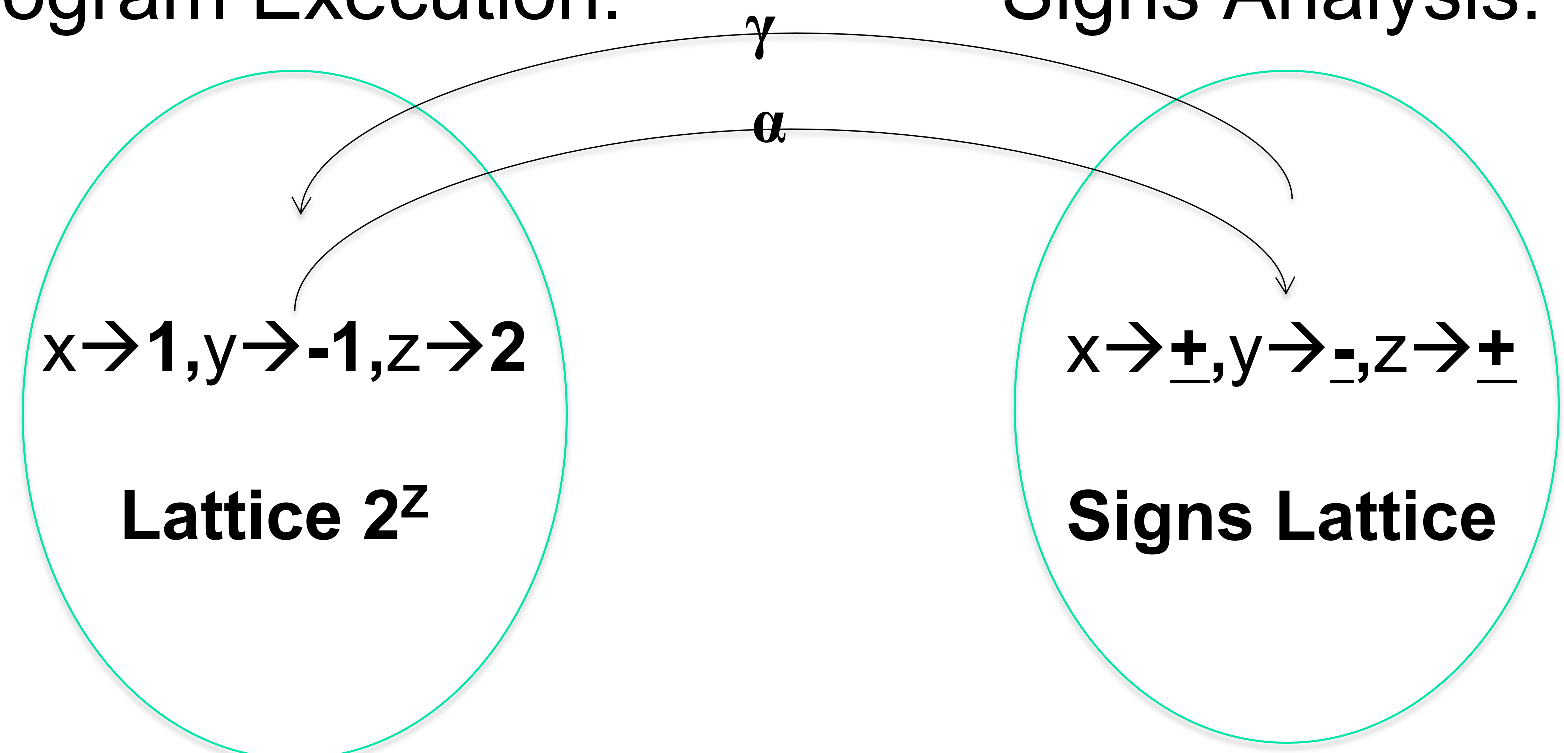
Types and Type Based Analysis: Lambda Calculus, Intro to Haskell

Announcements

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

Overview

Program Execution: Signs Analysis:



$\|[x=y+z]\|(\langle x\to 1, y\to -1, z\to 2\rangle, \langle\ldots\rangle\ldots)$
=
$(\langle x\to 1, y\to -1, y\to 2\rangle, \langle\ldots\rangle\ldots)$

$\widehat{\|[x=y+z]\|}(\langle a_x\to \underline{+},\ a_y\to \underline{-},\ a_z\to \underline{+}\rangle)$
=
$(\langle a_x\to T,\ a_y\to \underline{-},\ a_z\to \underline{0}\rangle)$

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

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., λ**x. 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., <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?

$x\ y\ z = (x\ y)\ z \neq x\ (y\ z)$

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., $\lambda x.\ x\ z = \lambda x.\ (\ x\ z\ ) = (\ \lambda x.\ (\ x\ z\ )\ ) = (\ \lambda x.\ (x\ z)\ ) \neq (\ (\ \lambda x.\ x\ )\ z\ )$
- 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., $(\lambda x.\lambda y.\ x + y)\ 3\ 2 \rightarrow (\lambda y.\ 3 + y)\ 2 \rightarrow 3 + 2 = 5$

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

Free and Bound Variables

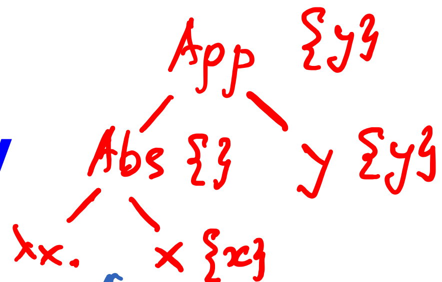
- Abstraction **(λx. E)** is also referred as binding
- Variable **x** is said to be bound in **λx. E**

- 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 λ**x. E**: free(λ**x.E**) = free(**E**) - {**x**}

Free and Bound Variables

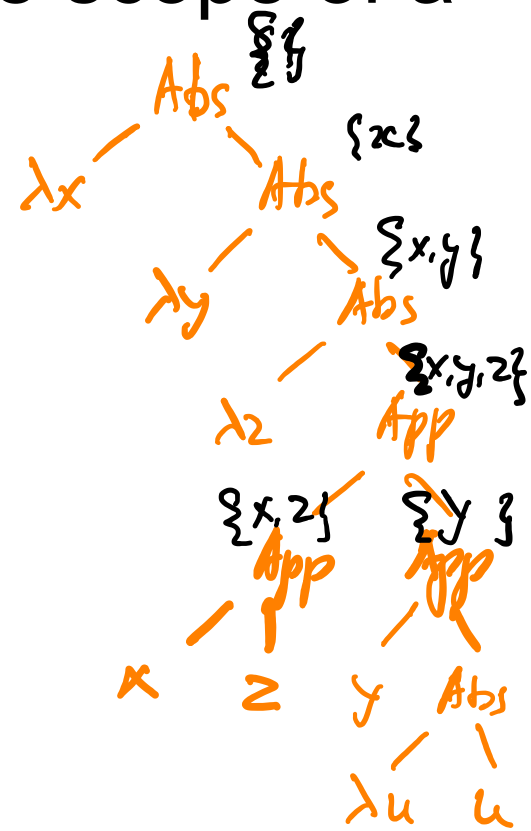
- A variable **x** is bound if it is in the scope of a lambda abstraction: as in **λx. E**
- Variable is free otherwise

1. (λx. x) y



2. (λz. z z) (λx. x)

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)** $\lambda y.\ \cancel{(y\ w)}\ y$

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

$(\lambda \cancel{x}. E_1)[M/x]$

- $(\lambda$**x.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: $(\lambda y.E_1)[M/x]$ = $(\lambda y.E_1)$ if **x = y**
    $(\lambda y.E_1)[M/x]$ = $\lambda z.((E_1[z/y])[M/x])$ 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)
  - λ**x. E** $\rightarrow_\alpha$ λ**z. (E[z/x])** provided **z** is not free in **E**
  - e.g., λ**x. x x** is the same as λ**z. z z**

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

    Note: **E[M/x]** as defined on previous slide!
  - e.g., **(**λ**x. x) z** $\rightarrow_\beta$ **z**

Rules of Lambda Calculus: Exercises

- Reduce

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

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

$((\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))\ (\lambda u.\ u))(\lambda v.\ v) \rightarrow ?$

Rules of Lambda Calculus: Exercises

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

$(\lambda y.\ \lambda z.\ (\lambda u.\ u)\ z\ (y\ z))\ (\lambda v.\ v) \rightarrow$

$(\lambda y.\ \lambda z.\ z\ (y\ z))\ (\lambda v.\ v) \rightarrow$

$\lambda z.\ z\ ((\lambda v.\ v)\ z) \rightarrow$

$\lambda z.\ z\ z$

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

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

- 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

- $\lambda \mathbf{z.\ z\ z}$ is in NF, HNF, or WHNF? NF
- $(\lambda \mathbf{z.\ z\ z})\ (\lambda \mathbf{x.\ x})$ is in? Neither
- $\lambda \mathbf{x.}\lambda \mathbf{y.}\lambda \mathbf{z.\ x\ z\ (y\ (}\lambda \mathbf{u.\ u))}$ is in? NF
- $\overbrace{(\lambda \mathbf{x.}\lambda \mathbf{y.\ x})}^{E_1}\ \overbrace{\mathbf{z}}^{E_2}\ \overbrace{\mathbf{((}\lambda \mathbf{x.\ z\ x)\ (}\lambda \mathbf{x.\ z\ x))}}^{E_3}$ is in? Neither
- $\mathbf{z\ ((}\lambda \mathbf{x.\ z\ x)\ (}\lambda \mathbf{x.\ z\ x))}$ is in? Head and W Head
- $(\lambda \mathbf{z.}\underbrace{(\lambda \mathbf{x.}\lambda \mathbf{y.\ x})}_{E_1}\ \underbrace{\mathbf{z}}_{E_2}\ \underbrace{\mathbf{((}\lambda \mathbf{x.\ z\ x)\ (}\lambda \mathbf{x.\ z\ x))}}_{E_3}\mathbf{)}$ is in? WHNF

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.

<u>(λx.λy.λz. x z (y z)) I</u> I I

→ **<u>(λy.λz. I z (y z)) I</u> I**

→ **<u>(λz. I z (I z)) I</u>**

→ **I I (I I) = <u>(λx. x) I</u> (I I)**

→ **I (I I) = <u>(λx. x) (I I)</u>**

→ **<u>I I</u> = <u>(λx. x) I</u> → 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
- 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

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

$\underline{(\ \lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z)\ )\ I}\ I\ I$

$\rightarrow \underline{(\lambda y.\lambda z.\ I\ z\ (y\ z)\ )\ I}\ I$

$\rightarrow \underline{(\lambda z.\ I\ z\ (I\ z)\ )\ I}$

…

$\underline{\lambda x.x}$

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 $(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))\ (\lambda u.\ u)\ (\lambda v.\ v)$

- 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 **$(\lambda x.\ x\ x)\ (\ (\lambda y.\ y)\ (\lambda z.\ z)\ )$**
- Using applicative order reduction:

$(\lambda x.\ x\ x)\ (\ \underline{(\lambda y.\ y)\ (\lambda z.\ z)}\ )$

$\rightarrow$ **$\underline{(\lambda x.\ x\ x)\ (\lambda z.\ z)}$**

$\rightarrow$ **$\underline{(\lambda z.\ z)\ (\lambda z.\ z)} \rightarrow (\lambda z.\ z)$**

- Using normal order reduction

$\underline{(\lambda x.\ x\ x)\ (\ (\lambda y.\ y)\ (\lambda z.\ z)\ )}$

$\rightarrow$ **$\underline{(\lambda y.\ y)\ (\lambda z.\ z)}\ (\ (\lambda y.\ y)\ (\lambda z.\ z)\ )$**

$\rightarrow$ **$\underline{(\lambda z.\ z)\ (\ (\lambda y.\ y)\ (\lambda z.\ z)\ )}$**

$\rightarrow$ **$\underline{(\lambda y.\ y)\ (\lambda z.\ z)} \rightarrow (\lambda z.\ z)$**

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 $(\lambda x.\lambda y.\ x\ y)\ ((\lambda z.\ z)\ w)$
- 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 $\mathbf{E} ::= \mathbf{x} \mid \lambda\mathbf{x}.\ \mathbf{E_1} \mid \mathbf{E_1}\ \mathbf{E_2}$

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

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

Another Interpreter

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

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

- 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

- 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?

> **[]**

> **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?

- 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**

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**.

Haskell Resources

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