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:

\[
\text{const} :: \text{a} \to \text{b} \to \text{a}
\]

\[
\text{const } x \ y = x
\]

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

\[
\text{apply } g \ x = g \ x
\]
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:

```haskell
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

\[ \text{sum :: } (\text{Num } a) \Rightarrow a \rightarrow \text{List } a \rightarrow a \]

\[ \text{sum } n \text{ Nil} = n \]

\[ \text{sum } n \text{ (Cons } x \text{ xs)} = \text{sum } (n+x) \text{ xs} \]

- One view of type classes: predicates
  - \( (\text{Num } a) \) is a predicate in type definitions
  - Constrains the specific types we can instantiate a generic function with
- A type class has associated laws
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”
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 \texttt{compose} computations
  - E.g., \texttt{f} may return a value of type \texttt{a} or \texttt{Nothing}

  Composing computations becomes tedious:
  \begin{verbatim}
  case (f s) of
    Nothing  \rightarrow Nothing
    Just m   \rightarrow case (f m) ...
  \end{verbatim}

- In Haskell, monads \texttt{encapsulate} IO and other \texttt{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!
Haskell’s Monad class requires 2 operations, 
\[ \text{>>=} \] (bind) and \text{return}.

\[
\text{class Monad } m \text{ where}
\]

\[
\text{// } \text{>>=} \text{ (the bind operation) takes a monad}
\]

\[
\text{// } m \text{ a, and a function that takes a and turns it into a monad m b}
\]

\[
\text{\textcolor{red}{\Rightarrow (>>=)} :: m a \rightarrow (a 
\rightarrow m b) \rightarrow m b}
\]

\[
\text{// return encapsulates a value into the monad}
\]

\[
\text{\textcolor{red}{\Rightarrow \text{return}} :: a \rightarrow m a}
\]
The **Maybe Monad**

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

instance Monad Maybe where
  Nothing >>= f = Nothing
  (Just x) >>= f = f x
  return = Just

- 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,… elements of type \( b \), e.g.,

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

\[
> [1,2,3] >>= f \quad --- \text{yields} \quad [2,3,4]
\]
The List Monad

parents :: Sheep \rightarrow [Sheep]
parents s = MaybeToList (mother s) ++
          MaybeToList (father s)

grandParents :: Sheep \rightarrow [Sheep]
grandParents s = (parents s) >>= parents

Programming Languages CSCI 4430, A. Milanova
The **do** Notation

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

> \[ f \ x = x + 1 \]
> \[ g \ x = x \times 5 \]
> \[ [1, 2, 3] >>= (\text{return} \ . \ f) >>= (\text{return} \ . \ g) \]

Or

> \[ [1, 2, 3] >>= (x \rightarrow [x + 1]) >>= (y \rightarrow [y \times 5]) \]

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

> \[ \text{do} \{ \ v \leftarrow [1, 2, 3]; \ w \leftarrow (x \rightarrow [x + 1]) \ v; \ (y \rightarrow [y \times 5]) \ w. \} \]
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,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]]

fixed n env = head [ bool | (var,bool) <- env, 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 -> IOMode -> 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