Intro to Haskell

So Far

- Essential functional programming concepts
  - Reduction semantics
  - Lists and recursion
  - Higher-order functions
    - Map and fold (also known as reduce)
  - Scoping
  - Evaluation order
- Scheme
- Lambda calculus --- theoretical foundation

Coming Up: Haskell

- Haskell: a functional programming language
  - Rich syntax (syntactic sugar), rich modules
  - Lazy evaluation
  - Static typing and type inference
  - Algebraic data types and pattern matching
  - Monads

Lecture Outline

- Haskell: getting started
- Interpreters for the Lambda calculus

- Key ideas
  - Rich syntax, rich libraries (modules)
  - Lazy evaluation
  - Static typing and polymorphic type inference
  - Algebraic data types and pattern matching

Haskell Resources

- https://www.haskell.org/
  - Try tutorial on front page to get started!
- http://www.seas.upenn.edu/~cis194/spring13/
- Stack Overflow!
- Getting started: slides + tutorial

Getting Started

- Download the Glasgow Haskell Compiler:
  - https://www.haskell.org/ghc
- Run Haskell in interactive mode:
  - **ghci**
  - Type functions in a file (e.g., **fun.hs**), then load the file and call functions interactively

Prelude > :l fun.hs

[1 of 1] Compiling Main (fun.hs, interpreted)

Ok, one module loaded.

***Main > square 25**

Getting Started: Infix Syntax

- You can use prefix syntax, like in Scheme:

> ((+) 1 2)   or   (+) 1 2   or   (+ 1) 2

3

> (quot 5 2) --- or quot 5 2

2

- Or you can use **infix syntax**:

> 1 + 2 + 3

> 5 `quot` 2 --- function value to infix operator

Getting Started: Lists

- Lists are important in Haskell too!

> [1,2]

[1,2]

Syntactic sugar:

> “ana” == [‘a’,’n’,’a’] --- also, [‘a’,’n’,’a’] == ‘a‘ : [‘n...

True --- strings are of type [Char], Char lists

> map (+ 1) [1,2]

[2,3]

- Caveat: in Haskell, all elements of a list must be of same type! You can’t have **[[1,2],2]**!

Getting Started: Lists

- **map**, **foldl**, **foldr**, **filter** and more are built-in!

Note: different order of arguments from ones we defined in Scheme.
foldl : (b * a → b) * b * [a] → b

In Haskell, functions are curried:
foldl:: (b → a → b) → b → [a] → b
→ is right associative:
a → b → c is a → (b → c)

> foldl (+) 0 [1,2,3]

6

> foldr (-) 0 [1,2,3]

2

> filter ((<) 0) [-1,2,0,5]

[2,5]

Getting Started: Functions

- Function definition:

> square x = x*x --- *name params = body*

- Evaluation:

> square 5

25

- Anonymous functions:

> map (\x->x+1) [1,2,3] --- "\x->" is "λx."

[2,3,4]

Getting Started: Functions

- Function definition:

> square x = x*x --- *name params = body*

- Just as in Scheme, you can define a function using the lambda construct:

> square = \x->x*x

> square 5

Getting Started: Higher-order Functions

- Of course, higher-order functions are everywhere!

--- defining **apply_n** in ghci:

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

--- applies f n times on x: e.g., f (f (f (f x)

> apply_n (+ 1) 10 0 or **apply_n (+ 1) 10 0**

10

> fun a b = apply_n (+ 1) a b

Getting Started: Let Bindings

- **let** in Haskell is a **letrec**

> let square x = x*x in square 5

25

> let lis = [‘a’,’n’,’a’] in head lis

‘a’

> let lis = [‘a’,’n’,’a’] in tail lis

“na”

Getting Started: Indentation

- Haskell supports ; and **{ }** to delineate blocks
- Haskell supports indentation too!

```
isEven n =
  let
     even n = if n == 0 then True else odd (n-1)
     odd n = if n == 0 then False else even (n-1)
  in
     even n
> isEven 100
```

Define function in file. Can’t use indentation syntax in ghci!

Lecture Outline

- Haskell: getting started
- Interpreters for the Lambda Calculus

- Key ideas
  - Rich syntax, rich libraries (modules)
  - Lazy evaluation
  - Static typing and polymorphic type inference
  - Algebraic data types and pattern matching

Interpreters for the Lambda Calculus (for Haskell Homework!)

- An interpreter for the lambda calculus is a program that reduces lambda expressions to "answers"

- We must specify
  - Definition of "answer". Which normal form?
  - Reduction strategy. How do we choose redexes in an expression?

An Interpreter

Haskell syntax:
let …. in
case f of
→

- Definition by cases on $\mathbf{E} ::= \mathbf{x} \mid \lambda \mathbf{x}.\ \mathbf{E_1} \mid \mathbf{E_1}\ \mathbf{E_2}$

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

interpret(λ**x.E**$_1$) = λ**x.E**$_1$

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

Apply function before "interpreting" the argument

- What normal form: Weak head normal form
- What strategy: Normal order

Another Interpreter

- Definition by cases on $\mathbf{E} ::= \mathbf{x} \mid \lambda\mathbf{x}.\ \mathbf{E_1} \mid \mathbf{E_1}\ \mathbf{E_2}$

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

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

$$\begin{aligned}\text{interpret}(\mathbf{E_1}\ \mathbf{E_2}) = \ &\text{let } \mathbf{f} = \text{interpret}(\mathbf{E_1})\\ &\qquad \mathbf{a} = \text{interpret}(\mathbf{E_2})\\ &\text{in case } \mathbf{f} \text{ of}\\ &\qquad \lambda\mathbf{x}.\mathbf{E_3} \rightarrow \text{interpret}(\mathbf{E_3}[\mathbf{a}/\mathbf{x}])\\ &\qquad - \rightarrow \mathbf{f}\ \mathbf{a}\end{aligned}$$

- What normal form: Weak head normal form
- What strategy: Applicative order

An Interpreter

- In Haskell Homework
- First, you will write the pseudocode for an interpreter that
  - Reduces to answers in Normal Form
  - Using Normal Order

- Then, you'll code this interpreter in Haskell

Lecture Outline

- Haskell: getting started
- Interpreters for the Lambda Calculus

- Key ideas
  - Rich syntax, rich libraries (modules)
  - Lazy evaluation
  - Static typing and polymorphic type inference
  - Algebraic data types and pattern matching

Lazy Evaluation

- Unlike Scheme (and most programming languages) Haskell does 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)) ->



(define (fun n)
  (cons n (fun (+ n 1))))
> (car (fun 0))
>
```

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)

>

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

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

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!

> 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 (+ 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: a functional programming language
- Key ideas
  - Rich syntax, rich libraries (modules)
  - Lazy evaluation
  - Static typing and polymorphic type inference
  - Algebraic data types and pattern matching

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)
```

> **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
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 corresponding arguments with pattern params

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

The End