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

1

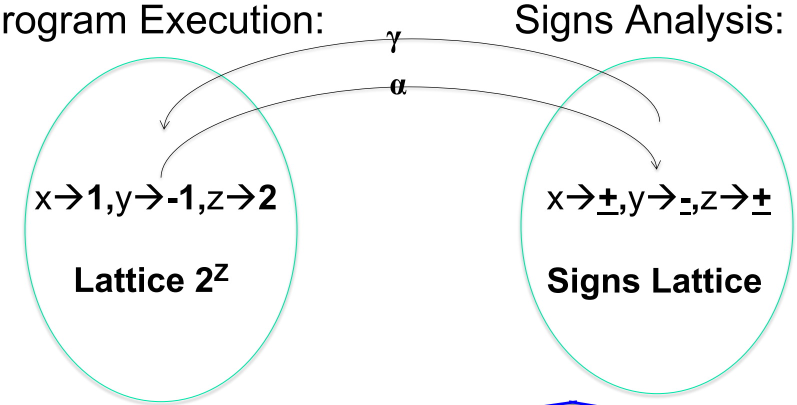
Announcements

- Quiz 4 on Abstract Interpretation
- HW5 is out
- Moving on to Types and Type-based Analysis
- Have a great Spring break!

2

Overview

Program Execution: Signs Analysis:



|[x=y+z]|({(1,-1,2),(0,1,2),…})
=
{(1,-1,2),(…),…}

|[x=y+z]|((±, -, ±))
=
(T, -, 0)

3

3

Outline

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

4

Lambda Calculus

- A theory of functions
  - Theory behind functional programming
  - Turing-complete: any computable function can be expressed and evaluated using the calculus
- Lambda (λ) calculus expresses function definition and function application
  - **f(x)=x*x** becomes **λx. x*x**
  - **g(x)=x+1** becomes **λx. x+1**
  - **f(5)** becomes **(λx. x*x) 5 → 5*5 → 25**

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**

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., it is left-associative)
  - 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?

Syntactic Conventions

- Application has higher precedence 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

Semantics of Lambda Calculus

- An expression has as its meaning the value that results after evaluation is carried out

- Somewhat informally, evaluation is the process of reducing expressions

  E.g., **(λx.λy. x + y) 3 2 → (λy. 3 + y) 2 → 3 + 2 = 5**

  (Note: this example is just an informal illustration. There is no **+** in the pure lambda calculus!)

Free and Bound Variables

- Abstraction **(λx. E)** is also referred as binding
- Variable **x** is said to be bound in **λx. E**

- The set of free variables of **E** is the set of variables that appear unbound in **E**
- Defined by cases on **E**
  - Var **x**: free(**x**) = {**x**}
  - App $\mathbf{E_1}$ $\mathbf{E_2}$: free($\mathbf{E_1}$ $\mathbf{E_2}$) = free($\mathbf{E_1}$) U free($\mathbf{E_2}$)
  - Abs **λx. E**: free(**λx.E**) = free(**E**) - {**x**}

Free and Bound Variables

- A variable **x** is bound if it is in the scope of a lambda abstraction: as in **λx. E**
- Variable is free otherwise

1. (λx. x) y

2. (λz. z z) (λx. x)

Free and Bound Variables

3. λx.λy.λz. x z (y (λu. u))

Free and Bound Variables

- We must take free and bound variables into account when reducing expressions

  E.g., **(λx.λy. x y) (y w)**

  - First, rename bound **y** in **λy. x y** to **z**: **λz. x z**

  (λx.λy. x y) (y w) → (λx.λz. x z) (y w)

  - Second, apply the reduction rule that substitutes **(y w)** for **x** in the body **(λz. x z)**

  (λz. x z) [(y w)/x] → (λz. (y w) z) = λz. y w z

Substitution, formally

- **(λx.E) M → E[M/x]** replaces all free occurrences of **x** in **E** by **M**
- **E[M/x]** is defined by cases on **E**:
  - Var: **y[M/x]** = **M** if **x = y**

    y[M/x] = **y** otherwise
  - App: **(E_1 E_2)[M/x]** = **(E_1[M/x] E_2[M/x])**
  - Abs: **(λy.E_1)[M/x]** = **(λy.E_1)** if **x = y**

    (λy.E_1)[M/x] = **λz.((E_1[z/y])[M/x])** otherwise,

    where **z** NOT in free(**E_1**) U free(**M**) U {**x**}

Substitution, formally

(λx.λy. x y) (y w)

→ **(λy. x y)[(y w)/x]**

→ **λ1_. (((x y)[1_/y])[(y w)/x])**

→ **λ1_. ((x 1_)[(y w)/x])**

→ **λ1_. ((y w) 1_)**

→ **λ1_. y w 1_**

You will have to implement precisely this substitution algorithm in Haskell

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 on previous slide!
  - 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) → ?

Rules of Lambda Calculus: Exercises

(λx.λy.λz. x z (y z)) (λu. u) (λv. v) $\rightarrow_{\alpha\beta}$

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
  - **$(x\ E_1\ E_2\ \ldots\ E_n)$** is in HNF
- Weak head normal form (WHNF)
  - **x** is in WHNF
  - **(λx. E)** is in WHNF
  - **$(x\ E_1\ E_2\ \ldots\ E_n)$** is in WHNF

Questions

- **λz. z z** is in NF, HNF, or WHNF?
- **(λz. z z) (λx. x)** is in?
- **λx.λy.λz. x z (y (λu. u))** is in?

- **(λx.λy. x) z ((λx. z x) (λx. z x))** is in?
- **z ((λx. z x) (λx. z x))** is in?
- **(λz.(λx.λy. x) z ((λx. z x) (λx. z x)))** is in?

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**

Outline

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

Reduction Strategy

- Let us look at **(λx.λy.λz. x z (y z)) (λu. u) (λv. v)**

- Actually, there are (at least) two "reduction paths":

Path 1: **(λx.λy.λz. x z (y z)) (λu. u) (λv. v)** $\rightarrow_\beta$
(λy.λz. (λu. u) z (y z)) (λv. v) $\rightarrow_\beta$
(λz. (λu. u) z ((λv. v) z)) $\rightarrow_\beta$ **(λz. z ((λv. v) z))** $\rightarrow_\beta$
λz. z z

Path 2: **(λx.λy.λz. x z (y z)) (λu. u) (λv. v)** $\rightarrow_\beta$
(λy.λz. (λu. u) z (y z)) (λv. v) $\rightarrow_\beta$
(λy.λz. z (y z)) (λv. v) $\rightarrow_\beta$ **(λz. z ((λv. v) z))** $\rightarrow_\beta$
λz. 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) ((λy. y) (λz. z))

→ **(λx. x x) (λz. z)**

→ **(λz. z) (λz. z) → (λz. z)**

- Using normal order reduction

(λx. x x) ((λy. y) (λz. z))

→ **(λy. y) (λz. z) ((λy. y) (λz. z))**

→ **(λz. z) ((λy. y) (λz. z))**

→ **(λy. y) (λz. z) → (λz. z)**

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
  - The definition of "answer". Which normal form?
  - The reduction strategy. How do we choose redexes in an expression?

An 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.E1) = λx.E1
interpret(E1 E2) = let f = interpret(E1)
                   in case f of
                        λx.E3 -> interpret(E3[E2/x])
                          -   -> f E2
```

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

Another Interpreter

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

```
interpret(x) = x
interpret(λx.E1) = λx.E1
interpret(E1 E2) = let f = interpret(E1)
                       a = interpret(E2)
                   in case f of
                        λx.E3 → interpret(E3[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
  - 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

- Haskell implements 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?

> **[]**

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

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

- Lazy evaluation allows us to work with infinite structures!

37

37

Static Typing and Type Inference

- Unlike Python, 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?

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

38

38

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**

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

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

```
isFree::Name → Expr → Bool
isFree v e =
   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**.