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

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: \((\lambda x. \lambda y. \lambda z. x \, z \, (y \, z)) \, (\lambda u. \, u) \, (\lambda v. \, v) \to_\beta \)
\((\lambda y. \lambda z. (\lambda u. \, u) \, z \, (y \, z)) \, (\lambda v. \, v) \to_\beta \)
\((\lambda z. (\lambda u. \, u) \, z \, ((\lambda v. \, v) \, z)) \to_\beta (\lambda z. \, z \, ((\lambda v. \, v) \, z)) \to_\beta \lambda z. \, z \, z\)

Path 2: \((\lambda x. \lambda y. \lambda z. x \, z \, (y \, z)) \, (\lambda u. \, u) \, (\lambda v. \, v) \to_\beta \)
\((\lambda y. \lambda z. (\lambda u. \, u) \, z \, (y \, z)) \, (\lambda v. \, v) \to_\beta \)
\((\lambda y. \lambda z. \, z \, (y \, z)) \, (\lambda v. \, v) \to_\beta (\lambda z. \, z \, ((\lambda v. \, v) \, z)) \to_\beta \lambda z. \, z \, z\)
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

![Diagram of reduction strategy with redexes labeled 1, 2, and 3]
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 \((\lambda x. x x) \ (\lambda y. y) \ (\lambda z. z)\)\)

- 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 \((\lambda x. \ x \ x) \ (\lambda x. \ x \ x)\). What happens when we apply \(\beta\)-reduction to this expression?

\[
(\lambda x. \ x \ x) \ (\lambda x. \ x \ x) \Rightarrow \beta \ (\lambda x. \ x \ x) \ (\lambda x. \ x \ x) \Rightarrow^* ...
\]

Then look at \((\lambda x. \lambda y. \ y) \ ((\lambda x. \ x \ x) \ (\lambda x. \ x \ x)) \ z\)

- 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 $\lambda$-expressions $M$, $P$ and $Q$, if $M \rightarrow^* P$ and $M \rightarrow^* Q$, then there must exist an expression $R$ such that $P \rightarrow^* R$ and $Q \rightarrow^* R$
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?
Exercises

- Evaluate \((\lambda x. \lambda y. \ x \ y) \ ((\lambda z. \ z) \ w)\)
- Using applicative order reduction

- Using normal order reduction
Exercise

Let $S = \lambda xyz. x z (y z)$ and let $I = \lambda x. x$

Evaluate $S I I I I$ using applicative order
Exercise

Let $S = \lambda xyz. \ x \ z \ (y \ z)$ and let $I = \lambda x. \ x$

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
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 $\lambda$ terms, e.g., `true=\lambda x.\lambda y. x`)
- `0`, `iszero`, `pred`, `succ`

**Reduction rules:**
- `if true M N \rightarrow_\delta M`
- `if false M N \rightarrow_\delta N`
- `iszero 0 \rightarrow_\delta true`
- `iszero (succ^k 0) \rightarrow_\delta false, k>0`
- `iszero (pred^k 0) \rightarrow_\delta false, k>0`
- `succ (pred M) \rightarrow_\delta M`
- `pred (succ M) \rightarrow_\delta M`
## From an Applied Lambda Calculus to a Functional Language

<table>
<thead>
<tr>
<th>Construct</th>
<th>Applied $\lambda$-Calculus</th>
<th>A Language (ML)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
<td>$x$</td>
<td>$x$</td>
</tr>
<tr>
<td>Constant</td>
<td>$c$</td>
<td>$c$</td>
</tr>
<tr>
<td>Application</td>
<td>$M ; N$</td>
<td>$M ; N$</td>
</tr>
<tr>
<td>Abstraction</td>
<td>$\lambda x. M$</td>
<td>$\text{fun } x \Rightarrow M$</td>
</tr>
<tr>
<td>Integer</td>
<td>$\text{succ}^k 0, ; k&gt;0$</td>
<td>$k$</td>
</tr>
<tr>
<td></td>
<td>$\text{pred}^k 0, ; k&gt;0$</td>
<td>$-k$</td>
</tr>
<tr>
<td>Conditional</td>
<td>$\text{if } P ; M ; N$</td>
<td>$\text{if } P ; \text{then } M ; \text{else } N$</td>
</tr>
<tr>
<td>Let</td>
<td>$(\lambda x. M) ; N$</td>
<td>$\text{let val } x = N ; \text{in } M ; \text{end}$</td>
</tr>
</tbody>
</table>
The Fixed-Point Operator

- One more constant and one more rule:
  \[ \text{fix} \quad \text{fix } M \rightarrow^{\delta} M (\text{fix } M) \]
  \[ M(M(M(\ldots ))) \]

- Needed to define recursive functions:
  \[ \text{plus } x \text{ } y = \begin{cases} y & \text{if } x = 0 \\ \text{plus} \ (\text{pred } x) \ (\text{succ } y) & \text{otherwise} \end{cases} \]

- Therefore, we need:
  \[ \text{plus} = \lambda x. \lambda y. \text{if } (\text{iszero } x) \ y \ (\text{plus } (\text{pred } x) \ (\text{succ } y)) \]
The Fixed-Point Operator

But how do we define \textbf{plus}?

Define \textbf{plus} = \textbf{fix } M, where

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

\[
\left(\text{fix } M\right) \times y = \begin{cases}
  y & \text{if } x \text{ is 0} \\
  (\text{fix } M) \ (\text{pred } x) \ (\text{succ } y) & \text{otherwise}
\end{cases}
\]

We must show that

\[ \text{fix } M =^{\delta \beta} \]

\[ \lambda x. \lambda y. \text{if} \ (\text{iszero } x) \ y \ ((\text{fix } M) \ (\text{pred } x) \ (\text{succ } y)) \]
plus = fix M

\[
\begin{align*}
(1) \quad & \text{fix } M \rightarrow \mu M (\text{fix } M) \\
(2) \quad & \mu = \lambda f. \lambda x. \lambda y. \text{if } (\text{iszero } x) y (f (\text{pred } x) (\text{succ } y))
\end{align*}
\]

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

\[
(fix M) a b \rightarrow M (fix M) a b =
\]

\[
(\lambda f. \lambda x. \lambda y. \text{if } (\text{iszero } x) y (f (\text{pred } x) (\text{succ } y))) (\text{fix } M) a b
\]

\[
\rightarrow (\lambda x. \lambda y. \text{if } (\text{iszero } x) y (\text{if } (\text{iszero } a) b ((\text{fix } M) (\text{pred } a) (\text{succ } b))) (\text{fix } M) a b
\]

\[
\rightarrow (\lambda x. \lambda y. \text{if } (\text{iszero } x) y (\text{if } (\text{iszero } a) b ((\text{fix } M) (\text{pred } a) (\text{succ } b))) (\text{fix } M) a b
\]
The Fixed-Point Operator

We have to show

$$\text{fix } M = \delta\beta \lambda x.\lambda y. \text{if } (\text{iszero } x) y ((\text{fix } M) (\text{pred } x) (\text{succ } y))$$

$$\text{fix } M = \delta M (\text{fix } M) =$$

$$\lambda f. \lambda x.\lambda y. \text{if } (\text{iszero } x) y (f (\text{pred } x) (\text{succ } y))) (\text{fix } M) = \beta \lambda x.\lambda y. \text{if } (\text{iszero } x) y ((\text{fix } M) (\text{pred } x) (\text{succ } y))$$
The Fixed-Point Operator

Define \textbf{times} = \\
\text{fix} (\lambda f. \lambda x. \lambda y. \text{if} \ (\text{iszero} \ (\text{pred} \ x)) \ y \ (\text{plus} \ y \ (f \ (\text{pred} \ x) \ y)))

Exercise: define \textbf{factorial} = ?
The Y Combinator

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

\[ Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x)) \]

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

\[
Y M = (\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))) M \rightarrow^\beta (\lambda x. M (x x)) (\lambda x. M (x x)) \rightarrow^\beta
\]

\[
M ((\lambda x. M (x x)) (\lambda x. M (x x))) = M (Y M)
\]
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
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

(Variable)

\[
\frac{x: \tau \in \Gamma}{\Gamma |- x : \tau}
\]

(Application)

\[
\frac{\Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma}{\Gamma |- (E_1 \ E_2) : \tau}
\]

(binding)

\[
\frac{\Gamma, x: \sigma |- E_1 : \tau}{\Gamma |- (\lambda x: \sigma. \ E_1) : \sigma \rightarrow \tau}
\]

looks up the type of \( x \) in environment \( \Gamma \)

binding: augments environment \( \Gamma \) with binding of \( x \) to type \( \sigma \)
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
  - \( \tau ::= b \mid \tau \to \tau \)
  - A type is a basic type \( b \) (we will only consider \text{int} and \text{bool}, for simplicity), or a function type

- Examples
  - \text{int}
  - \text{bool} \to (\text{int} \to \text{int}) \quad // \quad \text{is right-associative}, thus can write just \text{bool} \to \text{int} \to \text{int}

- Syntax of terms:
  - \( E ::= x \mid (\lambda x: \tau. \ E_1) \mid (\ E_1 \ E_2) \)
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 \( \Gamma \) entails that \( E \) has type \( \tau \)

Type judgment \( \Gamma |- E_1 : \sigma \rightarrow \tau \) \( \Gamma |- E_2 : \sigma \) \( \Gamma |- (E_1 E_2) : \tau \)
Semantics

Variable

\[ \frac{x: \tau \in \Gamma}{\Gamma |- x : \tau} \]  
(Variable)

Application

\[ \frac{\Gamma |- E_1 : \sigma \rightarrow \tau \quad \Gamma |- E_2 : \sigma}{\Gamma |- (E_1 \ E_2) : \tau} \]  
(Application)

Abstraction

\[ \frac{\Gamma, x: \sigma |- E_1 : \tau}{\Gamma |- (\lambda x: \sigma. \ E_1) : \sigma \rightarrow \tau} \]  
(Abstraction)

binding: augments environment \( \Gamma \) with binding of \( x \) to type \( \sigma \)

looks up the type of \( x \) in environment \( \Gamma \)
Examples

- Deduce the type for

\[ \lambda x: \text{int} . \lambda y: \text{bool} . \ x \] in the nil environment
Extensions

Γ |- c : int

Γ |- E₁ : int    Γ |- E₂ : int
Γ |- E₁ + E₂ : int

Γ |- E₁ = E₂ : bool

Γ |- b : bool    Γ |- E₁ : τ    Γ |- E₂ : τ
Γ |- if b then E₁ else E₂ : τ
Examples

- Is this a valid type?
  
  \[ \text{Nil} |- \lambda x : \text{int}. \lambda y : \text{bool}. \ x+y : \text{int} \rightarrow \text{bool} \rightarrow \text{int} \]

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

- Is this a valid type?
  
  \[ \text{Nil} |- \lambda x : \text{bool}. \lambda y : \text{int}. \ \text{if } x \ \text{then } y \ \text{else } y+1 : \\text{bool} \rightarrow \text{int} \rightarrow \text{int} \]
Examples

Can we deduce the type of this term?

\( \lambda f. \lambda x. \text{if } x=1 \text{ then } x \text{ else } (f(f(x-1))) : ? \)

\[ \begin{align*}
\Gamma |- E_1 : \text{int} & \quad \Gamma |- E_2 : \text{int} \\
\quad & \quad \quad \quad \Gamma |- E_1=E_2 : \text{bool} \\
\quad & \quad \quad \quad \Gamma |- E_1 : \text{int} \\
\quad & \quad \quad \quad \Gamma |- E_2 : \text{int} \\
\quad & \quad \quad \quad \Gamma |- E_1+E_2 : \text{int} \\
\quad & \quad \quad \quad \Gamma |- b : \text{bool} \\
\quad & \quad \quad \quad \Gamma |- E_1 : \tau \\
\quad & \quad \quad \quad \Gamma |- E_2 : \tau \\
\Gamma |- \text{if } b \text{ then } E_1 \text{ else } E_2 : \tau
\end{align*} \]
Examples

- Can we deduce the type of this term?

\[ \text{foldl} = \]

\[ \lambda f. \lambda x. \lambda y. \text{if } x = () \text{ then } y \text{ else } (\text{foldl } f (\text{cdr } x) (f \ y \ (\text{car } x))) : \]

\[ \Gamma |- E : \text{list } \tau \]

\[ \Gamma |- (\text{car } E) : \tau \]

\[ \Gamma |- (\text{cdr } E) : \text{list } \tau \]
Examples

How about this

\((\lambda x. \ x \ (\lambda y. \ y) \ (x \ 1)) \ (\lambda z. \ z) : ?\)

- **x** cannot have two “different” types
  - \((x \ 1)\) demands \texttt{int} \to ?
  - \((x \ (\lambda y. \ y))\) demands \((\tau \to \tau) \to ?\)

- Program does not reach a “stuck state” but is nevertheless rejected. A sound type system typically rejects some correct programs
The End