Announcements

- Quiz 6

- HW5 due on Friday
- Exam 2 in one week
- Practice tests on Submitty
- Review and practice on Friday

Lambda Calculus

Lecture Outline

- Quiz 6
- Lambda calculus
  - Reduction strategies (catch-up)

- Applied lambda calculus
- Introduction to types and type systems
- Simply typed lambda calculus (System F_1)
  - If we have time

Reduction Strategy

(S I) I

- Look again 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":

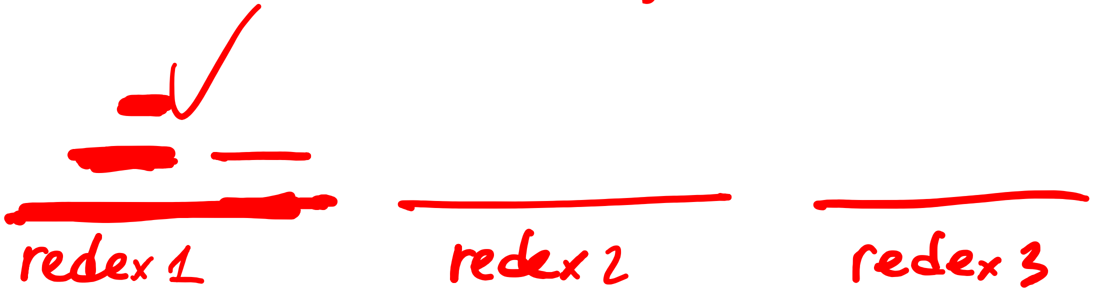
Path 1: $\underline{(\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z))\ (\lambda u.\ u)}\ (\lambda v.\ v) \rightarrow_\beta$
$\underline{(\lambda y.\lambda z.\ (\lambda u.\ u)\ z\ (y\ z))\ (\lambda v.\ v)} \rightarrow_\beta$
$(\lambda z.\ \underline{(\lambda u.\ u)\ z}\ ((\lambda v.\ v)\ z)) \rightarrow_\beta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

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

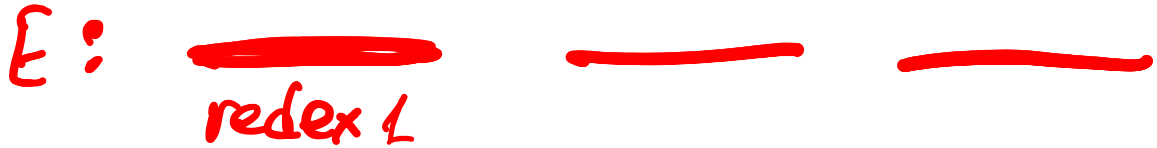
- A reduction strategy (also called evaluation order) is a strategy for choosing redexes
  - How do we arrive at a normal form (answer)?
- Applicative order reduction chooses the leftmost-innermost redex in an expression
  - Also referred to as call-by-value reduction

E.g.



Reduction Strategy

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



Reduction Strategy: Examples

- Evaluate $\underline{(\lambda x.\, x\; x)\; (\, \underline{(\lambda y.\, y)\; (\lambda z.\, z)}\,)}$

Applicative order

Normal order

- Using applicative order reduction:

- Using normal order reduction

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?

$$(\lambda x.\, x\, x)\ (\lambda x.\, x\, x) \rightarrow_\beta (\lambda x.\, x\, x)\ (\lambda x.\, x\, x) \rightarrow_\beta \ldots$$

  - Then look at $\underline{\mathbf{(\lambda x.\lambda y.\ y)\ (\underline{(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)})}}$ **z**
    - 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
- Church-Rosser Theorem, more formally:
  - For all pure λ-expressions **M**, **P** and **Q**, if $\mathbf{M} \rightarrow^* \mathbf{P}$ and $\mathbf{M} \rightarrow^* \mathbf{Q}$, then there must exist an expression **R** such that $\mathbf{P} \rightarrow^* \mathbf{R}$ and $\mathbf{Q} \rightarrow^* \mathbf{R}$

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 programming languages use? $(e_1\ \underline{e_2}\ \underline{e_3})$

Exercises

- Evaluate $\underline{(\lambda x.\lambda y.\ x\ y)\ (\underline{(\lambda z.\ z)\ w})}$
- Using applicative order reduction
- Using normal order reduction

Exercise

$\lambda x. \lambda y. \lambda z. \ldots$

- Let **S = λxyz. x z (y z)** and let **I = λx. x**
- Evaluate **S I I I** using applicative order

Exercise

- Let **S = λxyz. x z (y z)** and let **I = λx. x**
- Evaluate **S I I I** using normal order

Lecture Outline

- Quiz 6
- Lambda calculus
    - Reduction strategies (catch-up)

- Applied lambda calculus
- Introduction to types and type systems
- Simply typed lambda calculus (System F_1)
    - If we have time

Applied Lambda Calculus (from Sethi)

- **$E ::= c \mid x \mid (\lambda x.E_1) \mid (E_1\ E_2)$**

An applied lambda calculus augments the pure lambda calculus with constants. It defines its set of constants and reduction rules. For example:

Constants:

if, true, false

(all these are λ terms,

e.g., true=$\lambda \mathbf{x}.\lambda \mathbf{y}.\ \mathbf{x}$)

0, iszero, pred, succ

Reduction rules:

$\text{if true } M\ N \rightarrow_\delta M$

$\text{if false } M\ N \rightarrow_\delta N$

$\text{iszero } 0 \rightarrow_\delta \text{true}$

$\text{iszero } (\text{succ}^k\ 0) \rightarrow_\delta \text{false},\ k>0$

$\text{iszero } (\text{pred}^k\ 0) \rightarrow_\delta \text{false},\ k>0$

$\text{succ } (\text{pred } M) \rightarrow_\delta M$

$\text{pred } (\text{succ } M) \rightarrow_\delta M$

From an Applied Lambda Calculus to a Functional Language

Construct	Applied λ-Calculus	A Language (ML)
Variable	x	x
Constant	c	c
Application	M N	M N
Abstraction	$\lambda x.M$	fun x => M
Integer	$\text{succ}^k\ 0$, k>0	k
	$\text{pred}^k\ 0$, k>0	-k
Conditional	if P M N	if P then M else N
Let	$(\lambda x.M)\ N$	let val x = N in M end

((lambda (x) M)

\x → x+1

let polymorphism →

(let ((x N)) M)

The Fixed-Point Operator

- One more constant and one more rule:

  fix $\qquad$ $\underline{\textbf{fix } M \rightarrow_{\delta} M\ (\textbf{fix } M)}$

  $\boxed{M(M(M\ldots))}$

- Needed to define recursive functions:

$$\underline{\textbf{plus}}\ \mathbf{x\ y} = \begin{cases} \mathbf{y} & \text{if } \mathbf{x = 0} \\ \underline{\textbf{plus}}\ (\mathbf{pred\ x})\ (\mathbf{succ\ y}) & \text{otherwise} \end{cases}$$

$\boxed{x\text{-}1}$ $\qquad$ $\boxed{y+1}$

- Therefore, we need:

$\underline{\textbf{plus}} = \lambda\mathbf{x}.\lambda\mathbf{y}.\ \textbf{if}\ (\textbf{iszero}\ \mathbf{x})\ \mathbf{y}\ (\underline{\textbf{plus}}\ (\textbf{pred}\ \mathbf{x})\ (\textbf{succ}\ \mathbf{y}))$

The Fixed-Point Operator

- But how do we define **plus**?

Define **plus** = **fix M**, where

M = λ**f**. λ**x**.λ**y**. **if** (**iszero x**) **y** (**f** (**pred x**) (**succ y**))

$$(\text{fix } M)\ x\ y \overset{?}{=} \begin{cases} y & \text{if } x \text{ is } 0 \\ (\text{fix } M)\ (\text{pred } x)\ (\text{succ } y) & \text{otherwise} \end{cases}$$

We must show that

fix M $=_{\delta\beta}$

λx.λy. if (iszero x) y ((fix M) (pred x) (succ y))

plus = fix M

(1) $\text{fix } M \rightarrow_\beta M\ (\text{fix } M)$

(2) $M = \lambda f.\ \lambda x.\ \lambda y.\ \text{if } (\text{iszero } x)\ y\ (f\ (\text{pred } x)\ (\text{succ } y))$

$$(\text{fix } M)\ a\ b \overset{?}{=} \begin{cases} b & \text{if } a \text{ is } 0 \\ (\text{fix } M)\ (\text{pred } a)\ (\text{succ } b) & \text{otherwise} \end{cases}$$

$$(\underline{\text{fix } M})\ a\ b \longrightarrow \underline{M\ (\text{fix } M)}\ a\ b =$$

$$\underline{(\lambda f.\ \lambda x.\ \lambda y.\ \text{if } (\text{iszero } x)\ y\ (f\ (\text{pred } x)\ (\text{succ } y)))\ \boxed{(\text{fix } M)}}\ a\ b$$

$$\rightarrow_\beta (\lambda x.\ \lambda y.\ \text{if } (\text{iszero } x)\ y\ ((\text{fix } M)\ (\text{pred } x)\ (\text{succ } y)))\ a\ b \rightarrow_\beta$$

$$\underline{\text{if } (\text{iszero } a)\ b\ ((\text{fix } M)\ (\text{pred } a)\ (\text{succ } b))}$$

The Fixed-Point Operator

We have to show

fix M $=_{\delta\beta}$

λx.λy. if (iszero x) y ((fix M) (pred x) (succ y))

fix M $=_{\delta}$ **M (fix M) =**

(λf. λx.λy. if (iszero x) y (f (pred x) (succ y))) (fix M) $=_{\beta}$

λx.λy. if (iszero x) y ((fix M) (pred x) (succ y))

The Fixed-Point Operator

Define **times** =

fix (λf.λx.λy. if (iszero (pred x)) y (plus y (f (pred x) y)))

Exercise: define **factorial** = ?

The Y Combinator

$\phi(X) = X$

[redacted] [redacted]

- **fix** is, of course, a lambda expression!
- One possibility, the famous Y combinator:

Y = λf. (λx. f (x x)) (λx. f (x x))

Show that **Y M** indeed reduces to **M (Y M)**

$$YM = (\lambda f. (\lambda x. f(x\,x))(\lambda x. f(x\,x)))\,M \rightarrow_\beta$$

$$\underline{(\lambda x. M(x\,x))(\lambda x. M(x\,x))} \rightarrow_\beta$$

$$M(\underbrace{(\lambda x. M(x\,x))(\lambda x. M(x\,x))}_{YM}) = M(YM)$$

Types!

- Constants are convenient
- But they raise problems because they permit “bad” terms such as
  - **if 0 y z** (0 doesn’t make sense as first argument; true/false values do)
  - **(0 x)** (0 does not apply as a function)
  - **succ true** (undefined in our language)
  - **plus true 0** etc.

Types!

- Why types?
  - Safety. Catch semantic errors early
  - Data abstraction. Simple types and ADTs
  - Documentation (statically-typed languages only)
    - Type signature is a form of specification!
- Statically typed vs. dynamically typed languages
- Type annotations vs. type inference
- Type safe vs. type unsafe

Type System

- Informally, it is a set of rules that we apply on syntactic constructs in the language

- In type theory, it is defined in terms of
  - Type environment
  - Typing rules, also called type judgments
  - This is typically referred to as the type system

Example, More On This Later

looks up the type of **x** in environment **Γ**

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x : \tau}$$

(Variable)

$$\frac{\Gamma \vdash E_1 : \sigma \to \tau \qquad \Gamma \vdash E_2 : \sigma}{\Gamma \vdash (E_1\ E_2) : \tau}$$

(Application)

binding: augments environment **Γ** with binding of **x** to type **σ**

$$\frac{\Gamma, x:\sigma \vdash E_1 : \tau}{\Gamma \vdash (\lambda x:\sigma.\ E_1) : \sigma \to \tau}$$

(Abstraction)

Type System

- A type system either accepts a term (i.e., term is “well-typed”), or rejects it
- Type soundness, also called type safety
  - Well-typed terms never “go wrong”
  - A sound type system never accepts a term that can “go wrong”
  - A complete type system never rejects a term that cannot “go wrong”
  - Whether a term can “go wrong” is undecidable
    - Type systems choose type soundness (i.e., safety)

Putting It All Together, Formally

- Simply typed lambda calculus (System F_1)

- Syntax of the simply typed lambda calculus
- The type system: type expressions, environment, and type judgments
- The dynamic semantics
  - Stuck states
- Type soundness theorem: progress and preservation theorem

Type Expressions

- Introducing type expressions
  - $\tau ::= \mathbf{b} \mid \tau \to \tau$
  - A type is a basic type **b** (we will only consider **int** and **bool**, for simplicity), or a function type
- Examples

  int

  bool → (int → int) // → is right-associative, thus can write just **bool → int → int**
- Syntax of terms:
  - $\mathbf{E ::= x \mid (\, \lambda x{:}\tau.\ E_1\,) \mid (\, E_1\ E_2\,)}$

Type Environment and Type Judgments

- A term in the simply typed lambda calculus is
  - Type correct i.e., well-typed, or
  - Type incorrect
- The rules that judge type correctness are given in the form of type judgments in an environment
  - Environment $\Gamma \;|\!-\; E : \tau$ (|- is the turnstile)
  - Read: environment **Γ** entails that **E** has type **τ**
  - Type judgment

Premises

$$\frac{\Gamma \;|\!-\; E_1 : \sigma \to \tau \qquad \Gamma \;|\!-\; E_2 : \sigma}{\Gamma \;|\!-\; (E_1\; E_2) : \tau}$$

Conclusion

Semantics

looks up the type of **x** in environment **Γ**

$$\frac{\mathbf{x}:\tau \in \Gamma}{\Gamma \mid\!\!- \mathbf{x} : \tau} \qquad \text{(Variable)}$$

$$\frac{\Gamma \mid\!\!- E_1 : \sigma \to \tau \quad \Gamma \mid\!\!- E_2 : \sigma}{\Gamma \mid\!\!- (E_1\ E_2) : \tau} \qquad \text{(Application)}$$

binding: augments environment **Γ** with binding of **x** to type **σ**

$$\frac{\Gamma, \mathbf{x}:\sigma \mid\!\!- E_1 : \tau}{\Gamma \mid\!\!- (\lambda \mathbf{x}:\sigma.\ E_1) : \sigma \to \tau} \qquad \text{(Abstraction)}$$

Examples

- Deduce the type for

  λ**x: int.**λ**y: bool. x** in the **nil** environment

Extensions

$$\frac{}{\Gamma \mid\!- c : \text{int}} \qquad \frac{\Gamma \mid\!- E_1 : \text{int} \qquad \Gamma \mid\!- E_2 : \text{int}}{\Gamma \mid\!- E_1 + E_2 : \text{int}}$$

$$\frac{\Gamma \mid\!- E_1 : \text{int} \qquad \Gamma \mid\!- E_2 : \text{int}}{\Gamma \mid\!- E_1 = E_2 : \text{bool}}$$

(Comparison)

$$\frac{\Gamma \mid\!- b : \text{bool} \quad \Gamma \mid\!- E_1 : \tau \quad \Gamma \mid\!- E_2 : \tau}{\Gamma \mid\!- \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau}$$

Examples

- Is this a valid type?

  Nil |- λx: int.λy: bool. x+y : int → bool → int

  - No. It gets rightfully rejected. Term reaches a state that goes wrong as it applies **+** on a value of the wrong type (**y** is **bool**, **+** is defined on **ints**)

- Is this a valid type?

  Nil |- λx: bool.λy: int. if x then y else y+1 :
  bool → int → int

Examples

- Can we deduce the type of this term?

λf. λx. if x=1 then x else (f (f (x-1))) : ?

$$\frac{\Gamma \mid\!- E_1 : \text{int} \qquad \Gamma \mid\!- E_2 : \text{int}}{\Gamma \mid\!- E_1 = E_2 : \text{bool}}$$

$$\frac{\Gamma \mid\!- E_1 : \text{int} \qquad \Gamma \mid\!- E_2 : \text{int}}{\Gamma \mid\!- E_1 + E_2 : \text{int}}$$

$$\frac{\Gamma \mid\!- b : \text{bool} \qquad \Gamma \mid\!- E_1 : \tau \qquad \Gamma \mid\!- E_2 : \tau}{\Gamma \mid\!- \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau}$$

Examples

- Can we deduce the type of this term?

foldl =

λf.λx.λy.if x=() then y else (foldl f (cdr x) (f y (car x))):

$$\frac{\Gamma \mid\!\text{-}\ E : \text{list}\ \tau}{\Gamma \mid\!\text{-}\ (\text{car}\ E) : \tau} \qquad \frac{\Gamma \mid\!\text{-}\ E : \text{list}\ \tau}{\Gamma \mid\!\text{-}\ (\text{cdr}\ E) : \text{list}\ \tau}$$

Examples

- How about this

(λx. x (λy. y) (x 1)) (λz. z) : ?

- **x** cannot have two "different" types
  - **(x 1)** demands **int → ?**
  - **(x (λy. y))** demands **($\tau \to \tau$) → ?**
- Program does not reach a "stuck state" but is nevertheless rejected. A sound type system typically rejects some correct programs

The End