

Intro to Haskell, conclusion

Announcements

- Quiz 7
- HW6 due Tuesday Nov. 29
 - Please to install GHC as soon as possible
 - Post on Submitty forum if you hit a snag
- Released Exam 2 and latest Rainbow grades
 - Please let me know if you find anything amiss

Q2

Lecture Outline

■ Haskell

- Basic syntax and interpreters
- Lazy evaluation
- Static typing and static type inference
- Algebraic data types and pattern matching

- Type classes
- Monads ... and more

Generic Functions in Haskell

- We can generalize a function when a function makes no assumptions about the type:

const :: **a** -> **b** -> **a**

const x y = x

$$a \rightarrow b \rightarrow c = a \rightarrow (b \rightarrow c)$$

apply :: $\overbrace{(a \rightarrow b)}^g \rightarrow \overbrace{a}^x \rightarrow \overbrace{b}^{\text{return type}}$

apply g x = g x

Generic Functions

-- List datatype

data List a = Nil | Cons a (List a)

■ Can we write a function **sum** over a list of **a**'s?

sum :: a -> List a -> a

• **sum n Nil = n**

• **sum n (Cons x xs) = sum (n+x) xs**

■ Type error: **No instance for (Num a) arising from a use of '+'**

■ **a** no longer unconstrained. Type and function definition imply we apply **+** on **a** but

■ **+** is not defined on **all types!**

Haskell Type Classes

- Not to be confused with Java classes/interfaces
- Let us define a **type class** containing the arithmetic and comparison operators:

```
class Num a where  
  (==)  :: a -> a -> Bool  
  (+)   :: a -> a -> a  
  ...  
instance Num Int where  
  x == y = ...  
  ...  
instance Num Float where  
  ...
```

Read: A type **a** is an instance of the type class **Num** if it provides “overloaded” definitions of operators **==**, **+**, ...

Read: **Int** and **Float** are instances of **Num**

Generic Functions with Type Class

(Num a, Eq a) =>

sum :: (Num a) => a -> List a -> a

sum n Nil = n

sum n (Cons x xs) = sum (n+x) xs

- One view of type classes: predicates
 - **(Num a)** is a predicate in type definitions
 - Constrains the specific types we can instantiate a generic function with
- A type class has associated laws

Type Class Hierarchy

```
class Eq a where
```

```
  (==), (/=) :: a -> a -> Bool
```

```
class (Eq a) => Ord a where
```

```
  (<), (<=), (>), (>=) :: a -> a -> Bool
```

```
  min, max           :: a -> a -> a
```

- Each type class corresponds to one concept
- Class constraints give rise to a hierarchy
- **Eq** is a superclass of **Ord**
 - **Ord** inherits specification of **(==)** and **(/=)**
 - Notion of “true subtyping”

Lecture Outline

- Haskell
 - Covered syntax and interpreters
 - Lazy evaluation
 - Static typing and static type inference
 - Algebraic data types and pattern matching
 - Type classes
 - Monads ... and more

Monads

- One source: All About Monads (haskell.org)
- Another source: textbook
- A way to cleanly **compose** computations
 - E.g., **f** may return a value of type **a** or **Nothing**

Composing computations becomes tedious:

case (f s) of

Nothing → Nothing

Just m → case (f m) ...

>>=

- In Haskell, monads **encapsulate** IO and other **imperative** features

An Example: Cloned Sheep

type Sheep = ...

father :: Sheep → Maybe Sheep

father s = ...

mother :: Sheep → Maybe Sheep

mother s = ...

(A sheep may have a mother and a father, just a mother, or just a father.)

maternalGrandfather :: Sheep → Maybe Sheep

maternalGrandfather s = **case** (mother s) **of**

Nothing → Nothing

Just m → father m

An Example

mothersPaternalGrandfather :: Sheep → Maybe Sheep

mothersPaternalGrandfather **s** = **case** (mother **s**) **of**

Nothing → Nothing

Just **m** → **case** (father **m**) **of**

Nothing → Nothing

Just **gf** → father **gf**

- Tedious, unreadable, difficult to maintain
- Monads help!

The Monad Type Class

- Haskell's Monad class requires 2 operations, `>>=` (bind) and `return`

`class Monad m where`

`// >>=` (the bind operation) takes a monad
`// m a`, and a function that takes `a` and turns
`// it into a monad m b`

$\rightarrow (>>=) :: \overbrace{m\ a} \rightarrow \overbrace{(a \rightarrow m\ b)} \rightarrow \overbrace{m\ b}$

`// return` encapsulates a value into the monad

$\rightarrow \text{return} :: a \rightarrow m\ a$

The Maybe Monad

data Maybe a = Nothing | Just a

instance Monad Maybe where

Nothing >>= f = Nothing

(Just x) >>= f = f x

return = Just

(return e) >>=
oneStep >>=
oneStep >>=

■ Cloned Sheep example:

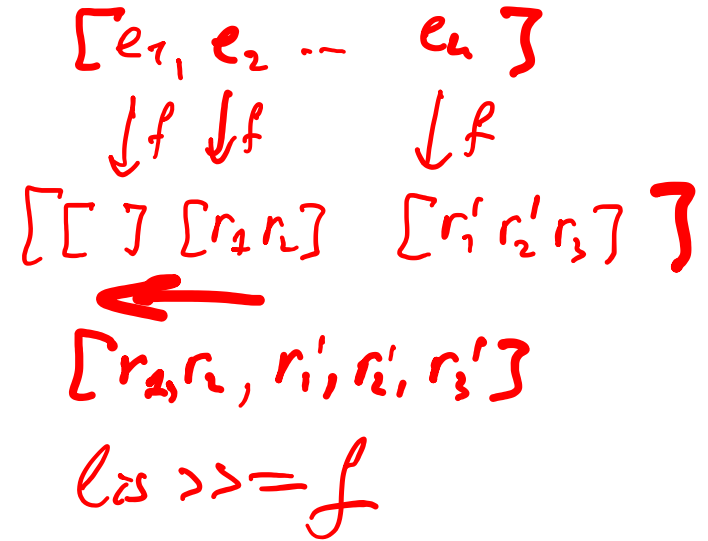
mothersPaternalGrandfather **s =**

→ (return s) >>= mother >>= father >>= father

(Note: if at any point, some function returns Nothing, Nothing gets cleanly propagated.)

The List Monad

- The List type is a monad!
- $\text{lis} \gg= f = \text{concat} (\text{map } f \text{ lis})$
- $\text{return } x = [x]$



Note: $\text{concat} :: [[a]] \rightarrow [a]$

e.g., $\text{concat} [[1,2],[3,4],[5,6]]$ yields $[1,2,3,4,5,6]$

- Use **any** f s.t. $f :: a \rightarrow [b]$. f may yield a list of $0, 1, 2, \dots$ elements of type b , e.g.,

> $f \ x = [x+1]$

> $[1,2,3] \gg= f \ \text{--- yields ?}$

$[2,3,4]$

The List Monad

MaybeToList (Nothing) = []
MaybeToList (Just x) = [x]
Maybe a → [a]

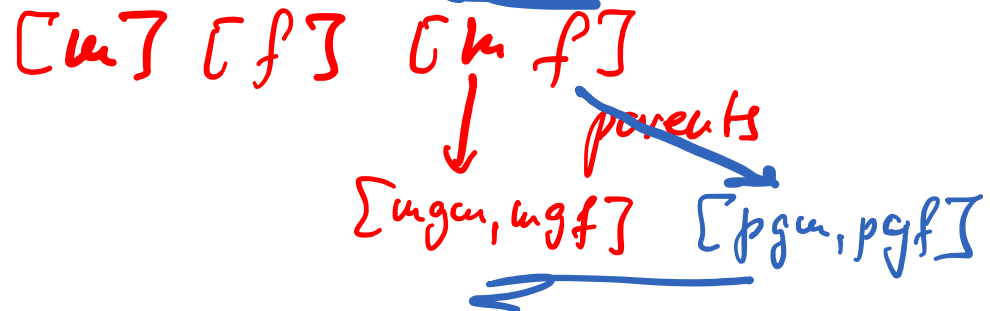
parents :: Sheep → [Sheep]

parents s = MaybeToList (mother s) ++

MaybeToList (father s)

grandParents :: Sheep → [Sheep]

grandParents s = (parents s) >>= parents



The do Notation

- **do** notation is syntactic sugar for monadic bind

> f x = x+1

> g x = x*5

> [1,2,3] >>= (return . f) >>= (return . g)

[2, 3, 4]

[10, 15, 20]

composition

Or

> [1,2,3] >>= \x->[x+1] >>= \y->[y*5]

Or, make encapsulated element explicit with **do**

> **do** { v <- [1,2,3]; w <- (\x->[x+1]) v; (\y->[y*5]) w; }

[1,2,3] >>= (return . id)

1
2

List Comprehensions

```
> [ x | x <- [1,2,3,4] ]
```

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

```
> [ x | x <- [1,2,3,4], x `mod` 2 == 0 ]
```

```
[2,4]
```

[x | x <- [1..], x 'mod' 2 == 0]

```
> [ [x,y] | x <- [1,2,3], y <- [6,5,4] ]
```

```
[[1,6],[1,5],[1,4],[2,6],[2,5],[2,4],[3,6],[3,5],[3,4]]
```

find n euv = head [bool | (var, bool) ← euv, var == n]

List Comprehensions

- List comprehensions are syntactic sugar on top of the **do** notation!

[x | x <- [1,2,3,4]] is syntactic sugar for
do { x <- [1,2,3,4]; return x }

[[x,y] | x <- [1,2,3], y <- [6,5,4]] is syntactic sugar for

do { x <- [1,2,3]; y<-[6,5,4]; return [x,y] }

- Which in turn, we can translate into monadic bind...

So What's the Point of the Monad...

- Conveniently chains (builds) computation

- **Encapsulates** “mutable” state. E.g., **IO**:

openFile :: FilePath -> IO Mode -> **IO** Handle

hClose :: Handle -> **IO** () -- void

hIsEOF :: Handle -> **IO** Bool

hGetChar :: Handle -> **IO** Char

These operations break “referentially transparency”.
For example, **hGetChar** typically returns different value
when called twice in a row!

The End
