact about Haskell’s garbage collector is that many of the objects on the heap die really quickly, and others last a long time (which is true of many programming languages), so things are by default placed in one sector of the heap, called Gen 0 or the nursery, because they’re expected to die soon (which is a really morbid extended metaphor!); then, as it persists (so if it’s not collected), it is moved to Gen 1, which is scanned less often. Once again, it’s possible to get more interesting heap data and pretty-print it to postscript files.

Of course, if you try to benchmark \texttt{len} with twenty million elements, it runs out of stack space quickly, which I guess is a crude form of profiling. It indicates something is horribly wrong with a length program… in the very least, it should be possible to run it in not very much space.

Thanks to lazy evaluation, this appears in memory as a giant sum $1 + 1 + \cdots$, and is evaluated by the stack. Ouch. Notice that this isn’t lazy evaluation: the same code written in ML or Python or C would have the same problem. Thus, maybe the following code is better.

\begin{verbatim}
len2 xs = go 0 xs
  where go n (x:xs) = go (n+1) xs
        go n [] = n
\end{verbatim}

This seems a bit faster than \texttt{len}, and uses no stack allocation, which is another improvement. Can we do better? It’s possible that the strictness analyzer, which lives in \texttt{ghc-land}, tries to discover whether a recursive function’s result is only used for the last result, and optimize accordingly. When this is turned off (`-00`), there’s a much more interesting profile: there are lots of thunks, and a reasonable amount of stack allocation. But then, using \texttt{seq}, it’s possible to make the program take less space (but still a lot). However, looking into the \texttt{GHC} summary statistics, 80\% of the time is spent garbage collecting unless \texttt{seq} is used. That isn’t good.

One good way to understand the code that is being benchmarked is to look at the intermediate representation, a language called Core. Think of this as Haskell without the syntactic sugar. One can access the IR by calling \texttt{ghc -c -ddump-simpl Length.hs}. This is harder to read, of course, but is lower-level and corresponds much more to what the program actually does. For example, \texttt{+} becomes \texttt{GHC.Num.+ # GHC.Types.Int} (i.e. instantiated to the \texttt{Int} type), and so on. But relatively little needs to be understood, and it’s also true that relatively little of it is actually hard to understand. Another name for Core is System FC (which comes up in papers and the \texttt{GHC} wiki). Core is then transformed into another intermediate language, to which several optimizations are applied.

In System FC, all evaluation is controlled via the \texttt{case} expression, which demands things to be evaluated to \texttt{WHNF}.

\begin{verbatim}
--Haskell
foo (Bar a b) = (- ... -)
-- Core
foo wa = case wa of _ { Bar a b -> (- ... -) }
\end{verbatim}

It’s a very common mode of failure for new Haskell programmers to ignore this simplified code and instead use strictness (bang patterns, \texttt{seq}, and \texttt{DeepSeq}) everywhere, which is not necessarily all that useful. Instead, it’s better to think about what is actually going on.

One common theme is that linked-lists are not all that efficient space-wise (and time-wise), but it’s the default for lists. This is an artifact of early Haskell, when people were surprised at all that it was possible to get a lazy program to run efficiently. Thus, for text applications it may be nicer to use \texttt{Data.ByteString}: often, there’s a 10- to 50-times speedup. However, this is bad for internationalization, so there’s a \texttt{PackedText} type for Unicode code points, and a \texttt{Vector} type up on Hackage which supports packed types (and is good for mathematical work). These aren’t the final word either, because huge files don’t work well for strict types… thus, each of these comes with a lazy variant. But provided that the cost of dealing with an item in a list is more than the cost of traversing the list, Haskell lists are better — the answer in general is “it depends.”
Today’s lecture was given by David Terei, the TA.

Today’s class will provide a way of understanding of how the higher-level constructs in Haskell boil down to actual machine code. The basic pipeline is Haskell → GHC Haskell (i.e. raw Integers, unsafePerformIO, and so on) → an intermediate representation called Core (a small functional language), to another small functional language called STG, through an imperative language called Cmm or C-- (which is basically a pretty-printer for assembly, sort of like LLVM but less popular), to assembly.

One of the first things GHC does is support Haskell on top of an unsafe variant called GHC Haskell, which has primitive types such as Int#, Array#, TVar#, states, exceptions, and such. These live in GHC.Prim, and offer higher performance but are unsafe. This is a nice way to implement Haskell. All variants of Int are represented internally by an Int#.

```haskell
data Int32 = I32# Int # deriving (Eq, Ord, Typeable)
instance Num Int32 where -- ...
```

Thus, using Int8 to save space isn’t actually saving space, at least right now.

For I/O, it’s a state-passing monad.

```haskell
newtype IO a = IO (State# RealWorld -> (# State# RealWorld, a #))
returnIO :: a -> IO a
returnIO x = IO $ \s -> (# s, x #)
bindIO :: IO a -> (a -> IO b) -> IO b
bindIO (IO m) k = IO $ \ s -> case m s of (# new_s , a #) -> unIO (k a)
```

The RealWorld token is passed throughout to ensure that the actions are ordered in the way one would want for I/O. In some sense, I/O is really a bunch of transformations on the world! The RealWorld token is only a compile-time construct, and goes away in the assembly code.

To write commands such as unsafePerformIO, one simply throws away the RealWorld token, and then can more or less do what it wants.

The next step in the pipeline is Core, a small functional, lazy language. It has only variables, literals, let, case, lambdas, and one other idea, and it has a very small grammar. Everything else in Haskell boils down to this. Then, the majority of the optimizations are performed on core.

```haskell
-- Haskell
id :: a -> a
id x = x

idChar :: Char -> Char
idChar = id

-- Core
id :: forall a. a -> a
id = \ (@ a) (x :: a) -> x

idChar :: GHC.Types.Char -> GHC.Types.Char
idChar = id @ GHC.Types.Char
```

Then, where statements become let statements, pattern matching becomes case statements, and typeclasses become data structures, akin to dictionaries.

```haskell
-- Haskell
typeclass MyEnum a where
toil :: a -> Int
fromId :: Int -> a

-- Core
data MyEnum a = DMyEnum (a -> Int) (Int -> a)

-- basically, we're passing around the MyEnum dictionary
toil :: forall a. MyEnum a -> a -> GHC.Types.Int
toil = \ (@ a) (d :: MyEnum a) (x :: a) ->
  case d of
    DMyEnum f1 _ -> f1 x
Sometimes, one has to lift one dictionary to another, e.g. if one wants to use Maybe. The code that gets generated takes
in the type parameter @a and a dictionary MyEnum a.

The {-#UNPACK#} pragma makes Core store the raw types, which leads to better performance, because then data
structures operate on direct integers, rather than pointers to them. For example, consider an unpacked Point type with
two coordinates. Then:

```haskell
{-# UNPACK #-}

addP :: P -> Int
addP (P x y) = x + y
```

However, this is only usually a good idea. Sometimes it isn’t, so it’s necessary to be mindful of the places where it will
be a performance loss. For example, if the function is inherited from some polymorphic type, bad stuff happens.

```haskell
module M where
{-# NOINLINE add #-}
add x y = x + y

module P where
addP P x y = add x y
```

In Core, this forces the code to unbox the points, then box them again. Core is a great tool to see how fast one’s code
performs; all evaluation is via case statements.

Now, many of Haskell’s optimizations are done as Core-to-Core transformations. For example, functional languages
gain a huge boost from inlining (perhaps about twice as much as imperative languages). A naïve implementation of the
factorial function leads to lots of thunk allocating and dereferencing pointers and such, but the compiler can reason that
the machine-level operations will be ok (unless it’s undefined, which throws an exception). Thus, it’s possible to unbox
the Ints in the function, which makes the Core code look much more imperative and efficient. However, the original
function is still kept so as to preserve the same interface.

Another big optimization is something known as SpecConstr. The idea here is to increase the amount of strictness in
functions. For example, the following function isn’t strict in its first argument.

```haskell
drop :: Int -> [a] -> [a]
drop n [] = []
drop 0 xs = xs
drop n (x:xs) = drop (n-1) xs
```

Thus, drop ⊥ [] = [], but drop ⊥ (x:xs) = ⊥.

Thus, it compiles into the following code.

```haskell
drop n xs = case xs of
  [] -> []
  (y:ys) -> case n of
    1# n# -> case n# of
    0 -> []
      -- without this optimization, more boxing and unboxing.
    _ -> drop’ (n# -# 1#) xs

-- works faster with unboxed n
drop’ n# xs = case xs of
  [] -> []
  (y:ys) -> case n# of
    0# -> []
    _ -> drop’ (n# -# 1#) xs
```
This can cause an explosion in the code size, so GHC sets limits on what can be done.

After Core, GHC compiles to another languaged called STG. It’s similar to Core, but laziness and allocation of thunks is very explicit. case is the only place evaluation occurs (like Core) but let is the only place allocation occurs. There’s an idea called graph reduction, which is an interesting way to think and reason about how Haskell code is evaluated.

```markdown
↓
↓
↘
↓
↓
→
→
↘
```

This is true for functional languages in general. The idea is that the leaves are evaluated, then the nodes that depend on them, and so on.

Haskell represents a lot of data as closures: a header and a payload, with a pointer to the code and metadata for the closure (largely used for the garbage collector). These are used for everything: even the value 10 is represented with a closure. For example, a data declaration has a header, then some pointers, then some non-pointers, depending on the structure of the type. Function closures have entry code equal to the function code. Closures also provide a nice framework for partial function application (one can store information about the arguments and the function they should be applied to). Finally, thunks are closures, which represents the data already computed (so that multiple users of the thunk don’t have to recompute data).

The calling convention is to pass some arguments in registers (in 64-bit mode; in 32-bit mode, all arguments are passed on the stack) and then others on the stack. The register R1 often stores the current closure in x86.

Thunks should update their node in a graph to be the computed value once they’re evaluated; GHC uses a self-evaluating model, where a thunk updates itself in the graph.

```haskell
-- very simple thunk
mk :: Int -> Int
mk x = x + 1
```

Here’s the code GHC generates for this.

```c
// thunk entry - set up stack, evaluate x
mk_entry ()
entry:
    if (Sp - 24 < SpLim) goto gc; // need more space; ask the garbage collector
    I64[Sp - 16] = stg_upd_frame_info; // sets thunk to be updated
    I64[Sp - 8] = R1;
    I64[Sp - 24] = mk_exit // set up continuation
    Sp = Sp - 24;
    R1 = I64[R1 + 8]; // so that x is the only data in the closure
    jump I64[R1] (); // evaluate R1. Since it’s a closure, everything can be evaluated.
    // maybe it’s a thunk.
    gc: jump std_gc_enter_1 ()
}
// something analogous happens in the exit case
mk_exit ()
entry:
    Hp = Hp + 16;
    if (Hp > Hplim) goto gc;
    v::I64 = I64[R1] + 1;
    I64[Hp - 8] = GHC_TYPES_I_con_info; // set up an Int closure
    I64[Hp + 0] = v::I64;
    R1 = Hp;
```

58
The function `stg_upd_info` takes a thunk and replaces pointers to it with pointers to its value once it's been evaluated. This seems like quite a nice synchronization issue, and it seems like thunks should evaluate atomically, but in this case race conditions are OK, because all this does is cause duplication of work. Thunks evaluate deterministically, so this is OK, and the chance of a race occurring is pretty small anyways.

The thunk closure has a result slot, which is empty when it isn't yet evaluated. Then, the update code fills in the result slot, and then changes the pointer in an information table. This is safe in many, many architectures, because word-aligned writes are atomic, so there may be work duplicated, but there won't be any corruption.

In particular, since pointers have to be word-aligned, the last two (three on 64-bit architectures) must be zero, so they can be used to store extra information. Thus, GHC uses these bits at the end of a pointer to store information about what the pointer points to: if it's a constructor that has been evaluated, the tag contains the constructor number, and for a function, it provides the arity of the function. For unevaluated thunks, these are unaligned. The constructors are per type, so this is actually more efficient than one might imagine. This makes the C code above a little more complicated, but not terribly so.

Pointer tagging makes one's own data types more efficient: if the closure is a constructor, then a parameterized type `data MyBool a = MTrue a | MFalse a` is as efficient as using an `Int#` (that is, unboxed!) to represent `True` and `False`.

17. The GHC Runtime System: 5/27/14

Today's lecture was given by Edward Z. Yang!

Last time, we saw the chain from Haskell to Core to STG to C-- to assembly, but this doesn't actually create executable code; instead, there's a bunch of code compiled from C, called the runtime system, linked with every Haskell program. This is important for performance, allowing one to more easily answer questions such as the ones at [http://stackoverflow.com/questions/23462004/code-becomes-slower-as-more-boxed-arrays-are-allocated](http://stackoverflow.com/questions/23462004/code-becomes-slower-as-more-boxed-arrays-are-allocated) and [http://www.quora.com/Computer-Programming/Why-are-Haskell-green-threads-more-efficient-performant-than-native-threads](http://www.quora.com/Computer-Programming/Why-are-Haskell-green-threads-more-efficient-performant-than-native-threads).

Finally, there are many different runtime systems, most famously the JVM. These have a lot in common with those of Haskell, so learning about one leads to more information about the rest. But of course, runtime systems can be interesting in their own right.

There is a lot contained in Haskell's runtime system: a storage manager (garbage collector and block manager), a scheduler, a dynamic linker (so that object files can be loaded at runtime in GHCi), the STM implementation, profiling (both heap profiling and time profiling), and more. One could spend many hours talking about these; this lecture will focus on only the first two.

**Garbage Collection.** Recall that garbage collection allows programmers to allocate indefinitely and reclaim memory that is no longer in scope. One standard scheme is reference counting, which keeps track of a number of pointers to each point in memory and deallocates something when its pointer count hits 0. However, this is time-intensive, and doesn’t handle loops well. Nonetheless, this or variants of it are used by PHP, Perl, and Python. An alternative (more involved) is mark-and-sweep, which marks out objects that are accessible from objects in scope, and then sweeps the entire heap for unmarked memory and deallocates it. This is also pretty simple, and is used by GoLang and Ruby. However, this algorithm causes fragmentation (swiss cheese memory, so to speak), which makes allocation more complicated. This can be mitigated by compacting the heap after sweeping, but there’s another issue, which is that sweeping the entire heap takes quite some time.

This leads to a generational copying collector, used by GHC and the JVM. The generational hypothesis claims that most objects die young, which is reasonable in general and even more so in functional languages! This leads to a nicer algorithm.

First, consider the stop-and-copy algorithm, used by a copying garbage collection. There are two phases, called evacuation and scavenging. The heap is divided into a “from space” and a “to space,” and allocated memory only can live in the from space. For evacuation, allocated memory (in order from a root pointer) is copied into the “to space,” but the pointers aren’t adjusted. Instead, they’re kept as forwarding pointers, which makes this act like a queue, where objects pointed to aren’t yet moved over: if $A \rightarrow B$ and $A$ is live, then so is $B$, so during scavenging, the new copy of $A$ has its pointer moved to the new copy of $B$, and the old copy of $B$ is updated to point to the new copy (so other references can forward their pointers). Once scavenging is done, the only objects alive in the “to space” are the live objects from before, so the “to space” and “from space” can be switched. This is nice in that the newly allocated memory is contiguous, so there’s no need to play with a free list.

But GHC uses a modified version of this, called generational copy collection. In this algorithm, there are multiple “to spaces.” A generational garbage collector works by partitioning the heap into several generations, and garbage-collecting...
them at different times. Thus, the generational copy collector has two (or more) “to spaces,” one called a nursery and one
called generation 1. The idea is that over two minor garbage collections, some objects might not persist, so they stay in
the nursery, which is garbage-collected more frequently.

Since Haskell is functional, then it’s usually the case that newly allocated objects only point to older ones, which
makes life nicer sometimes. But another interesting fact is that the more garbage there is, the faster this runs, because
there’s less to copy. This is unlike other algorithms, and makes it faster (though still not as fast as manual deallocation
of memory in C, of course).

The above algorithm is bog-standard, but difficult to implement: it’s beautiful, but used in very few languages. A
common problem is that accurate garbage collection requires knowing what the pointers are (e.g. via an object’s info
table; in particular, which fields in an object are or aren’t pointers). This is annoying to code, generate, or maintain,
making this less appealing.

But this isn’t even the most important issue: the assumption before (that older objects can’t point to newer ones) isn’t
true. For example, IORefs make this difficult. For example, one might get a pointer from Generation 1 to the nursery —
but a minor garbage collection will only look at the nursery, miss this pointer, and be incorrect. The standard solution is
to add mutated objects to a set called the mutable set (also called the remembered set), which is examined in minor
garbage collections (and cleared in a major garbage collection). This is sufficient to preserve correctness, but isn’t the
whole story.

This is hard to implement: every time a pointer is mutated, another object needs to be added to the mutable set.
This is bad in languages such as Java, where there can be a lot of pointer mutation; thus, this leads to a factor of two
performance hit, and thus other ideas are implemented (such as marking specific portions of memory to also look at) —
but this is much more of a problem in Java than in Haskell. Mutation is rare in pure languages like Haskell, and if you
are mutating a lot of IORefs, then perhaps there are other issues with your program.

... which is to say, IORefs are almost always used for long-distance communication, rather than computations.
Furthermore, mutations in general are quite rare. Laziness, though, is also a kind of mutation (thunks can be updated,
causin pointers to be where one might not expected). One solution is to promote the result of a thunk to Generation 1,
so that there’s less fussiness with mutable sets. This could work for IORefs, as in correctness, but this leads to lots of
things being copied into Generation 1 really quickly, and then dying and being missed in minor collections.

The standard caveat is that copy collectors require twice as much memory, for the two sections, but there are various
tricks which make this less of an issue (e.g. reusing memory in the copy phase, so that the factor of two is an upper limit,
but unlikely in practice).

GHC also uses a neat idea called the parallel garbage collector⁷ This is a garbage collector which parallelizes the
scavenging process. However, the garbage collector itself isn’t concurrent with the runtime: all Haskell threads stop
when the garbage collector runs.

Of course, concurrency comes with its own problems. What happens when two GC threads try to scavenge the same
object? They might make two copies of the same object, and the naive fix is to add a lock to every object, which is really
expensive. Purity once again comes to the rescue: if the raced object isn’t an IORef (which can be detected), then it’s
immutable, so it’s OK if two objects get made, and the slight bump in memory is all right, much like the optimization
where the same thunk is copied in some cases. This is the difference between unsafePerformIO and unsafeDupableIO,
the latter of which can be collected in this way, and therefore might get duplicated!

The lesson which keeps coming back is that purity implies flexibility, at least in garbage collection. This answers the
StackOverflow question mentioned above: a type called MutableArray has to be placed in the mutable set by default,
so allocating lots of them isn’t so good of an idea. (This doesn’t apply to MutableByteArrays, since they don’t have
pointers.) There’s work to be done at making Haskell better at mutable GC.

Scheduling. The scheduler is the heart of the runtime system; in some sense, it uses a function called StgRun to run
Haskell code, and this calls StgReturn to return to the scheduler.

A thread is represented by a TSO object, which lives on the heap. This contains a stack object pointer, which points to
a stack object (careful! This is also heap allocated). This contains information about the closure in question, the current
stack pointer, and so on. The stack is segmented: there’s the thread object, and some stack objects, and that’s it. This
means making threads is cheap: it doesn’t cost much to make threads, and since the stack is segmented, threads can
start with a small stack.

The scheduler itself has a thread queue pointing to a bunch of thread objects. It repeatedly picks a thread, runs it,
checks for heap overflows (and then collects garbage), and so on. This is considered part of the root set for garbage
collection. In multi-core systems (and with the right compiler flags), there’s a separate scheduler loop (called a Haskell
execution context, or HEC) for each core. These can be thought of as locks, which the OS threads take out. For example,
if two OS threads are running Haskell code and a third is idle, so if one of the other threads goes and does something

⁷“Java has something similar, but I understand how GHC’s works.”
else (like an FFI call), then the third thread jumps in and begins scheduling. Then, the garbage collector takes all of the locks, preventing code from running which garbage is collected.

How does the scheduler balance work amongst the threads? When a thread is out of work, it releases a lock on its run queue; then, a thread which has work may acquire the lock, split its work between them, and then release the lock, giving the other thread more work to do. This is a throughput model, which eliminates the need to fancy inter-thread communication.

Bound threads also fit into this scheme: they are required to run on a specific core; if a bound thread is acquired by the wrong OS thread, then that thread pings the correct one and tells it to acquire the lock, runs it, and then passes the scheduler back to the initial thread (so that it can handle other bound threads, should the need arise). Ideally, one would have all of the bound threads living in one core or one HEC, but in cases where there aren’t very many bound threads, it would be inefficient to lock one core up with only a few pieces of work.

MVars are very important for the Haskell user, so it’s worth peering behind the curtain for these too. An MVar contains a queue of threads blocked on it, so when a thread blocks on an MVar the scheduler moves it onto the MVar queue. This is hard with proper OS threads, so it’s really nice to link threads to the data structure they run on (which is also true for other concurrency primitives in GHC). Interestingly, if an MVar falls out of scope, all of the threads blocking on it are garbage-collected.

In summary, the scheduler makes everything live on the heap: stacks, thread objects, concurrency primitives, etc. This is nice. One fun aside is that Rust and Go, hip new programming languages, are discovering that contiguous stacks are a bit messier, are looking to similar systems. GHC’s approach is not without problems, e.g. a “hot swapping” problem where stacks are continually allocated and deallocated in certain kinds of code. This leads to cheap threads, which are nice, and purity means an awful lot of code is already threadsafe by default. Thus, it’s not necessary to be as careful as in Java, because shared memory gives no insulation against races and such. But pure code can’t leak across threads.

These are two of the many aspects of the runtime system: there’s more to each, of course. There’s lots and lots of code and many papers involved in this.

18. Library-Level Optimization: 5/29/14

“Sometimes [the inliner] hits the pub and doesn’t get up in the morning, and then your code isn’t inlined.”

One good strategy is inlining: in GHC-land, inlining is very important for performance. Function application is cheap, but not applying a function at all is cheaper.

```haskell
all :: (a -> Bool) -> [a] -> Bool
all p = and . map p

and :: [Bool] -> Bool
and = foldr && True
-- in some sense, foldr replaces every cons cell with the function one specifies.
```

The above definition, from the Prelude, isn’t quite optimal: it generates a list by mapping p, and then applying and. This is fine with respect to composability, but allocating the intermediate list isn’t strictly necessary. Of course, lazy evaluation means that we’ll just get a thunk, and the extra cells are generated, but not up front. Here’s a more efficient implementation:

```haskell
all' p = go
  where go (x:xs)
        | p x    = go xs
        | otherwise = False
        | _       = True
```

This eliminates the call to map, turning 2 traversals of a list into 1 traversal. This does actually make a difference in performance.

If the inliner can see the body of all’ and p, then this would be even more efficient. This doesn’t always work; the inliner can be somewhat finicky.

There’s a terrible name for a good thing: deforestation is the business of getting rid of intermediate or unnecessary data structures. There’s some cognitive dissonance here: out of context, this sounds bad. But deforestation makes our code more efficient.

GHC has used something called foldr-build-fusion to deforest many kinds of function compositions. It relies on the following somewhat strange definition.

```haskell
build :: (forall b. (a -> b -> b) -> b -> b) -> [a]
build g = g (:) []
```
This is called a list producer; it's not meaningful for code, but it acts as a hint to the inliner.

A special pragma {{-# INLINE [1] build #-} tells GHC to inline things. In C-like languages, it's somewhat coarse: there's an inline keyword and an always-inline pragma, but in Haskell, the simplifier (which is a sequence of Core-to-Core transformations which desugar the syntax, but preserve semantics). Each pass of the simplifier has a phase number, decreasing to zero; using the flag -dverbose-core2core, one can see the sequence of phase numbers. Then, the syntax INLINE[k] indicates to not inline a function until phase k, after which it really really should be inlined. This is a full AST transformation scheme: when someone else uses your library, the same inline hints apply, and can greatly improve the performance thereof.

For example, foldr is told to inline at phase zero. This relates to the fact that the inliner really can do a lot more: we can specify transformation rules, as follows. This is pretty magical.

```haskell
{-# RULES "map/map"forall f g xs. map f (map g xs) = map (f.g) xs #-}
```

The name "map/map" is unnecessary, but is spit out by compiler debug dumps, so it's nice to have around. This rule says that maps agree with function composition, and thus can be used to eliminate an intermediate list. However, this works only for this one isolated case; there are lots of others (e.g. map after or before filter). Rather than writing n² rules, here's the actual rule used for GHC.

```haskell
{-# RULES "map"[/1]forall f xs. map f xs = build (\c n -> fold (mapFB c f) n xs) #-}
```

This ~ notation is new; it means to want to inline something until phase k, and then not.

This leads to a simple equivalence called η-equivalence: \x -> f x is completely equivalent to f. Rewriting from the left to the right is called η-contraction, and the reverse process is called η-expansion. Now we can talk about what mapFB does: it's just η-expansion.

```haskell
mapFB :: ( elt -> lst -> list )
     -> (a -> elt )
     -> a -> lst -> lst
mapFB c f = \x ys -> c (f x) ys
{-# INLINE [0] mapFB #-}
```

Now, we can fuse repeated mapFB instances together with a rewrite rule, and then a reverse rule:

```haskell
{-# RULES "maplist" [1]
forall f.
foldr (mapFB (:) f) [] = map f
#-}
```

This seems all a bit Rube-Goldbergesque, but it does all make sense. This rule undoes some of the others. But there's one more yet that collapses a foldr and a build. These make life really confusing, but also all happens to work out right.

This is all very helpful, but it's the very first instance of deforestation in Haskell, twenty years old. It's pretty fragile, and doesn't work on foldl-style loops (it makes everything else worse). However, strict left folds are pretty common, e.g. length, sum, mean, and so on.

Yet there is still some progress to be made. Lists are inductively defined, given an empty list [] and the inductive case (:+). But it's possible to use coinduction to consume data, as in the Stream class.

```haskell
{-# LANGUAGE Rank 2 Types #-} -- used existentially.

data Stream a =
forall s. Stream
  (s -> Step s a) -- observer function
  !s -- current state

data Step s a = Done
  | Skip !s -- skip over stuff, which is useful for stuff like filter
  | Yield !a !s
```

That is, induction is used to produce stuff, and coinduction is used to consume stuff. This Stream class maintains a function for retrieving the stream data and a state (hidden from the user, requiring the rank-2 type).

---

8If you've taken enough abstract algebra classes, you're used to ideas and coideas, authors and coauthors, pants and copants, and so on.
One can convert between lists and streams.

```haskell
code
streamList :: [a] -> Stream a
streamList s = Stream next s
    where next [] = Done
          next (x:xs) = Yield x xs
{-# INLINE [0] streamList #-}

unstreamList :: Stream a -> [a]
unstreamList (Stream next s0) = unfold s0
    where unfold !s = case next s of
        Done -> []
        Skip s' -> unfold s'
        Yield x s' -> (x: unfold s ')
{-# INLINE [0] unstreamList #-}
```

Now, it’s easy to write left and right folds, albeit somewhat mechanically from the constructors. All of these functions are inlined at phase 0.

This stream representation is used in several well-used Haskell packages, particularly in packed data types such as vector and text. ByteString, however, doesn’t; it predates string/build fusion.

But these interact nicely with stream fusion, as in the following.

```haskell
code
{-# RULE "STREAM stream/unstrwam fusion"
   forall s.
   stream (unstream s) = s
#-}
```

Well, that wasn’t so interesting. But let’s throw in a map:

```haskell
code
import qualified Data.Text.Fusion as S
map :: (Char -> Char) -> Text -> Text
map f t = unstream (S.map f (stream t))
-- etc.
```

Then, it’s possible to eliminate an intermeiate map in composition.

One significant issue is unrolling recursive functions. GHC doesn’t mess with recursive functions (which is a good idea, since it would cause an infinite loop), so these stream transformations have to be nonrecursive. But when this happens, then many streams can be reworked into a pipeline where all of the needed transformations are placed into a stream, avoiding intermediate allocations. This is useful for library writers, but not users (except that they should use pipelining styles). Stream fusion, like foldr optimization, is still a bit finicky.

Stream fusion can be unpleasant to code.

```haskell
code
data I s = I1 !s
        | I2 !s ({-# UNPACK #-}) !Char
        | I3 !s
intersperse :: Char -> Stream Char -> Stream Char
intersperse c (Stream next0 s0) = Stream next (I1 c0)
    where
        -- ten lines of boilerplate...
```

However, sometimes it’s even necessary to write rather imperative code, with hand-rolled loops, to reduce heap allocation. Adding rules is a great way to make sure the fusion-based and the array-based versions of a function are paired with similar pieces of code to get better performance. The library writer has to write about twice as much code, but the result is considerably faster code.

The vector library has a more sophisticated version, because it came along later. To wit, there are monadic streams, so that one can do stream fusion over mutable arrays.

There’s actually still lots of recent research on streaming and related optimizations. This can be discovered on Reddit’s Haskell subforum (of all places; more useful than one might think!), e.g. http://www.reddit.com/r/haskell/comments/26j20h/nomeata_does_list_fusion_work/


“Swift brings mobile computing from the stone age to the mid-1990s, so instead of grunting and hitting each other with the sticks of Objective-C, we can now listen to Nirvana when we code.”
Swift is a new programming language released yesterday (!) by Apple for mobile computing. Chris Lattner (behind LLVM) and other luminaries are behind it, and the language seems promising. Yesterday at Facebook, for example, there was a lot of discussion about this new mobile language, since Facebook is probably the most complicated iOS app in existence. Apple has a cap of 60 MB for apps, which sets reasonable limits on the iOS’s runtime system, but Facebook has been skirting this for about a year, and has only been able to work with it due to increasingly clever fixes.

This language is relatively strongly typed, and this allows for static analysis — which is quite useful in the real world. Objective-C is very dynamic, but this dynamicity has a huge cost: every method adds a few dozen bits in descriptors and such, which is actually quite high compared to that in C++. The lack of safety in Objective-C also means that diffs to the Facebook iOS app sometimes crash on startup (though these are detected before being released); Swift has by default no null pointer, and a very hidden Maybe-like type: one can write var i: Int, but var i: Int? to indicate that it could be nullable.

Swift also doesn’t implicitly coerce anything, so NULL doesn’t evaluate to false, and nor does 0. But one can test if j {...} for nullity, within which j is unwrapped. This isn’t as awesome as Haskell, but is pretty good. Using generics, one can actually implement this (e.g. Maybe<T>, with Just<T> or Nothing) and do pattern-matching.

However, long experience with programming languages breeds cynicism. Some of the shinier aspects of Haskell are lost in translation to the real world, e.g. pattern matching can only be done one level deep, and types cannot be recursively defined (as with trees in Haskell). This is probably because information propagates in interesting ways (in this case, probably across Twitter). But there are closures, and there’s garbage collection (though it’s reference counting, which is a bit of a mixed blessing). People disagree, often zealously, as to which of these methods are best, but these aren’t the kinds of folks you want at your dinner parties.

Swift makes a distinction between reference types (called classes), passed by pointer, and value types, called structs, which are passed by value. Unfortunately, there’s no good way to convert one to the other, and many core types (e.g. the String type) are value types, which leads to extra memory calls. However, a lot of this is behind a barrier accessible to the language implementer but not the programmer, which is bad mojo and leads to anger. Go is particularly bad at this.

Swift also claims that its arrays are immutable, which they define strangely: the size of an array can’t be modified, but its contents can be changed! This means that the bounds are statically known, which is fine, but immutable is the wrong word to use there.

It’s likely that these will be fixed in upcoming years, as the language gets thought about more, and it certainly does a lot of things better than Objective-C. However, it doesn’t improve on its module system, nor its slow compilation (Go really is the only language that gets this really well; GHC, for all of its wonderful aspects, is slower than almost anything except Scala).

Swift is built on LLVM, so it can interoperate with Objective-C. Methods designed to work with both are tagged with an annotation, so that the compiler generates code for both versions.

The Jackknife. The Jackknife is a method of statistical analysis. Suppose one has a sample \([0, 0, 1, 0]\) from some population about which we don’t really know much. The mean of the sample is, unsurprisingly, 0.25, but this isn’t a very interesting property. One might instead ask how much it’s affected by the data within it, e.g. by computing its variance.

A package called Statistics.Sample is good for this, along with Data.Vector.Unboxed (often imported as U). The language extension OverloadedLists allows one to treat lists much like vectors. It turns out versioning issues put the code into Cabal hell, so let’s open up a Python script and calculate the variance, which is 0.25.

This is where the Jackknife enters into the play: it’s an advanced tool, but it’s pretty easy to understand. First, do a leave-one-out subsample process, where each entry is deleted once: \([0, 0, 1], [0, 0, 0]\), and so on. Three of these have variance 1/3, and the fourth has variance 0, so the mean of these four variances is \((3/4)(1/3) = 1/4\). What would it look like if the overall variance and that of the jackknifed sample were different?

\[^9\]Objective-C is really a shit programming language from any way you look at it. it takes the worst aspects of C and the worst aspects of Smalltalk, and looks like a turd that was shit out of several animals at once.\]
Well, laziness is king, so let’s write a QuickCheck test to discover a good counterexample for us.

```haskell
import Statistics.Resampling
import Statistics.Types
import qualified Data.Vector.Unboxed as U
import Test.QuickCheck

t_est est x y z xs = estimate estv ===
  -- this operator means “print the value if they’re not equal
  -- and otherwise is ==
  estimate Mean (jackknife est v)
  here v = Ufromlist . map (fromIntegral) (x:y:z:xs)
```

This quickly fails with a useful counterexample. The point is, QuickCheck isn’t just about testing code; it can do a bunch of checking things.

The jackknife is an \(O(n^2)\) algorithm, which is bad for larger samples, and was a bottleneck in the criterion benchmarking library for a while. But a function called `scanl` uses a technique called prefix sums, which was a faster way to speed up this sort of statistical algorithm. In other words, thinking about functional programming is useful for solving interesting problems too!