Haskell
a lazy, purely functional language

COS 326
David Walker
Princeton University

slides copyright 2013-2015 David Walker and Andrew W. Appel
permission granted to reuse these slides for non-commercial educational purposes
Another cool, typed, functional programming language

• Like OCaml in that:
  – it is a functional language with parametric polymorphism

• Unlike OCaml in that it is:
  – **pure**: functions with type a -> b have no effect!
  – **lazy**: f e does not evaluate e right away; passes e to f
  – has a **weak module system**; uses **type classes**
  – has a very sophisticated type system with **higher kinds**
  – has several cool extensions for concurrent and parallel programming including **transactional memory**
Glasgow in the 1990s

ghc stands for “Glasgow Haskell Compiler”

Phil Wadler
(now at U. Edinburgh)

Simon Peyton Jones
(now at Microsoft Research, Cambridge, England)

Creators of Haskell language

Edinburgh in the 1970s

Robin Milner, Luca Cardelli, Luis Damas, Mads Tofte . . . created ML language

Why are we getting all our functional programming languages from Scotland?
HASKELL BASICS
Haskell Definitions

• Mostly like OCaml, with a few small syntactic differences
  – parameters are immutable by default
  – let declarations introduce new functions and value

\[
\text{foo } z = \begin{align*}
\text{let triple } x &= x \times 3 \\
\text{in triple } z
\end{align*}
\]

– equivalently, we might use a "where" definition:

\[
\text{foo } z = \text{triple } z \\
\text{where} \\
\text{triple } x = x \times 3
\]
Haskell Indentation

- Haskell, like Python, but unlike Java, OCaml or math written in a notebook, has semantically meaningful indentation

- **Wrong:**

  ```haskell
  copies k n =
  case n of
    0 -> []
    _ -> k : copies k (n-1)
  ```

  ```haskell
  zap z =
  let x = z
  y = z + z
  in x + y
  ```

  ```haskell
  indent y = ...
  ```

  ```haskell
  indent z + z
  ```

  ```haskell
  zap z =
  let x = z
  y =
  z + z
  in
  x + y
  ```
Haskell Indentation

- Haskell, like Python, but unlike Java, OCaml or math written in a notebook, has semantically meaningful indentation

- **Right:**

beginning of x defines indentation level

```haskell
copies k n = 
  if n == 0 then 
    [] 
  else 
    k : copies k (n-1)

zap z = 
  let 
    x = z 
    y = z + z 
  in 
    x + y

zap z = 
  let 
    x = z 
    y = z + z 
  in 
    x + y
```
Haskell Types

• We have the option of declaring the type of a definition prior to the definition itself

zap :: Int -> Int
zap z =
  let x = z
  y = z + z
  in
  x + y

just to be annoying, Haskell uses "::" for "has type" and uses ":" for "cons" – the opposite of ML
Tuples

• Haskell uses tuples like ML
  – constructed by enclosing a sequence of values in parens:

    ('b', 4) :: (Char, Integer)

  – deconstructed (used) via pattern matching:

    easytoo :: (Integer, Integer, Integer) -> Integer
    easytoo (x, y, z) = x + y * z
• Lists are very similar to ML lists but the type is written [t] instead of "t list"

\[ [1, 2, 3] :: [\text{Integer}] \]

\[ ['a', 'b', 'c'] :: [\text{Char}] \]

• [ ] is the empty list (called nil)
• 1:2:[] is a list with 2 elements
• String is a synonym for [Char]
• We can build lists of lists:

\[ [ [1, 2], [3], [8, 9, 10] ] :: [ [ \text{Integer} ] ] \]
--- Sum the elements of a list

listSum :: [ Integer ] -> Integer

listSum [] = 0
listSum (x:xs) = x + listSum xs
Functions deconstructing lists

-- Sum the elements of a list

listSum :: [ Integer ] -> Integer

\[
\begin{align*}
\text{listSum } [ ] &= 0 \\
\text{listSum } (x:xs) &= x + \text{listSum } xs
\end{align*}
\]

length :: [a] -> Int

\[
\begin{align*}
\text{length } [ ] &= 0 \\
\text{length } (x:xs) &= 1 + \text{length } xs
\end{align*}
\]

lower case letter = any type at all (a type variable)

upper case letter = a concrete type
Functions deconstructing lists

```
-- Sum the elements of a list

listSum :: [ Integer ] -> Integer

listSum [ ] = 0
listSum (x:xs) = x + listSum xs
```

```
length :: [a] -> Int

length [ ] = 0
length (x:xs) = 1 + length xs
```

```
cat :: [a] -> [a] -> [a]

cat [ ] xs2 = xs2
cat (x:xs) xs2 = x:(cat xs xs2)
```
Functions deconstructing lists

-- Sum the elements of a list

listSum :: [ Integer ] -> Integer

listSum [ ] = 0
listSum (x:xs) = x + listSum xs

length :: [a] -> Int

length [ ] = 0
length (x:xs) = 1 + length xs

cat :: [a] -> [a] -> [a]

cat [ ] xs2 = xs2

(++) :: [a] -> [a] -> [a]

(++) [ ] xs2 = xs2
(++) (x:xs) xs2 = x:(xs ++ xs2)
PURITY &
SUBSTITUTION OF EQUALS FOR EQUALS
• A key law about Haskell programs:

\[
\begin{align*}
\text{let } x &= \langle \text{exp} \rangle \text{ in} \\
\ldots x \ldots x \ldots &= \ldots \langle \text{exp} \rangle \ldots \langle \text{exp} \rangle \ldots
\end{align*}
\]

• For example:

\[
\begin{align*}
\text{let } x &= 4 \ `\text{div}` 2 \text{ in} \\
\text{ } x + 5 + x &= (4 \ `\text{div}` 2) + 5 + (4 \ `\text{div}` 2) \\
\text{ } &= 9
\end{align*}
\]

*Note: not necessarily the same run time; (4 `div` 2) will be evaluated twice instead of once.
• We'd also like to use **functional abstraction** without penalty

\[
\text{halve :: Int -> Int} \quad \text{halve } n = n \ `\text{div}` 2
\]

• And instead of telling clients about all implementation details, simply expose key laws:

Lemma 1: for all \(n\), if even \(n\) then \((\text{halve } n + \text{halve } n) = n\)

• Now we can reason locally within the client:

\[
\begin{align*}
\text{let } x = \text{halve } 4 \text{ in } x + y + x &= (\text{halve } 4) + y + (\text{halve } 4) & \text{(substitution)} \\
&= (\text{halve } 4) + (\text{halve } 4) + y & \text{(arithmetic)} \\
&= 4 + y & \text{(Lemma 1)}
\end{align*}
\]
• What happens when we add mutable data structures?
• Consider this *OCaml* program:

```ocaml
let x = ref 0

let foo (y:int) : int =
  x := !x + 1;
  arg + !x;

foo : int -> int
```

• We lose a lot of reasoning power!

```ocaml
let y = foo 3 in
y + y
≠
foo 3 + foo 3
```
Computational Effects

- What happens when we add mutable data structures?
- Consider this OCaml program:

```ocaml
let x = ref 0

let foo (y:int) : int =
x := !x + 1;
arg + !x;

let y = foo 3 in
y + y ≠ foo 3 + foo 3
```

- We lose a lot of reasoning power!

```
let y = foo 3 in
y + y
```

```
8
```

```
foo 3 + foo 3
```

```
9
```
Computational Effects

• What happens when we add mutable data structures?
• Consider this *OCaml* program:

```ocaml
let foo (y:int) : int =
    print_int y;
    arg + !x;

let y = foo 3 in
y + y ≠
foo 3 + foo 3

6 printing "3"

6 printing "33"
```

• We lose a lot of reasoning power!
Computational Effects

- A function **has an effect** if its behavior cannot be specified exclusively as a relation between its input and its output
  - I/O is an effect
  - An imperative update of a data structure is an effect
- When functions can no longer be described exclusively in terms of the relationship between arguments and results
  - many, many fewer equational laws hold
  - In general, in OCaml,

\[
\text{let } x = \langle \text{exp} \rangle \text{ in } ... x ... x ... \neq ... \langle \text{exp} \rangle ... \langle \text{exp} \rangle ...
\]

- In general, in Haskell,

\[
\text{let } x = \langle \text{exp} \rangle \text{ in } ... x ... x ... = ... \langle \text{exp} \rangle ... \langle \text{exp} \rangle ...
\]
• A function has an effect if its behavior cannot be specified exclusively as a relation between its input and its output. I/O is an effect.
  - An imperative update to a data structure is an effect.
• When functions can no longer be described exclusively in terms of the relationship between arguments and results,
  - In general, in OCaml,
  - In general, in Haskell,

This is kind of magical! Haskell has substitution of equals for equals AND ALSO HAS EFFECTS. But how?

let x = <exp> in .... x ... = .... <exp> ... <exp> ...
DIGRESSION:
MONADS AND
LIST COMPREHENSIONS
Example: Find all \((a,b,c)\) such that
\[
1 \leq a \leq b \leq c \leq 25 \quad \text{and} \quad a^2 + b^2 = c^2
\]

\[
p(a,b,c) = a^2 + b^2 = c^2
\]

triples = do
  a <- [1..25]
  b <- [a..25]
  c <- [b..25]
  guard (p a b c)
  return (a,b,c)

Source: Greg Bacon,
List comprehensions

Example: Find all \((a,b,c)\) such that

\[
1 \leq a \leq b \leq c \leq 25 \quad \text{and} \quad a^2 + b^2 = c^2
\]

\[
p \ a \ b \ c = a^2 + b^2 = c^2
\]

\[
\text{triples} = \text{do}
\]
\[
a \leftarrow [1..25]
\]
\[
b \leftarrow [a..25]
\]
\[
c \leftarrow [b..25]
\]
\[
\text{guard} (p \ a \ b \ c)
\]
\[
\text{return} (a,b,c)
\]

It looks like some sort of “iterator” feature; but Haskell doesn’t have built-in “iterators.” It's got something more general!

Iterators arise as a natural consequence of:

- pure functional programming
- + lazy evaluation
- + lists
- + a nice notation for *monadic computations*

Now, let’s “unpack” this to see how that works.

List comprehensions

Example: Find all \((a,b,c)\) such that
\[
1 \leq a \leq b \leq c \leq 25 \quad \text{and} \quad a^2+b^2=c^2
\]

triples = do {
  a <- [1..25] ;
  b <- [a..25] ;
  c <- [b..25] ;
  guard (p a b c) ;
  return (a,b,c)
}

Source: Greg Bacon,
List comprehensions

Example: Find all \((a,b,c)\) such that
\[
1 \leq a \leq b \leq c \leq 25 \quad \text{and} \quad a^2 + b^2 = c^2
\]

\[
\text{triples} = \text{do} \{
\quad a \leftarrow \[1..25\] ; \\
\quad b \leftarrow \[a..25\] ; \\
\quad c \leftarrow \[b..25\] ; \\
\quad \text{guard} \ (p \ a \ b \ c) ; \\
\quad \text{return} \ (a,b,c)
\}
\]

This is just the “range” operator for lists; \([1..5]\) is just \([1,2,3,4,5]\). Easy to construct this with “let rec” in ML, or as a recursive function in Haskell. No magic here.
“a <- e ; ...” is just a notational abbreviation for the monadic **bind** operator that you’ve seen before. Haskell makes it easy to introduce notational abbreviations.

```haskell
triples = do {
    a <- [1..25] ;
    b <- [a..25] ;
    c <- [b..25] ;
    guard (p a b c) ;
    return (a,b,c)
}
```

That’s λ from lambda-calculus; in ML you’d write “fun”.

```
triples = [1..25] >>= a -> [a..25] >>= b -> [b..25] >>= c ->
          guard (p a b c) >>
          return (a,b,c)
```
“a <- e ; ...” is just a notational abbreviation for the monadic **bind** operator that you’ve seen before. Haskell makes it easy to introduce notational abbreviations.

```haskell
triples = do {
    a <- [1..25] ;
    b <- [a..25] ;
    c <- [b..25] ;
    guard (p a b c) ;
    return (a,b,c)
}
```

That’s $\lambda$ from lambda-calculus; in ML you’d write “fun”

```haskell
do {a <- e1; e2} == e1 >>= (\a -> e2) == e1 >>= (fun a -> e2)
```
List Monad

You’ve seen monads before, but let me remind you:

(* in Ocaml notation, not Haskell! *)

module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

pronounced “bind”

One way to think of a monad 'a M:
It is a special sort of "container" for 'a.

return x : puts x into the container
c >>= f : extracts items from c;
then pushes them through f, which generates new container

Bind (c >>= f) allows you to compute with the items inside the container
You’ve seen monads before, but let me remind you:

(* in Ocaml notation, not Haskell! *)

module type MONAD = sig

type 'a M

val return : 'a -> 'a M

val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

pronounced “bind”

One way to think of a monad 'a M:
It is a special sort of "container" for 'a.

return x : puts x into the container

c >>= f : extracts items from c;
then pushes them through f,
which generates new container

Bind (c >>= f) allows you to compute with
the items inside the container

When we did error processing, the
container was an option type:
'a option. At most one thing was in
the container at a time.
Here, the monad we want is the List monad with concatMap

(* in Ocaml notation, not Haskell! *)

module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

One more thing:  >>= is just a “bind” that throws away its argument.

let (>>) (m: 'a M) (f: 'b M) : 'b M = m >>= fun _ -> f
List Monad

Here, the monad we want is the List monad with concatMap

(* in Ocaml notation, not Haskell! *)

module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

(* in Ocaml notation, not Haskell! *)

module ListMonad : MONAD

  type 'a M = 'a list

  let return (x: 'a) = [x]

  let (>>=) (as: 'a list) (f: 'a -> 'b list) : 'b list = reduce append nil (map f as)

end
List Monad

Here, the monad we want is the List monad with `concatMap`.

```ocaml
(* in Ocaml notation, not Haskell! *)
module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end
```

```ocaml
module ListMonad : MONAD

  type 'a M = 'a list

  let return (x: 'a) = [x]

  let (>>=) (as: 'a list) (f: 'a -> 'b list) : 'b list =
    reduce append nil (map f as)

end
```

```
let double a = [a;a]

[1;2;3] >>= double  ==  reduce append nil [ [1;1]; [2;2]; [3;3] ]  ==  [1;1;2;2;3;3]
```
Example

\[
[1,2,3] \gg= \ a \mapsto \ return \ (a,a)\\
\]

\[
\begin{align*}
\text{let } f\ a &= \ return \ (a,a) \text{ in} \\
\text{reduce append nil (map } f\ [1,2,3]) \\
\end{align*}
\]

\[
\begin{align*}
return\ x &= [x] \\
(\gg=)\ xs\ f &= \text{reduce append nil (map } f\ xs)
\end{align*}
\]
Example

\[
\text{[1,2,3]} \quad >>= \quad \lambda a \to \quad \text{return} \quad (a,a)
\]

\[
= \quad \text{let } f \ a = \text{return} \ (a,a) \ \text{in} \quad \text{reduce} \ \text{append} \ \text{nil} \ (\text{map} \ f \ [1,2,3])
\]

\[
= \quad \text{let } f \ a = \text{return} \ (a,a) \ \text{in} \quad \text{reduce} \ \text{append} \ \text{nil} \ [f \text{ 1, f 2, f 3}]
\]

\[
\text{return} \ x = [x]
\]

\[
(\text{>>=}) \ \text{xs} \ f = \text{reduce} \ \text{append} \ \text{nil} \ (\text{map} \ f \ \text{xs})
\]
Example

\[ [1,2,3] \gg\gg \ a \rightarrow \ \text{return (a,a)} \]

\[
\begin{align*}
\text{return } x &= [x] \\
(\gg\gg\gg) \text{ xs f} &= \text{reduce append nil (map f xs)}
\end{align*}
\]
Example

\[1,2,3 \gg= \ \lambda a -> \ \text{return} \ (a,a)\]

\[=\]

let \( f \ a = \ \text{return} \ (a,a) \) in \nreduce append nil \ (map \ f \ [1,2,3])

\[=\]

let \( f \ a = \ \text{return} \ (a,a) \) in \nreduce append nil \ [f 1, f 2, f 3]

\[=\]

reduce append nil \ [return (1,1),
return (2,2),
return (3,3) ]

\[=\]

reduce append nil \ [(1,1),
(2,2),
(3,3)]

\[\]

return \( x = [x] \)

\[\gg=\] \( xs \ f = \ \text{reduce append nil} \ (\text{map} \ f \ xs) \)
Example

\[ [1,2,3] \triangleright= \ \backslash a \rightarrow \,
\ return \ (a,a) \]

\[ = \]

\[ \text{let } f \ a = \ return \ (a,a) \ 
\text{in} \ reduce \ append \ nil \ (\text{map} \ f \ [1,2,3]) \]

\[ = \]

\[ \text{let } f \ a = \ return \ (a,a) 
\text{in} \ reduce \ append \ nil \ [f \ 1, f \ 2, f \ 3] \]

\[ = \]

\[ \text{reduce append nil} \ [\text{return} \ (1,1), 
\text{return} \ (2,2), 
\text{return} \ (3,3)] \]

\[ = \]

\[ \text{reduce append nil} \ [(1,1), 
[(2,2)], 
[(3,3)] \] \]

\[ = \]

\[ \text{return} \ x = [x] \]

\[ = \]

\[ [ (1,1), (2,2), (3,3) ] \]

\[ (\triangleright=) \ xs \ f = \text{reduce append nil} \ (\text{map} \ f \ xs) \]
Example

foo =
[1..3]  >>= \a ->
[a..4]  >>= \b ->
return (a,b)

= let f a = [a..4] >>= \b -> return (a,b)
in  reduce app nil (map f [1,2,3])

= let f a = [a..4] >>= \b -> return (a,b)
in  reduce app nil [f 1, f 2, f 3]

= reduce app nil
  [[1,2,3,4] >>= \b -> return (1,b),
   [2,3,4] >>= \b -> return (2,b),
   [3,4]  >>= \b -> return (3,b)]

= return (x: ‘a) = [x]

 >>= (al: [a]) (f: a -> [b]) : [b] = reduce app nil (map f al)
Example

foo = [1..3] >>= \a -> [a..4] >>= \b -> return (a, b)
return (x: 'a) = [x]

(>>=) (al: [a]) (f: a -> [b]) : [b] = reduce app nil (map f al)

let f a = [a..4] >>= \b -> return (a, b)
in reduce app nil (map f [1,2,3])

let f a = [a..4] >>= \b -> return (a, b)
in reduce app nil [f 1, f 2, f 3]

reduce app nil [[1,2,3,4] >>= \b -> return (1, b), [2,3,4] >>= \b -> return (2, b), [3,4] >>= \b -> return (3, b)]
Example

foo = 
[1..3] >>= \a ->
[a..4] >>= \b ->
return (a,b)

return (x: 'a) = [x]

(>>=) (al: [a]) (f: a -> [b]) : [b] = reduce app nil (map f al)
Example

\[
[1..3] >>= \lambda a \rightarrow [a..4] >>= \lambda b \rightarrow return (a,b)
\]

= reduce app nil

\[
[reduce app nil [((1,1)],[(1,2)],[(1,3)],[(1,4)]],
reduce app nil [((2,2)],[(2,3)],[(2,4)]],
reduce app nil [((3,3)],[(3,4)]]
\]

= reduce app nil

\[
[((1,1),(1,2),(1,3),(1,4)],
[(2,2),(2,3),(2,4)]],
[(3,3),(3,4)]
\]

= 

\[
[(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4)]
\]

return (x: 'a) = [x]

(>>=) (al: [a]) (f: a \rightarrow [b]) : [b] = reduce app nil (map f al)
Is it really an iterator?

```
do
  a <- [1..3]
  b <- [a..4]
  return (a,b)
```

= 

```
[(1,1),(1,2),(1,3),(1,4),(2,2),(2,3),(2,4),(3,3),(3,4)]
```

It doesn’t seem like “in the spirit of an iterator” to return the whole list; it should return the pairs (1,1); (1,2); ... one at a time, on demand. That’s what we mean by an iterator.

Aha! Every expression in Haskell is lazy. Everything is on-demand. This really is an iterator!
Returning to the “Pythagorean triples” ...

do
a <- [1..25]
b <- [a..25]
c <- [b..25]
guard (p a b c) >>=
return (a,b,c)
guard (p a b c) >>= \_ ->
guard (p a b c) >>= \_ ->
guard (p a b c) >>= \_ ->

\verb|guard| :: (MonadPlus m) => Bool -> m ()
guard True = return ()
guard False = mzero

“return ()” in the List monad is \[ () \]
The “zero” for the List monad is \[ \]
```haskell
do
  a <- [1..5]
  guard (isprime a)
  return (a+1)

= [1..5] >>\ a \rightarrow guard (isprime a) >>\ _ \rightarrow return (a+1)

f a = guard (isprime a) >>\ _ \rightarrow return (a+1)
reduce app nil (map f [1,2,3,4,5])

= reduce app nil [
  guard (isprime 1) >>\ _ \rightarrow return (1+1),
  guard (isprime 2) >>\ _ \rightarrow return (2+1),
  guard (isprime 3) >>\ _ \rightarrow return (3+1),
  guard (isprime 4) >>\ _ \rightarrow return (4+1),
  guard (isprime 5) >>\ _ \rightarrow return (5+1) ]

guard True = [ () ]
guard False = [ ]
```
Example of “guard”

```
Example of “guard”

```

do
  a <- [1..5]
guard (isprime a)
return (a+1)

reduce app nil [
  guard False >> \_ return (1+1),
  guard True  >> \_ return (2+1),
  guard True  >> \_ return (3+1),
  guard False >> \_ return (4+1),
guard True  >> \_ return (5+1) ]

reduce app nil [
  fold app nil (map (\_ return (1+1)) []),
  fold app nil (map (\_ return (2+1)) [ () ]),
  fold app nil (map (\_ return (3+1)) [ () ]),
  fold app nil (map (\_ return (4+1)) []),
  fold app nil (map (\_ return (5+1)) [ () ])]

= reduce app nil [ [], [2+1], [3+1], [] [5+1] ]
= [2+1, 3+1, 5+1]
= [3,4,6]
Returning to the “Pythagorean triples” ...

```
do
  a <- [1..25]
  b <- [a..25]
  c <- [b..25]
  guard (p a b c)
return (a,b,c)
```

This is called a “list comprehension”

In monadic computation, 
_in any collection monad_ (not just List), _guard_ serves as a filter for comprehensions

There was **no magic** here (except for lazy functional programming). 
Monads, “bind”, “guard”, the List monad, are all defined as user-level functions in the Haskell language itself.
HASKELL EFFECTS
INPUT AND OUTPUT
I/O in Haskell

• Haskell has a special kind of value called an action that describes an effect on the world

• Pure actions, which just do something and have no interesting result are values of type IO ()

• Eg: putStr takes a string and yields an action describing the act of displaying this string on stdout

  -- writes string to stdout
  putStr :: String -> IO ()

  -- writes string to stdout followed by newline
  putStrLn :: String -> IO ()
• When do actions **actually happen**?

• Actions happen under two circumstances:*

  1. the action defined by `main` happens when your program is executed (ie: when ghc compiles your program and then you run it)

  2. the action defined by any expression happens when that expression is written at the ghci interpreter prompt (this is pretty similar to the last one – the expression is essential an entire program)

* there is one other circumstance: Haskell contains some special, unsafe functions that will perform I/O, most notably `System.IO.Unsafe.unsafePerformIO` so I lied when earlier when I said the equation holds in general for Haskell programs ...
I/O in Haskell

hello.hs:

```haskell
main :: IO ()
main = putStrLn "Hello world"
```

in my shell:

```bash
$ ghc hello.hs
[1 of 1] Compiling Main ( hello.hs, hello.o )
Linking hello.exe ...

$ ./hello.exe
hello world!
```
bar.hs:

```haskell
bar :: Int -> IO ()
bar n =
    putStrLn (show n ++ " is a super number")

main :: IO ()
main = bar 6
```

in my shell:

```bash
$ ghci.sh
GHCi, version 7.0.3: http://www.haskell.org/ghc/ :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
Prelude> :l bar
[1 of 1] Compiling Main  ( bar.hs, interpreted )
Ok, modules loaded: Main.
*Main> bar 17
17 is a super number
*Main> main
6 is a super number
*Main>
```
Actions

- Actions are descriptions of effects on the world. Simply writing an action does not, by itself cause anything to happen.

```haskell
bar.hs:

```hellos :: [IO ()] hellos = [putStrLn "Hi", putStrLn "Hey", putStrLn "Top of the morning to you"]

main = hellos !! 2
```

In my shell:

```haskell
Prelude> :l hellos
...
*Main> main
Top of the morning to you
*Main>
```

$l !! n$ gives $n$th element of list $l$
• Actions are just like any other value -- we can store them, pass them to functions, rearrange them, etc:

```haskell
sequence_ :: [IO ()] -> IO ()
```

```haskell
baz.hs:

```haskell
hellos :: [IO ()]
hellos = [putStrLn "Hi",
         putStrLn "Hey",
         putStrLn "Top of the morning to you"]
```

```haskell
main = sequence_ (reverse hellos)
```

```
in my shell: Prelude> :l hellos
...
*Main> main
Top of the morning to you
Hey
Hey
HI
```
Combining Actions

• The infix operator $\gg$ takes two actions $a$ and $b$ and yields an action that describes the effect of executing $a$ then executing $b$ afterward

```haskell
howdy :: IO ()
howdy = putStr "how" $\gg$ putStrLn "dy"
```

• To combine many actions, use do notation:

```haskell
bonjour :: IO ()
bonjour = do putStr "Bonjour!"
          putStrLn " "
          putStrLn "Comment ca va?"
```
Quick Aside: Back to SEQEQ*

• Do we still have it? Yes!

let a = putStrLn "hello" in
do
  a
  a
doo = putStrLn "hello"
putStrLn "hello"
putStrLn "hello"

an action made up of doing `a` and then doing `a` again where `a` is the putStrLn "hello" action

an action made up of doing the putStrLn "hello" action and then doing the putStrLn "hello" action again

* SEQEQ = substitution of equals for equals
• Some actions have an effect and yield a result:

  -- get a line of input
  getLine :: IO String

  -- get all of standard input until end-of-file encountered
  getContents :: IO String

  -- get command line argument list
  getArgs :: IO [String]

• What can we do with these kinds of actions?
  – we can extract the value and sequence the effect with another:
Input Actions

• Some actions have an effect and yield a result:

  -- get a line of input
  getLine :: IO String

  -- get all of standard input until end-of-file encountered
  getContents :: IO String

  -- get command line argument list
  getArgs :: IO [String]

• What can we do with these kinds of actions?
  – we can extract the value and sequence the effect with another:

    do
      s <- getLine
      putStrLn s
Some actions have an effect and yield a result:

-- get a line of input
getLine :: IO String

-- get all of standard input until end-of-file encountered
getContents :: IO String

-- get command line argument list
getArgs :: IO [String]

What can we do with these kinds of actions?

– we can extract the value and sequence the effect with another:

\[
\text{do } s \leftarrow \text{getLine} \quad \text{putStrLn } s
\]

s has type string
getLine has type IO string
Some actions have an effect and yield a result:

-- get a line of input
getLine :: IO String

-- get all of standard input until end-of-file encountered
getContents :: IO String

-- get command line argument list
getArgs :: IO [String]

do s <- getLine
   putStrLn s

Think of type IO String as a box containing a computation that will do some work and then produce a string. getLine is such a box. the <- gets the String out of the box.
• A whole program:

```haskell
main :: IO ()
main = do
    putStrLn "What's your name?"
    s <- getLine
    putStrLn ("Hey, ", s)
    putStrLn (" , cool name!")
```
import System.IO
import System.Environment

processArgs :: [String] -> String
processArgs [a] = a
processArgs _ = ""

echo :: String -> IO ()
echo "" = putStrLn "Bad Args!"
echo fileName = do
  s <- readFile fileName
  putStrLn "Here it is:"
  putStrLn "***********"
  putStrLn s
  putStrLn "\n***********"

main :: IO ()
main = do
  args <- getArgs
  let fileName = processArgs args
  echo fileName
SEQEQ (Again!)

- Recall: $s1 + s2$ concatenates String $s1$ with String $s2$
- A valid reasoning step:

```haskell
let s = "hello" in
do putStrLn (s ++ s) = do putStrLn ("hello" ++ "hello")
```
SEQEQ (Again!)

• Recall: $s_1 + s_2$ concatenates String $s_1$ with String $s_2$

• A valid reasoning step:

\[
\begin{align*}
\text{let } s &= "hello" \text{ in} \\
&\quad \text{do} \\
&\quad \text{putStrLn } (s + s) \\
\end{align*}
\]

\[
\begin{align*}
&= \text{do} \\
&\quad \text{putStrLn } ("hello" + "hello") \\
\end{align*}
\]

• A valid reasoning step:

\[
\begin{align*}
&\text{do} \\
&\quad \text{let } s = "hello" \\
&\quad \text{putStrLn } (s + s) \\
\end{align*}
\]

\[
\begin{align*}
&= \text{do} \\
&\quad \text{putStrLn } ("hello" + "hello") \\
\end{align*}
\]
SEQEQ (Again!)

- Recall: \( s_1 ++ s_2 \) concatenates \textbf{String} \( s_1 \) with \textbf{String} \( s_2 \)
- A valid reasoning step:

\[
\text{let } s = "hello" \text{ in do putStrLn (s ++ s) } = \text{ do putStrLn ("hello" ++ "hello")}
\]

- A valid reasoning step:

\[
\text{do let } s = "hello" \text{ in putStrLn (s ++ s) } = \text{ do putStrLn ("hello" ++ "hello")}
\]

- Wait, what about this:

\[
\text{do s <- getLine putStrLn (s ++ s) } \neq \text{ do putStrLn (getLine ++ getLine)}
\]

\textbf{wrong type: } \( \text{getLine :: IO String} \)
SEQEQ (Again!)

- Invalid reasoning step?

```
let s = getLine in
  do
    putStrLn (s ++ s)
  ?
  do
    putStrLn (getLine ++ getLine)
```
• Invalid reasoning step?

```
let s = getLine in
  do
    putStrLn (s ++ s)
```

wrong type:
s :: IO String

```
=>

do
  putStrLn (getLine ++ getLine)
```

wrong type:
getLine :: IO String
• Invalid reasoning step?

```
let s = getLine in
  do
    putStrLn (s ++ s) =?= do
    putStrLn (getLine ++ getLine)
```

Wrong type:
s :: IO String
Wrong type:
getLine :: IO String

• The Haskell type system shows \( x \leftarrow e \) is different from \( \text{let } x = e \)
  – \( x \) has a different type in each case
  – \( \text{let } x = e \) enables substitution of \( e \) for \( x \) in what follows
  – \( x \leftarrow e \) does not enable substitution -- attempting substitution leaves you with code that won't even type check because \( x \) and \( e \) have different types (type \( T \) vs. type \( \text{IO } T \))
The List monad was not “magic.” List comprehensions, “iterators”, guards, could all be expressed in the pure functional language with no magic extensions.

The IO monad *is* magic. But all the monadic combinators for manipulating it, sequencing, etc., are not magic; they are expressible in the pure functional language.
HASKELL EFFECTS:
REFERENCES
Coming Back to References Again

• Remember this *OCaml* function:

```ocaml
let x = ref 0
let foo (y:int) : int =
  x := !x + 1;
  arg + !x;

foo : int -> int
```

• We noticed that:

```ocaml
let y = foo 3 in
y + y
```

≠

```ocaml
foo 3 + foo 3 : int
```

• What if we write something similar in Haskell?
Coming Back to References Again

• Remember this **OCaml** function:

```
let x = ref 0
let foo (y:int) : int =
  x := !x + 1;
  arg + !x;

foo : int -> int
```

• We noticed that:

```
let y = foo 3 in
y + y

≠

foo 3 + foo 3 : int
```

• What if we write something similar in Haskell?

```
x :: Ref int
foo :: int -> int
foo y =
  x := read x + 1;
  arg + !x;

foo y

doesn't type check
```
Haskell Types Tell You Where the Effects Aren't!

Haskell function types are pure -- totally effect-free

foo :: int -> int

Haskell’s type system forces* purity on functions with type a -> b
- no printing
- no mutable data
- no reading from files
- no concurrency
- no benign effects (like memoization)

* except for unsafePerformIO

exp :: int

Same with expressions. Expressions with just type int have no effect!
Haskell Types Tell You Where the Effects Aren't!

foo :: int -> int

totally pure function

<code> :: IO int

suspended (lazy) computation that performs effects when executed
Haskell Types Tell You Where the Effects Aren't!

foo :: int -> int

<code> :: IO int

bar :: int -> IO int

totally pure function

suspended (lazy) computation that performs effects when executed

totally pure function that returns suspended effectful action
Haskell Types Tell You Where the Effects Aren't!

```haskell
print :: string -> IO ()

reverse :: string -> string

reverse "hello" :: string

print (reverse "hello") :: IO ()
```

the type system always tells you when an effect has happened – effects can’t “escape” the I/O monad
read :: Ref a -> IO a

(+) :: int -> int -> int

r :: Ref int

(read r) + 3 :: int

Doesn’t type check
References

read :: Ref a -> IO a

(+) :: int -> int -> int

r :: Ref int

do
  x <- read r
  return (x + 3)

the "return" action has no effect; it just returns its value
creates a composition action that reads a reference and produces the value in that reference plus 3
Mutable State

Haskell uses **new**, **read**, and **write*** functions within the IO Monad to manage mutable state.

```haskell
main = do  r <- new 0  -- let r = ref 0 in
       inc r  -- r := !r+1;
       s <- read r  -- s := !r;
       print s

inc :: Ref Int -> IO ()
inc r = do
  v <- read r  -- v := !r
  write r (v+1)  -- r := !v +1
```

* actually newRef, readRef, writeRef, ...
MONADS
The Bigger Picture

- Haskell's type system (at least the part of it that we have seen so far) is very similar to the ML type system
  - eg: typing rules for pure primitives (functions, pairs, lists, etc) are similar in both cases:
    - if \( f : t_1 \rightarrow t_2 \) and \( e : t_1 \) then \( (f \ e) : t_2 \)
    - if \( e_1 : t_1 \) and \( e_2 : t_2 \) then \( (e_1, e_2) : (t_1, t_2) \)

- What Haskell has done that is special is:
  - it has been very careful to give effectful primitives special types involving "IO t"
  - allows composition of actions with type "IO t"
    - the IO data type + its composition functions are called a monad
  - it has syntax for programming with monads (do notation)
  - ML could do those things too (ie: these decisions do not depend upon a language have lazy evaluation)
We can talk about what monads *are* by referring to their interface

Recall an interface declares some new abstract types and some operations over values with those abstract types. For example:

```ocaml
module type CONTAINER = sig

    type 'a t (* the type of the container *)

    val empty : 'a t

    val insert : 'a -> 'a t -> 'a t

    val remove : 'a t -> 'a option * 'a t

    val fold : ('a -> 'b -> 'b) -> 'b -> 'a t -> 'b

end
```

There are lots of different implementations of such containers: queues, stacks, sets, randomized sets, ...
We can talk about what monads are by referring to their interface

Recall an interface declares some new abstract types and some operations over values with those abstract types. For example:

```ocaml
module type CONTAINER = sig
  type 'a t (* the type of the container *)
  val empty : 'a t
  val insert : 'a -> 'a t -> 'a t
  val remove : 'a t -> 'a option * 'a t
  val fold : ('a -> 'b -> 'b) -> 'b -> 'a t -> 'b
end
```

There are lots of different implementations of such containers: queues, stacks, sets, randomized sets, ...

Interfaces can come with some equations one expects every implementation to satisfy. For example:

```ocaml
fold f base empty == base
```

The equations specify some, but not all of the behavior of the module (eg: stacks and queues remove elements in different orders)
Monads

A monad is just a particular interface. Two views:

– (1) interface for a very generic container, with operations designed to support composition of computations over the contents of containers

– (2) interface for an abstract computation that does some “bookkeeping” on the side. By “bookkeeping” we mean “effects”. Once again, the support for composition is key.

– since functional programmers know that functions are data, the two views actually coincide
Monads

A monad is just a particular interface. Two views:

– (1) interface for a very generic container
  • supports composition of computations over the contents of containers
– (2) interface for an abstract computation that does some “book keeping.”
  • bookkeeping is code for “has an effect”. Once again, the support for composition is key.
– since functional programmers know that functions are data, the two views actually coincide

Many different kinds of monads:

– monads for handling/accumulating errors
– monads for processing collections en masse
– monads for logging strings that should be printed
– monads for coordinating concurrent threads (see OCaml Async library)
– monads for backtracking search
– monads for transactional memory
Monads

A monad is just a particular interface. Two views:

- (1) interface for a very generic container
  • supports composition of computations over the contents of containers
- (2) interface for an abstract computation that does some “book keeping.”
  • bookkeeping is code for “has an effect”. Once again, the support for composition is key.
- since functional programmers know that functions are data, the two views actually coincide

Because a monad is just a particular interface (with many useful implementations), you can implement monads in any language

- But, Haskell is famous for them because it has a special built-in syntax that makes monads particularly easy and elegant to use
- F#, Scala have adopted similar syntactic ideas
- Monads also play a very special role in the overall design of the Haskell language due to the confinement of effects
  • Monads are the right way to enable both effects and substitution of equals for equals
What is the monad interface?

Consider first the “container interpretation”:

- ‘a M is a container for values with type ‘a
- return x puts x in the container
- bind c f takes the values in c out of the container and applies f to them, forming a new container holding the results
  - bind c f is often written as: c >>= f

module type MONAD = sig

  type ‘a M

  val return : ‘a -> ‘a M

  val (>>=) : ‘a M -> (‘a -> ‘b M) -> ‘b M

end

+ some equations specifying how return and bind are required to interact
module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

module OptionMonad = struct

  type 'a M = 'a option

  let return x = Some x

  let (>>=) c f =
    match c with
    | None -> None
    | Some v -> f v

end

The Options as a Container

put value in a container
take value v out of a container c and then apply f, producing a new container
The Options as a Container

module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

module OptionMonad = struct

  type 'a M = 'a option

  let return x = Some x

  let (>>=) c f =
  match c with
  None -> None
  | Some v -> f v

end

using the option container:

type file_name = string
val read_file : file_name -> string M

let concat f1 f2 =
  readfile f1 >>=(fun contents1 ->
    readfile f2 >>=(fun contents2 ->
      return (contents1 ^ contents2))

put value in a container

take value v out of a container c and then apply f, producing a new container
Book-keeping Interpretation: The Option Monad as Possibly Erroneous Computation

module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

module ErrorMonad = struct

  type 'a M = 'a option

  let return x = Some x

  let (>>=) c f =
    match c with
    | None -> None
    | Some v -> f v

end

using the error monad:

type file_name = string
val read_file : file_name -> string M

let concat f1 f2 =
  readfile f1 >>= (fun contents1 ->
    readfile f2 >>= (fun contents2 ->
    return (contents1 ^ contents2))

compose bookkeeping:
check to see if error has occurred,
if so return None,
else continue

setting up bookkeeping for error processing
module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

module ListMonad = struct

  type 'a M = 'a list

  let return x = [x]

  let (>>=) c f = List.flatten (List.map f c)

end

random_sample : unit -> int M
monte_carlo : int -> int -> int -> result

let experiments : result M =
  random_sample() >>= (fun s1 ->
    random_sample() >>= (fun s2 ->
      random_sample() >>= (fun s3 ->
        return (monte_carlo s1 s2 s3))
      >>= (fun s3 ->
        return (monte_carlo s1 s2 s3))
    >>= (fun s3 ->
      return (monte_carlo s1 s2 s3))
  >>= (fun s3 ->
      return (monte_carlo s1 s2 s3))
      >>= (fun s3 ->
        return (monte_carlo s1 s2 s3))
    >>= (fun s3 ->
      return (monte_carlo s1 s2 s3))
));

apply f to all elements of the list c, creating a list of lists and then flatten results in to single list
module type MONAD = sig
  type 'a M
  val return : 'a -> 'a M
  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M
end

module ListMonad = struct
  type 'a M = 'a list
  let return x = [x]
  let (>>=) c f = List.flatten (List.map f c)
end

using the list monad:
random_sample : unit -> int M
monte_carlo : int -> int -> int -> result

let experiments : result M =
  random_sample() >>= (fun s1 ->
    random_sample() >>= (fun s2 ->
      random_sample() >>= (fun s3 ->
        return (monte_carlo s1 s2 s3)
    )
  )

one result; no nondeterminism
compose many possible results (c) with a nondeterministic continuation f
A Container with a String on the Side (aka: A logging/prin1ng monad)

```ocaml
module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

module LoggingMonad = struct

  type 'a M = 'a * string

  let return x = (x, "")

  let (>>=) c f =
      let (v, s) = c in
      let (v', s') = f v in
      (v', s ^ s')

  let do x =
      record read x "read it" >>= (fun v ->
          record write v "wrote it" >>= (fun _ ->
              record write v "wrote it again" >>= (fun _ ->
                  return v

end

using the logging monad:

record : ('a -> 'b) -> 'a -> string -> 'b M
let record f x s = (f x, s)

let do x =
    record read x "read it" >>= (fun v ->
    record write v "wrote it" >>= (fun _ ->
    record write v "wrote it again" >>= (fun _ ->
    return v
```

concatenate the log of c with the log produced by running f:

nothing logged yet
A Container with a State on the Side
(aka: A logging/printing monad)

module type MONAD = sig

  type 'a M

  val return : 'a -> 'a M

  val (>>=) : 'a M -> ('a -> 'b M) -> 'b M

end

module StateMonad = struct

  type state = address int map

  type 'a M = 'a * (state -> state)

  let return x = ???

  let (>>=) c f = ???

  let read a = ???

  let write a v = ???

end
Monad Laws

Just like one expects any CONTAINER to behave in a particular way, one has expectations of MONADs.

Left identity: “return does nothing observable”

(1) \( \text{return } v \gg= f \equiv f \cdot v \)

Right identity: “return still doesn’t do anything observable”

(2) \( \text{m } \gg= \text{return } \equiv \text{m} \)

Associativity: “composing m with f first and then doing g is the same as doing m with the composition of f and g”

(3) \( (\text{m } \gg= f ) \gg= g \equiv \text{m } \gg= (\text{fun } x \rightarrow f \cdot x \gg= g) \)
Breaking the Law

Just like one expects any CONTAINER to behave in a particular way, one has expectations of MONADs.

Left identity: “return does nothing observable”

(1) \( \text{return } v >>= f = f \ v \)

```plaintext
module LoggingMonad = struct
  type 'a M = 'a * string

  let return x = (x, "start")

  let (>>=) c f =
    let (v, s) = c in
    let (v',s') = f v in
    (v', s ^ s')
end
```

```plaintext
return 3 >>= fun x -> return x
== (3,"start") >>= fun x -> return x
== (3, "start" ^ "start")
== (3, "startstart")
```

```plaintext
(fun x -> return x) 3
== return 3
== (3, "start")
```
Breaking the Law

What are the consequences of breaking the law?

Well, if you told your friend you’ve implemented a monad and they can use it in your code, they will expect that they can rewrite their code using equations like this one:

```
return x >>= f == f x
```

If you tell your friend you’ve implemented the monad interface but none of the monad laws hold your friend will probably say: Ok, tell me what your functions do then and please stop using the word monad because it is confusing. It is like you are claiming to have implemented the QUEUE interface but insert and remove are First-In, First-Out like a stack.

In Haskell or F# or Scala, breaking the monad laws may have more severe consequences, because the compiler actually uses those laws to do some transformations of your code.
By now you should appreciate the significance of Haskell’s logo, combining the lambda \( \lambda \) and the monadic bind operator \( >>= \).
In languages like ML or Java, there is no way to distinguish between:

- pure functions with type `int -> int`
- effectful functions with type `int -> int`

In Haskell, the programmer can choose when to live in the IO monad and when to live in the realm of pure functional programming.

- **Counter-point:** We have shown that it is useful to be able to build pure abstractions using imperative infrastructure (eg: laziness, futures, parallel sequences, memoization). You can’t do that in Haskell (without escaping the type system via `unsafeI0`).

**Interesting perspective:** It is not Haskell that lacks imperative features, but rather the other languages that lack the ability to have a statically distinguishable pure subset.

A checked pure-impure separation facilitates reasoning about programs, especially concurrent ones.
The Central Challenge

Useful

Useless

Arbitrary effects

No effects
The Challenge of Effects

Arbitrary effects

Plan A
(everyone else)

Nirvana

Plan B
(Haskell)

No effects
Two Basic Approaches: Plan A

Examples
- Regions
- Ownership types
- Rust
  - following from research languages like Vault, Spec#, Cyclone

Arbitrary effects

Default = Any effect
Plan = Add restrictions
Two Basic Approaches: Plan B

Default = No effects
Plan = Selectively permit effects

Types play a major role

Two main approaches:
• Domain specific languages (SQL, Xquery, Google map/reduce)
• Wide-spectrum functional languages + controlled effects (e.g. Haskell)

Value oriented programming
Lots of Cross Over

Useful

Arbitrary effects

Plan A (everyone else)

Nirvana

Useless

No effects

Dangerous

Safe

Envy

according to Simon Peyton Jones

Plan B (Haskell)
Are we there yet?

Useful

Useless

Dangerous

Safe

Is Haskell here?

Or here?

Some issues:

• Can’t encapsulate effectful computation inside pure-functional types.
• Lazy thunks can give a performance penalty

No effects

Nirvana

Plan B (Haskell)
An Assessment and a Prediction

One of Haskell’s most significant contributions is to take purity seriously, and relentlessly pursue Plan B.

Imperative languages will embody growing (and checkable) pure subsets.

-- Simon Peyton Jones