Types and Type Based Analysis: Lambda Calculus, Intro to Haskell

Announcements

- Welcome back!
- HW5 is out
- Rainbow grades
- Moving on with Types and Type-based Analysis

Outline

- Pure lambda calculus, a review
  - Syntax and semantics (last time)
  - Free and bound variables (last time)
  - Substitution (last time)
  - Rules (last time)
  - Normal forms (last time)
  - Reduction strategies
- Interpreters for the Lambda calculus
- Coding them in Haskell

Syntax of Pure Lambda Calculus

- λ-calculus formulae (e.g., **λx. x y**) are called expressions or terms

- **E ::= x | (λx. E_1) | (E_1 E_2)**
  - A λ-expression is one of
    - Variable: **x**
    - Abstraction (i.e., function definition): **λx. E_1**
    - Application: **E_1 E_2**

$(\lambda x.\, x x)(\lambda z.\, z z)$

Syntactic Conventions

- Parentheses may be dropped from "stand-alone" terms **(E_1 E_2)** and **(λx. E)**
  - E.g., **(f x)** may be written as **f x**

- Function application groups from left-to-right (i.e., <u>it is left-associative</u>)
  - E.g., **x y z** abbreviates **((x y) z)**
  - E.g., **E_1 E_2 E_3 E_4** abbreviates **(((E_1 E_2) E_3) E_4)**
  - Parentheses in **x (y z)** are necessary! Why?

$(\lambda x.\, x x)(\lambda y.\, y)(\lambda z.\, z z) \equiv ((\lambda x.\, x x)(\lambda y.\, y))(\lambda z.\, z z)$

$\neq (\lambda x.\, x x)((\lambda y.\, y)(\lambda z.\, z z))$

Syntactic Conventions

- Application <u>has higher precedence</u> than abstraction
  - Another way to say this is that the scope of the dot extends as far to the right as possible
  - E.g., **λx. x z** = **λx. (x z)** = **(λx. (x z))** = **(λx. (x z)) ≠ ((λx. x) z)**
- WARNING: This is the most common syntactic convention (e.g., Pierce 2002). However, some books give abstraction higher precedence; you might have seen that different convention

Rules (Axioms) of Lambda Calculus

- α rule (α-conversion): renaming of parameter (choice of parameter name does not matter)
  - **λx. E** $\rightarrow_\alpha$ **λz. (E[z/x])** provided **z** is not free in **E**
  - e.g., **λx. x x** is the same as **λz. z z**

- β rule (β-reduction): function application (substitutes argument for parameter)
  - **(λx.E) M** $\rightarrow_\beta$ **E[M/x]**

  Note: **E[M/x]** as defined in class last time
  - e.g., **(λx. x) z** $\rightarrow_\beta$ **z**

Rules of Lambda Calculus: Exercises

- Reduce

(λx. x) y → ?

(λx. x) (λy. y) → ?

((λx.λy.λz. x z (y z)) (λu. u)) (λv. v) → ?

λz. z z

(λy. λz. z (y z)) (λv. v)

λz. z z

Reductions

- An expression **(λx.E) M** is called a redex (for reducible expression)
- An expression is in normal form if it cannot be β-reduced
- The normal form is the meaning of the term, the "answer"

Definitions of Normal Form

- Normal form (NF): a term without redexes
- Head normal form (HNF)
  - **x** is in HNF
  - **(λx. E)** is in HNF if **E** is in HNF
  - **(x E_1 E_2 … E_n)** is in HNF

    $\underbrace{x}_{E_1}\ \underbrace{((\lambda y.y)(\lambda z.z))}_{E_2}$
- Weak head normal form (WHNF)
  - **x** is in WHNF
  - **(λx. E)** is in WHNF
  - **(x E_1 E_2 … E_n)** is in WHNF

Questions

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

- **λz. z z** is in NF, HNF, or WHNF? NF => HNF => WHNF
- **(λz. z z) (λx. x)** is in? Neither
- **λx.λy.λz. x z (y (λu. u))** is in? NF
- **(λx.λy. x) z ((λx. z x) (λx. z x))** is in? Neither (E_1 = (λx.λy. x), E_2 = z, E_3 = ((λx. z x) (λx. z x)))
- **z ((λx. z x) (λx. z x))** is in? HNF and WHNF
- **(λz.(λx.λy. x) z ((λx. z x) (λx. z x)))** is in? WHNF

Simple Reduction Exercise

- **C = λx.λy.λf. f x y**
- **H = λf. f (λx.λy. x)** **T = λf. f (λx.λy. y)**
- What is **H (C a b)**?

→ (λf. f (λx.λy. x)) (C a b)

→ (C a b) (λx.λy. x)

→ ((λx.λy.λf. f x y) a b) (λx.λy. x)

→ (λf. f a b) (λx.λy. x)

→ (λx.λy. x) a b

→ a

Exercise

An expression with no free variables is called combinator. S, I, C, H, T are combinators.

- **S = λx.λy.λz. x z (y z)**
- **I = λx. x**
- What is **S I I I**?

Reducible expression is underlined at each step.

(λx.λy.λz. x z (y z)) I I I

→ **(λy.λz. I z (y z)) I I**

→ **(λz. I z (I z)) I**

→ **I I (I I) = (λx. x) I (I I)**

→ **I (I I) = (λx. x) (I I)**

→ **I I = (λx. x) I → I**

Aside: Trace Semantics

- Models a trace of program execution
- In the imperative world
  - Basic operation: assignment statement
  - Execution (transition system) is a sequence of state transitions

$(\ell_1, \sigma_1) \rightarrow (\ell_2, \sigma_2) \rightarrow \ldots \rightarrow (EXIT, \sigma_{EXIT})$

- Assignment: ℓ_j : **x** = E; ℓ_i : …

$(\ell_j, \sigma) \rightarrow (\ell_i, \sigma[x \leftarrow [[E]](\sigma)])$

- Assignment: ℓ_j : **x** = E_1 Op E_2 ; ℓ_i : …

$(\ell_j, \sigma) \rightarrow (\ell_i, \sigma[x \leftarrow [[E_1]](\sigma) \text{ Op } [[E_2]](\sigma)])$

Aside: Trace Semantics

$E \to E_1 \to E_2 \to \ldots$ NF

- In the functional world
  - Basic operation is function application
  - Execution (transition system) is a sequence of β-reductions

(λx.λy.λz. x z (y z)) I I I

→ **(λy.λz. I z (y z)) I I**

→ **(λz. I z (I z)) I**

…

λx.x

Outline

- Pure lambda calculus, a review
  - Syntax and semantics
  - Free and bound variables
  - Substitution
  - Rules (alpha rule, beta rule)
  - Normal forms
  - Reduction strategies
- Lambda calculus interpreters
- Coding them in Haskell

Reduction Strategy

- Let us look at **$(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))\ (\lambda u.\ u)\ (\lambda v.\ v)$**
- Actually, there are (at least) two "reduction paths":

Path 1: $\underline{(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))\ (\lambda u.\ u)}\ (\lambda v.\ v) \rightarrow_\beta$

$\underline{(\lambda y.\lambda z.\ (\lambda u.\ u)\ z\ (y\ z))\ (\lambda v.\ v)} \rightarrow_\beta$

$(\lambda z.\ \underline{(\lambda u.\ u)\ z}\ ((\lambda v.\ v)\ z)) \rightarrow_\beta (\lambda z.\ z\ (\underline{(\lambda v.\ v)\ z})) \rightarrow_\beta$

$\lambda z.\ z\ z$

Path 2: $\underline{(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))\ (\lambda u.\ u)}\ (\lambda v.\ v) \rightarrow_\beta$

$(\lambda y.\lambda z.\ \underline{(\lambda u.\ u)\ z}\ (y\ z))\ (\lambda v.\ v) \rightarrow_\beta$

$\underline{(\lambda y.\lambda z.\ z\ (y\ z))\ (\lambda v.\ v)} \rightarrow_\beta (\lambda z.\ z\ (\underline{(\lambda v.\ v)\ z})) \rightarrow_\beta$

$\lambda z.\ z\ z$

Reduction Strategy

- A reduction strategy (also called evaluation order) is a strategy for choosing redexes
  - How do we arrive at the normal form (answer)?
- Applicative order reduction chooses the leftmost-innermost redex in an expression
  - Also referred to as call-by-value reduction
- Normal order reduction chooses the leftmost-outermost redex in an expression
  - Also referred to as call-by-name reduction

Reduction Strategy: Examples

- Evaluate **(λx. x x) ((λy. y) (λz. z))**
- Using applicative order reduction:

(λx. x x) (λz. z) →

(λz. z)(λz. z) → λz. z

- Using normal order reduction

(λy. y)(λz. z) ((λy. y)(λz. z))

(λz. z) ((λy. y)(λz. z)) →

(λy. y)(λz. z) → λz. z

Applicative:

x : ~~x~~ NF

λx.E : λx. AP(E)

$E_1 E_2$: $E_1' \leftarrow AP(E_1)$

$E_2' \leftarrow AP(E_2)$

if E_1' is $\lambda x. E_1''$

$AP(E_1''[E_2'/x])$

else $E_1' E_2'$ NF

Normal Order:

HW

Reduction Strategy

- In our examples, both strategies produced the same result. This is not always the case
  - First, look at expression **(λx. x x) (λx. x x)**. What happens when we apply β-reduction to this expression?
  - Then look at **(λz.y) ((λx. x x) (λx. x x))**
    - Applicative order reduction – what happens?
    - Normal order reduction – what happens?

Church-Rosser Theorem

- Normal form implies that there are no more reductions possible

- Church-Rosser Theorem, informally
  - If normal form exists, then it is unique (i.e., result of computation does not depend on the order that reductions are applied; i.e., no expression can have two distinct normal forms)
  - If normal form exists, then normal order will find it

Reduction Strategy

- Intuitively:
- Applicative order (call-by-value) is an eager evaluation strategy. Also known as strict
- Normal order (call-by-name) is a lazy evaluation strategy
- What order of evaluation do most PLs use?

Exercises

- Evaluate **(λx.λy. x y) ((λz. z) w)**
- Using applicative order reduction
- Using normal order reduction

Interpreters

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

- We must specify NF, WHNF, HNF
  - The definition of “answer”. Which normal form?
  - The reduction strategy. How do we choose redexes in an expression? AP or NORM

An Interpreter

Haskell syntax:
let …. in
case f of
->

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

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

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

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**$_2$

NORMAL ORDER:

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

Another Interpreter

- Definition by cases on $\mathbf{E} ::= \mathbf{x} \mid \lambda \mathbf{x}.\ \mathbf{E}_1 \mid \mathbf{E}_1\ \mathbf{E}_2$

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

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

interpret(**E**$_1$ **E**$_2$) = let **f** = interpret(**E**$_1$)

a = interpret(**E**$_2$)

in case **f** of

APPLICATIVE:

λ**x.E**$_3$ → interpret(**E**$_3$**[a/x]**)

\- → **f a**

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

Outline

- Pure lambda calculus, a review
  - Syntax and semantics
  - Free and bound variables
  - Substitution
  - Rules (alpha rule, beta rule)
  - Reduction strategies
  - Normal form
- Lambda calculus interpreters
- Coding them in Haskell

Coding them in Haskell

- In HW5 you will code an interpreter in Haskell
- Haskell
  - A functional programming language

- Key ideas
  - Lazy evaluation
  - Static typing and polymorphic type inference
  - Algebraic data types and pattern matching
  - Monads … and more

Lazy Evaluation

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

> **f x = []** // **f** takes **x** and returns the empty list

> **f (repeat 1)** // returns?

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

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

> **[]**

[2..]

> **head (tail [1..])** // returns?

> **2** // **[1..]** is infinite list of integers

- Lazy evaluation allows us to 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 always have to write type annotations. Haskell infers types!
  - A lot more on type inference later!

> **f x = head x** // **f** returns the head of list **x**

> **f True** // returns?

:type f => [a] -> a

- Couldn't match expected type '[a]' with actual type 'Bool'
- In the first argument of 'f', namely 'True'

  In the expression: f True …

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 in HW5

- Constructors create new values
- Defining a lambda expression

```
type Name = String
data Expr = Var Name
          | Lambda Name Expr
          | App Expr Expr
```

> **e1 = Var “x”** // Lambda term **x**

> **e2 = Lambda “x” e1** // Lambda term **λx.x**

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**] Optional[int]

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

Data Constructors vs Type Constructors

- Data constructor constructs a "program object"
  - E.g., Var, Lambda, and App are data constructs
- Type constructor constructs a "type object"
  - E.g., Maybe is a unary type constructor

Maybe Expr

List Bool

Pattern Matching

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

- Examine values of an algebraic data type

```
anchorPnt :: Shape → Pnt
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 value matches, bind corresponding values of **s** and pattern

Pattern Matching in HW5

→ RIGHT ASSOCIATIVE

isFree::Name → (Expr → Bool)   (isFree v) → f

isFree v e =   (f e) → Bool

case e of

Var n → if (n == v) then True else False

Lambda …

Type signature of **isFree**. In Haskell, all functions are 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 **name** and an expression **E**, and returns true if variable **name** is free in **E**.

Haskell Resources

- http://www.seas.upenn.edu/~cis194/spring13/
- https://www.haskell.org/