Announcements

- Rainbow grades: HW1-6, Quiz1-5, Exam1
- Still grading: HW7, Quiz6, Exam2

- HW8 due today

- HW9, Haskell, out tonight, due Nov. 16th
  - Individual assignment
  - Start early!

Today’s Lecture Outline

- Haskell: a functional programming language
- Key ideas
  - Rich syntax (syntactic sugar), rich libraries (modules)
  - Lazy evaluation
  - Static typing and polymorphic type inference
  - Algebraic data types and pattern matching
  - Type classes
  - Monads … and more

Haskell Resources

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

Getting Started

- Download the Glasgow Haskell Compiler:
  - [https://www.haskell.org/ghc](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
  
  ```haskell
  Prelude > :l fun.hs
  [1 of 1] Compiling Main (fun.hs, interpreted)
  Ok, one module loaded.
  *Main > square 25
  5
  ```

Getting Started: Infix Syntax

- You can use prefix syntax, like in Scheme:
  - `(+) 1 2` --- or `(+) 1 2` 3
  - `(+)` interprets `+` to function value
  - `(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]
  - "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: Functions

- Function definition:
  - `square x = x*x` --- `name params = body`
  - Evaluation:
    - `square 5`
    - 25
  - Anonymous functions:
    - `map ((+) 1) [1,2,3]` --- "\(x\to\)" is "\(\lambda x.\)"
    - [2,3,4]

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., \(f (f (f x))\)
    - `apply_n ((+) 1) 10 0`
    - 10
    - `fun a b = apply_n ((+) 1) a b`
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
```

Interpreters for the Lambda Calculus (for HW9)

- 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 chose redexes in an expression?

An Interpreter (HW9)

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

```
interpret(x) = x
interpret(\lambda x. E) = \lambda x. E_1
interpret(E_1 E_2) = let f = interpret(E_1) 
                      in case f of 
                      \lambda x. E_3 \rightarrow interpret(E_3[f/x]) 
                      - \rightarrow f E_2
```

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

Another Interpreter (HW9)

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

```
interpret(x) = x
interpret(\lambda x. E) = \lambda x. E_1
interpret(E_1 E_2) = let f = interpret(E_1) 
                      a = interpret(E_2)
                      in case f of 
                      \lambda x. E_3 \rightarrow interpret(E_3[a/x]) 
                      - \rightarrow f a
```

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

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
  - Type classes
  - Monads ... and more

An Interpreter (HW9)

- In HW9
- First, you will write the pseudocode for an interpreter that
  - Reduces to answers in Normal Form
  - Uses applicative order reduction
- Then, you’ll code this interpreter in Haskell
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 = [] --- 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 work with infinite structures!

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 ...

Exercise: write a function that returns the (infinite) list of prime numbers

Algebraic Data Types

- Algebraic data types are tagged unions (aka sums) of products (aka records)

```haskell
data Shape = Line e1 e2
            | Triangle e3 e4 e5
            | Quad e6 e7 e8 e9

l1 :: Shape
l1 = Line e1 e2

q1 :: Shape = Triangle e3 e4 e5
```

Fall 18 CSCI 4430, A.Milanova (example from MIT 2015 Program Analysis OCW)
Algebraic Data Types in HW9

- Defining a lambda expression
  
  \[
  \text{type Name} = \text{String} \\
  \text{data Expr} = \begin{cases} 
  \text{Var Name} \\
  \text{Lambda Name Expr} \\
  \text{App Expr Expr} 
  \end{cases}
  \]

\[
> e_1 = \text{Var} "x" // \text{Lambda term } x \\
> e_2 = \text{Lambda} "x" e_1 // \text{Lambda term } \lambda x.x
\]

Exercise: Define an ADT for Expressions in your Scheme HW6

\[
\begin{align*}
\text{type Name} &= \text{String} \\
\text{data Expr} &= \begin{cases} 
  \text{Var Name} \\
  \text{Val Bool} \\
  \text{And Expr Expr} \\
  \text{Or Expr Expr} \\
  \text{Let Name Expr Expr}
  \end{cases}
\end{align*}
\]

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

evaluate e \ env = ...

Examples of Algebraic Data Types

\[
\text{data Bool} = \text{True} | \text{False}
\]

\[
\text{data Day} = \text{Mon} | \text{Tue} | \text{Wed} | \text{Thu} | \text{Fri} | \text{Sat} | \text{Sun}
\]

\[
\text{data List a} = \text{Nil} | \text{Cons a (List a)}
\]

\[
\text{data Tree a} = \text{Leaf a} | \text{Node (Tree a) (Tree a)}
\]

\[
\text{data Maybe a} = \text{Nothing} | \text{Just a}
\]

Maybe type denotes that result of computation can be a or Nothing. Maybe is a monad.

Pattern Matching

- Pattern matching "deconstructs" a term

\[
> \text{let h:t} = "ana" \text{ in t} \\
> "na"
\]

\[
> \text{let } (x,y) = (10,"ana") \text{ in x} \\
> 10
\]

Pattern Matching in HW9

\[
\text{isFree} :: \text{Name} \rightarrow \text{Expr} \rightarrow \text{Bool}
\]

\[
isFree v e = \begin{cases} 
  \text{Var n} & \text{if } (n == v) \text{ then True else False} \\
  \text{Lambda} & \text{...}
  \end{cases}
\]

Type signature of isFree in Haskell, all functions are \textit{curried}, i.e., they take just one argument.

isFree takes a variable name, and returns a function that takes an expression and returns a boolean.

Of course, we can interpret isFree as a function that takes a variable name \textit{name} and an expression \textit{E}, and returns true if variable \textit{name} is free in \textit{E}.
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 with Type Class

\[ \text{sum} :: (\text{Num } a) \Rightarrow a \rightarrow \text{List } a \rightarrow a \]
\[ \text{sum } n \ \text{Nil} = n \]
\[ \text{sum } n \ (\text{Cons } x \ \text{xs}) = \text{sum } (n+x) \ \text{xs} \]

- No. \( a \) no longer unconstrained. Type and function definition imply that + is of type \( a \rightarrow a \rightarrow a \) but + is not defined for all types!

---

Haskell Type Classes

- Define a type class containing the arithmetic operators:

\[ \text{class Num } a \ where \]
\[ (\Rightarrow) :: a \rightarrow a \rightarrow \text{Bool} \]
\[ (+) :: a \rightarrow a \rightarrow a \]

\[ \text{instance Num } \text{Int} \ where \]
\[ x \Rightarrow y = ... \]

\[ \text{instance Num } \text{Float} \ where \]
\[ ... \]

---

Type Class Hierarchy

\[ \text{class Eq } a \ where \]
\[ (\Rightarrow) :: a \rightarrow a \rightarrow \text{Bool} \]

\[ \text{class (Eq } a) \Rightarrow \text{Ord } a \ where \]
\[ (<), (\Rightarrow), (\geq) :: a \rightarrow a \rightarrow \text{Bool} \]
\[ \text{min}, \text{max} :: a \rightarrow a \rightarrow a \]

- Each type class corresponds to one concept
- Class constraints give rise to a hierarchy
- Eq is a superclass of Ord
  - Ord inherits specification of (\Rightarrow) and (\leq)
  - Notion of "true subtyping"

---

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
  - Type classes
  - Monads ... and more
Monads

- A way to cleanly compose computations
  - E.g., \( f \) may return a value of type \( a \) or \( \text{Nothing} \)
  - Composing computations becomes tedious:
    - \( \text{case} \ (f \ s) \ \\
        \text{of} \ \\
        \text{Nothing} \to \text{Nothing} \ \\
        \text{Just} \ m \to \text{case} \ (f \ m) \ldots \)

- 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 = …
  (Note: a cloned sheep may have both parents, or not…)
- maternalGrandfather :: Sheep \( \to \) Maybe Sheep
- maternalGrandfather = …

(maternalGrandfather \( s \) =
\[ \text{case} \ (\text{mother} \ s) \ \\
\text{of} \ \\
\text{Nothing} \to \text{Nothing} \ \\
\text{Just} \ m \to \text{father} \ m \])

Tedious, unreadable, difficult to maintain
Monads help!

The Monad Type Class

- Haskell’s Monad class requires 2 operations, \( \gg= \) (bind) and \( \text{return} \)

class Monad \( m \) where
  - \( \gg= \) (the bind operation) takes a monad \( m \ a \), and a function that takes \( a \) and turns it into a monad \( m \ b \)
  - \( \gg= \) :: \( m \ a \to (a \to m \ b) \to m \ b \)
  - \( \gg= \) returns a value into the monad
  - \( \gg= \) :: \( a \to m \ a \)

The Maybe Monad

- instance Monad Maybe where
  - Nothing \( \gg= \) \( f \) = Nothing
  - (Just \( x \)) \( \gg= \) \( f \) \( x \)
  - return \( x \) = Just

Cloned Sheep example:

mothersPaternalGrandfather \( s \) =
\[ \text{case} \ (\text{mother} \ s) \ \\
\text{of} \ \\
\text{Nothing} \to \text{Nothing} \ \\
\text{Just} \ m \to \text{father} \ m \]

The List Monad

- The List type constructor is a monad
  - \( li \gg= \) \( f \) = concat (map \( f \) \( li \))
  - return \( x \) = \[x\]

Note: concat::\([a]\to[a]\) e.g., concat [[1,2],[3,4],[5,6]] yields [1,2,3,4,5,6]

- Use \( \text{any} \) \( f \) s.t. \( f::a\to[b] \). \( f \) may yield a list of 0,1,2,… elements of type \( b \), e.g.,
  - \( f \) \( x \) = \([x+1]\)
  - \([1,2,3]\gg= f \) --- yields [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

The do Notation (Syntactic Sugar!)

> f x = x+1
> g x = x*5
> [1,2,3] >>= (return . f) >>= (return . g)
  Or
> [1,2,3] >>= (return . \x->x+1) >>= (return . \y->y*5)
  Or
> do { x <- [1,2,3]; y <- (return . f) x; (return . g) y }

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]]
--- Willy's all-pairs function from test

List Comprehensions

- List comprehensions are syntactic sugar …
  [ 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] ] synt. sugar for
  do { x <- [1,2,3]; y<-[6,5,4]; return [x,y] }

Monads

- A way to cleanly compose (build) computations
- A way to encapsulate IO and other imperative features