Intro to Haskell, continued
Announcements

- Moved Quiz 7 to Friday

- HW6 is posted, due Tuesday Nov. 29
  - Please to install GHC as soon as possible
  - Post on Submitty forum if you hit a snag

- We will release Exam 2 grades later this week
Lecture Outline

- Haskell
  - Covered basic syntax and interpreters
  - Lazy evaluation
  - Static typing and static type inference
  - Algebraic data types and pattern matching
  - Type classes
  - Monads … and more
Normal Order to WHNF
Interpreter

Definition by cases on $E ::= x \mid \lambda x.\ E_1 \mid E_1\ E_2$

$\text{interpret}(x) = x$

$\text{interpret}(\lambda x.\ E_1) = \lambda x.\ E_1$

$\text{interpret}(E_1\ E_2) = \text{let } f = \text{interpret}(E_1)\ \text{in case } f \text{ of}$

$\lambda x.\ E_3 \rightarrow \text{interpret}(E_3[E_2/x])$

$\rightarrow f\ E_2$

Haskell syntax:

$\text{let } \ldots \text{ in case } f \text{ of}$
Interpreter Example

\[ ((\lambda x. x) \ y) \ ((\lambda x. x) \ z) \]
Homework

- A step-by-step Normal order to Normal form interpreter

\[ ((\lambda x.x) y) ((\lambda x.x) z) \rightarrow \]

\[ \text{Just } y \ z \ \text{Done step} \]

\[ \text{Just } y \ (\lambda x.x) z \ \text{Done step} \]
Lazy Evaluation

- Unlike Scheme (and most programming languages), Haskell does use lazy evaluation, i.e., normal order reduction.
  - It won’t evaluate an expression until it is needed.

```haskell
> f x y = x*y
> f (5+1) (5+2)
--- evaluates to (5+1) * (5+2)
--- evaluates argument when needed
```
Lazy Evaluation

- In Scheme:

```scheme
(define (fun x y) (* x y))
> (fun (+ 5 1) (+ 5 2)) -> 
  (fun 6 7) -> 6 * 7 -> 42

(define (fun n)
  (cons n (fun (+ n 1)))))
> (car (fun 0))
> (car (cons 0 (fun 1))) -> ...

Infinite recursion
```
Lazy Evaluation

- In Haskell:

\[
\text{fun } n = n : \text{fun}(n+1)
\]

> head (fun 0)

\[
\begin{align*}
> (\lambda p \rightarrow p \text{tru})(\text{fun } 0) & \rightarrow \\
> (\text{fun } 0) \text{tru} & \rightarrow \\
> (0 : \text{fun}(0+1)) \text{tru} & \rightarrow \\
> (\lambda s b \rightarrow b \text{sfs}) 0 \text{fun}(0+1) \text{tru} & \rightarrow \\
> (\lambda b \rightarrow b 0 \text{fun}(0+1)) \text{tru} & \rightarrow \\
> \text{tru} 0 \text{fun}(0+1) & \rightarrow * \rightarrow \text{0} \rightarrow \text{head } n \text{ and tail fun}(n+1)
\end{align*}
\]
Lazy Evaluation

> f x = [] --- f takes x and returns the empty list
> f (repeat 1) --- repeat produces infinite list [1,1...
> []

> head ([1..]) --- [1..] is the infinite list of integers
> 1

- Lazy evaluation allows infinite structures!
def gen(start):
    n = start
    while True:
        yield n
        n = n + 1

gen_obj = gen(0)
print(next(gen_obj))
print(next(gen_obj))
print(next(gen_obj))
print(next(gen_obj))
Lazy Evaluation

- Generate the (infinite) list of even numbers

\[
\text{filter (\(x \rightarrow x \mod 2 == 0\)) [1..]}
\]

- Generate an (infinite) list of “fresh variables”

\[
[1..] \rightarrow ["1", "2", "3", ...]
\]

\[
\text{map (\(\lambda x \rightarrow \text{show } x \text{ ++ } ")"\)} [1..]
\]

Type conversion from int to string

String concatenation
Exercise: write a function that generates the (infinite) list of prime numbers
Static Typing and Type Inference

- Unlike Scheme, which is dynamically typed, Haskell is *statically typed*!
- Unlike Java/C++ we don’t have to write type annotations. Haskell *infers* types!

\[ \text{let } f \ x = \text{head } x \text{ in } f \text{ True} \]

- Couldn't match expected type `[a]` with actual type `Bool`
- In the first argument of `f`, namely `True`
  In the expression: `f` True …
Recall \( \text{apply\_n \ f \ n \ x} \):

\[
\text{apply\_n \ f \ n \ x = if \ n == 0 \ then \ x \ else \ apply\_n \ f \ (n-1) \ (f \ x)}
\]

\[
\text{apply\_n \ : \ (Eq \ b, \ Num \ b) \rightarrow (a \rightarrow a) \rightarrow b \rightarrow a \rightarrow a}
\]

\[
\text{return \ type}
\]

\[
\text{apply\_n \ (\ + \ \ 1) \ True \ 0}
\]

<interactive>:32:1: error:

- Could not deduce (Num Bool) arising from a use of ‘apply\_n’
  from the context: Num t2
  bound by the inferred type of it :: Num t2 => t2
  at <interactive>:32:1-22

- In the expression: apply\_n \ (\ + \ \ 1) \ True \ 0
  In an equation for ‘it’: it = apply\_n \ (\ + \ \ 1) \ True \ 0
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
Algebraic data types are tagged unions (aka sums) of products (aka records)

```
data Shape = Line Point Point
            | Triangle Point Point Point
            | Quad Point Point Point Point
```

Haskell keyword

new constructors (a.k.a. tags, disjuncts, summands)
Line is a binary constructor, Triangle is a ternary …
Algebraic Data Types

- Constructors **create** values of the data type

```latex
let
    l1::Shape
    l1 = Line e1 e2
    t1::Shape = Triangle e3 e4 e5
    q1::Shape = Quad e6 e7 e8 e9
in
```

Programming Languages CSCI 4430, A Milanova (example from MIT 2015 Program Analysis OCW)
Defining a lambda expression

```haskell
type Name = String

data Expr = Var Name
            | Lambda Name Expr
            | App Expr Expr

deriving (Eq, Show)
```

> e1 = Var "x"  // Lambda term x

> e2 = Lambda "x" e1  // Lambda term λx.x
Exercise: Define an ADT for Expressions as in your HW4

```haskell
type Name = String

data Expr = Var Name
            | Val Bool
            | Myand Expr Expr
            | Myor Expr Expr
            | Mylet Name Expr Expr

  deriving (Eq, Show)

evaluate :: Expr \rightarrow [(Name,Bool)] \rightarrow Bool

evaluate e env = ...
```
Pattern Matching

- Examine values of an algebraic data type

```
anchorPnt :: Shape -> Point
anchorPnt s = case s of
  Line  p1 p2 -> p1
  Triangle p3 p4 p5 -> p3
  Quad p6 p7 p8 p9 -> p6
```

- Two points
  - Test: does the given value match this pattern?
  - Binding: if it matches, deconstruct it and bind pattern params to corresponding arguments
Pattern Matching

Pattern matching “deconstructs” a term

> let h:t = "ana" in t
"na"

> let (x,y) = (10,"ana") in x
10
Examples of Algebraic Data Types

data Bool = True | False

data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

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

data Tree a = Leaf a | Node (Tree a) (Tree a)

data Maybe a = Nothing | Just a

Maybe type denotes that result of computation can be a or Nothing. Maybe is a monad.
Type Constructor vs. Data Constructor

Bool and Day are nullary type constructors:

\[
\begin{align*}
\text{data } \text{Bool} &= \text{True} \mid \text{False} \\
\text{data } \text{Day} &= \text{Mon} \mid \text{Tue} \mid \text{Wed} \mid \text{Thu} \mid \text{Fri} \mid \text{Sat} \mid \text{Sun}
\end{align*}
\]

E.g., x::Bool, y::Day

Maybe is a unary type constructor

\[
\begin{align*}
\text{data } \text{Maybe } \text{a} &= \text{Nothing} \mid \text{Just } \text{a}
\end{align*}
\]

E.g., s::Maybe Sheep, e::Maybe Expr

\[
\begin{align*}
e &= \text{Var} \ "x" \ \\
\ell &= \text{Lambda} \ "x" \ (\text{Var} \ "x")
\end{align*}
\]

E.g., \(\ell\) and \(\ell\) are constructed using type constructor Maybe

E.g., \(\ell\) is constructed using data constructors Var and lambda
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
Generic Functions in Haskell

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

\[
\text{const :: } a \rightarrow b \rightarrow a
\]
\[
\text{const } x \ y = x
\]

\[
\text{apply :: } (a\rightarrow b) \rightarrow a \rightarrow b
\]
\[
\text{apply } g \ x = g \ x
\]
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
- Define a **type class** containing the arithmetic 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 operations `==`, `+`, …

Read: `Int` and `Float` are instances of `Num`
Generic Functions with Type Class

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

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

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

- 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
Type Class Hierarchy

- 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
class Eq a where
  (==), (/=) :: a -> a -> Bool

class (Eq a) => Ord where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  min, max :: a -> a -> a
```
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 \to Maybe Sheep  
father = ...  
mother :: Sheep \to Maybe Sheep  
mother = ...  
(A sheep may have a mother and a father, just a mother, or just a father.)  
maternalGrandfather :: Sheep \to Maybe Sheep  
maternalGrandfather s = case (mother s) of  
  Nothing \to Nothing  
  Just m \to 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

  (>>>=) :: m a → (a → m b) → m b

  // return encapsulates a value into the monad

  return :: a → 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:
  ```haskell
  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!

\[
\text{lis >>= f} = \text{concat} (\text{map} \ f \ \text{lis})
\]

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

Note: \(\text{concat} :: [[a]] \rightarrow [a]\)

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

- Use \textbf{any} \(f\) s.t. \(f :: a \rightarrow [b]\). \(f\) may yield a list of 0,1,2,... elements of type \(b\), e.g.,

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

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

parents :: Sheep \rightarrow [Sheep]
parents \mathbf{s} = \text{MaybeToList} (\text{mother } \mathbf{s}) ++ \text{MaybeToList} (\text{father } \mathbf{s})

grandParents :: Sheep \rightarrow [Sheep]
grandParents \mathbf{s} = (\text{parents } \mathbf{s}) >>= \text{parents}
The **do** Notation

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

\[
\begin{align*}
> f \ x &= x+1 \\
> g \ x &= x*5 \\
> [1,2,3] &\gg= (\text{return} \ . \ f) \gg= (\text{return} \ . \ g)
\end{align*}
\]

Or

\[
\begin{align*}
> [1,2,3] &\gg= \{x\mapsto [x+1]\} \gg= \{y\mapsto [y*5]\}
\end{align*}
\]

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

\[
\begin{align*}
> \ do \ {\{ v \ <- [1,2,3]; \ w \ <- (\{x\mapsto [x+1]\}) \ v; \ (\{y\mapsto [y*5]\}) \ w \}}
\end{align*}
\]
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]]
List Comprehensions

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

\[ x \mid x \leftarrow [1,2,3,4] \] is syntactic sugar for
\[
\text{do } \{ x \leftarrow [1,2,3,4]; \text{ return } x \} \\
\]

\[ [x,y] \mid x \leftarrow [1,2,3], y \leftarrow [6,5,4] \] is syntactic sugar for
\[
\text{do } \{ x \leftarrow [1,2,3]; y\leftarrow[6,5,4]; \text{ return } [x,y] \} \\
\]

- 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