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

Haskell syntax:
let …. in
case f of
→

- Definition by cases on **E ::= x | λx. E$_1$ | E$_1$ E$_2$**

interpret(**x**) = **x**

interpret(**λx.E$_1$**) = **λx.E$_1$**

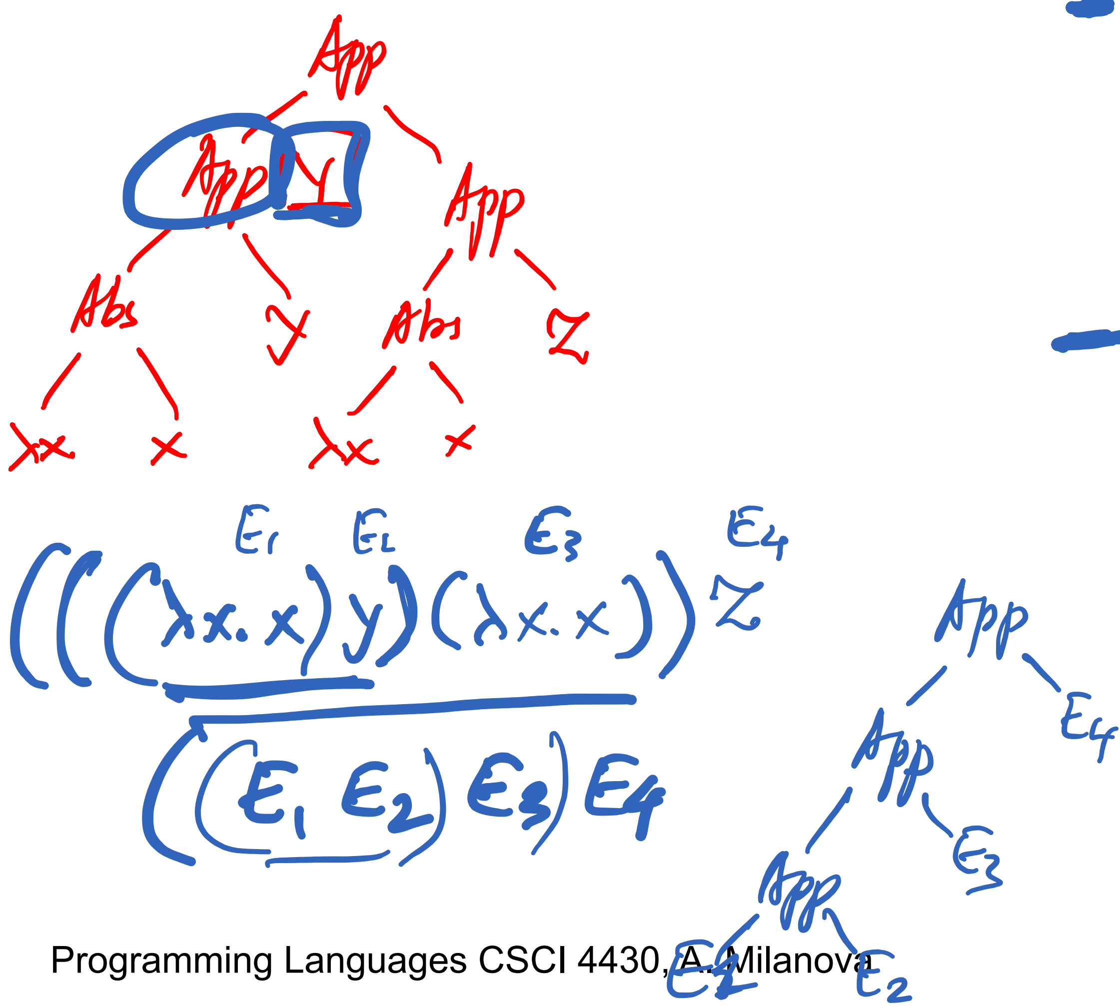
interpret(**E$_1$ E$_2$**) = let **f** = interpret(**E$_1$**)
in case **f** of
λx.E$_3$ → interpret(**E$_3$[E$_2$/x]**)
\- → **f E$_2$**

Apply function before "interpreting" the argument

NO red to WHNF → slide
NO red to HNF
NO red to NF → Homework 6
AO red to WHNF
AO red to HNF
AO red to NF

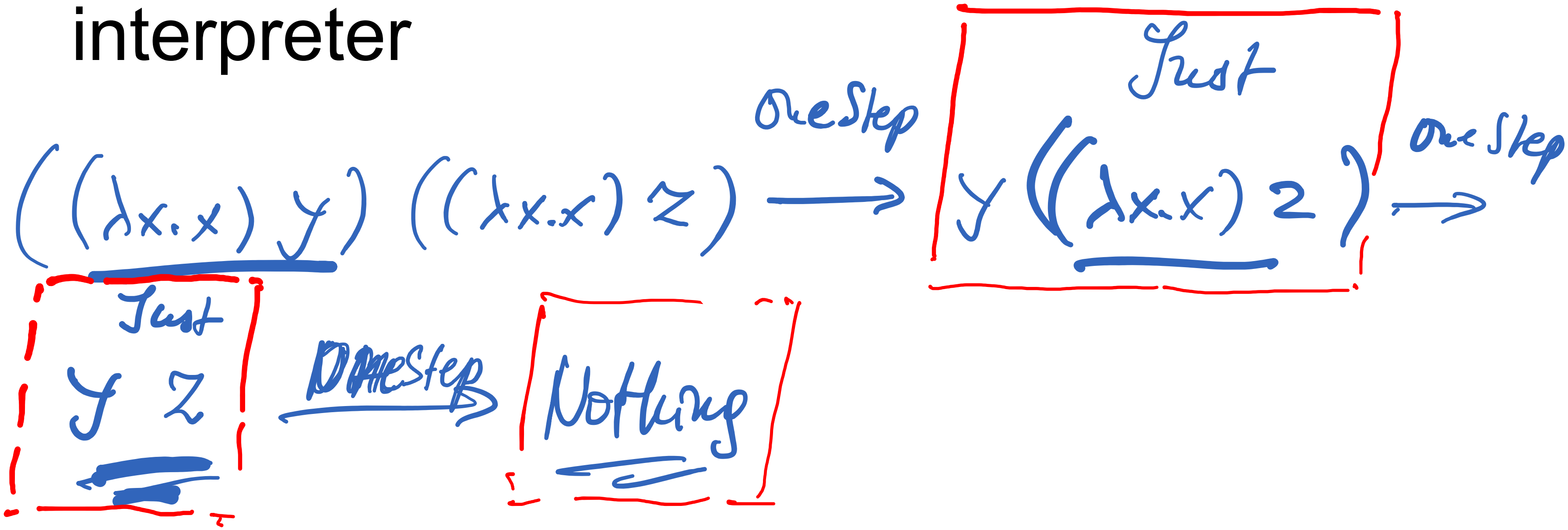
Interpreter Example

- $(\lambda x.x)\ y\ ((\lambda x.x)\ z)$



Homework

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



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

> f x y = x*y

> f (5+1) (5+2)

--- evaluates to **(5+1) * (5+2)**

--- evaluates argument when needed

Lazy Evaluation

- In 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

: denotes “cons” : constructs a list with head **n** and tail **fun(n+1)**

- In Haskell:

fun n = n : fun(n+1)

> head (fun 0) → 0

> (\p → p fru)(fun 0) →

(fun 0) fru →

(0 : fun(0+1)) fru

(\f s b → b f s) 0 fun(0+1) fru →

(\s b → b 0 s) fun(0+1) fru →

(\b → b 0 fun(0+1)) fru →

fru 0 fun(0+1) → 0

(last step throws away second argument fun(0+1), unevaluated)

fru = \x y → x

head = \p → p fru

: = cons = pair = \f s b → b f s

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!

Aside: Python Generators

```
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))
```

Lazy Evaluation

- Generate the (infinite) list of even numbers

filter (\x -> x 'mod' 2 == 0) [1..]

- Generate an (infinite) list of "fresh variables"

[1..] -> ["1_", "2_", "3_", ...]

map (\x -> (show x) ++ "_") [1..]

(show x): type conversion from int to string

++: string concatenation

Lazy Evaluation

- 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!

> f :: [a] -> a
> f x = head x

f :: [a] -> a
True -> b

> let f x = head x in f True

• Couldn't match expected type '[a]' with actual type 'Bool'

• In the first argument of 'f', namely 'True'

In the expression: f True …

Static Typing and Type Inference

- Recall **apply_n f n x:**

> apply_n f n x = if n==0 then x else apply_n f (n-1) (f x)

> apply_n :: (Eq b, Num b) => (a -> a) -> b -> a -> a

f n x return type

> 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

- 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
```

union

Haskell keyword

the new type

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

```
let
  l1::Shape
  l1 = Line e1 e2

  t1::Shape = Triangle e3 e4 e5
  q1::Shape = Quad e6 e7 e8 e9
in
```

…

Algebraic Data Types in Haskell Homework

- Defining a lambda expression

```
type Name = String
data Expr = Var Name
            | Lambda Name Expr
            | App Expr Expr
            deriving (Eq, Show)
```

x == y

e1 == e2

-- Allows comparison and display of Expr values

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

> **e2 = Lambda "x" e1** // Lambda term **λx.x**

Exercise: Define an ADT for Expressions as in your HW4

```
type Name = String
data Expr = Var Name
          | Val Bool
          | Myand Expr Expr
          | Myor Expr Expr
          | Mylet Name Expr Expr
          deriving (Eq, Show)
evaluate :: Expr → [(Name,Bool)] → Bool
evaluate e env = …
```

Pattern Matching

Type signature of anchorPnt: takes a Shape and returns a Point.

- 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

Polymorphic types.
a is a type parameter!

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:

data Bool = True | False

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

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

Maybe is a unary type constructor

data Maybe **a** = Nothing | Just **a**

E.g., s::Maybe Sheep, e::Maybe Expr -- Types, constructed using Type Constructor Maybe

e = Var "x"   e = Lambda "x" (Var "x")

-- expressions 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:

```
const :: a -> b -> a
const x y = x

apply :: (a->b)->a->b
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

```
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

```
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 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 = ...
mother :: Sheep → Maybe Sheep
mother = …
```

(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**

(>>=) :: **m a** → (**a** → **m b**) → **m b**

// return encapsulates a value into the monad

return :: **a** → **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
```

- 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!

lis **>>=** f = concat (map f lis)

return x = [x]

Note: concat::[[**a**]] → [**a**]

e.g., concat [[1,2],[3,4],[5,6]] yields [1,2,3,4,5,6]

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

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

  > **[1,2,3] >>= f** --- yields ?

The **List** Monad

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)
```

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 }
```

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 | 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