CSCI 740 - Programming Language Theory

Lecture 6
The λ-calculus
Instructor: Hossein Hojjat

September 11, 2017
Many features in programming languages are not essential: they are for convenience

<table>
<thead>
<tr>
<th>Feature</th>
<th>Code Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-argument functions</td>
<td><code>def f(x, y, z)</code></td>
</tr>
<tr>
<td>Use Currying</td>
<td></td>
</tr>
<tr>
<td>Loops</td>
<td><code>while (x == y)</code></td>
</tr>
<tr>
<td>Use recursion</td>
<td><code>{...}</code></td>
</tr>
<tr>
<td>Side effects</td>
<td><code>var (x = 1)</code></td>
</tr>
<tr>
<td>Use functional programming</td>
<td></td>
</tr>
</tbody>
</table>

What languages features are really necessary?
Core Language

- Goal: Come up with a “core” language
  - As small as possible
  - Still Turing complete
- Gives an opportunity of studying important language features
Core Language

- Goal: Come up with a “core” language
  - As small as possible
  - Still Turing complete
- Gives an opportunity of studying important language features

\( \lambda \)-calculus

- Peter J Landin (1960s):
  \( \lambda \)-calculus can be used to model a complex programming language
- A programming language is nothing more than \( \lambda \)-calculus plus some syntactic sugar
  - “A Correspondence Between ALGOL 60 and Church’s Lambda-Notation”(1965)
  - “The Next 700 Programming Languages”(1966)
- \( \lambda \)-calculus plays a similar role for PL research as Turing machines do for computability
Lambda Expressions

- A $\lambda$-calculus expression is defined as

\[
e ::= x \quad \text{variable} \\
| \quad \lambda x.e \quad \text{abstraction} \\
| \quad e\ e \quad \text{application}
\]

Application

Application is left associative
\[
(e_1\ e_2\ e_3\ e_4) \equiv (((e_1\ e_2)\ e_3)\ e_4)
\]

Abstraction

- Scope of the dot in an abstraction extends as far to the right as possible

\[
\lambda x. e_1\ e_2\ e_3 \quad \text{means} \quad (\lambda x. (e_1\ e_2\ e_3)) \quad \text{and not} \quad ((\lambda x. e_1\ e_2)\ e_3)
\]
Examples of Lambda Expressions

- The identity function

\[ \lambda x. x \]
Examples of Lambda Expressions

- The identity function

\[ \lambda x. x \]
Examples of Lambda Expressions

• The identity function

\[ \lambda x \cdot x \]

• Function that takes two other functions and produces their composition
Examples of Lambda Expressions

- The identity function
  \[ \lambda x. x \]

- Function that takes two other functions and produces their composition
  \[ \lambda f. \lambda g. (f \ g) \]
Examples of Lambda Expressions

- The identity function
  \[ \lambda x.x \]

- Function that takes two other functions and produces their composition
  \[ \lambda f. \lambda g. (f \ g) \]

- Function that flips the order of two arguments that are passed to a function
Examples of Lambda Expressions

- The identity function

$$\lambda x . x$$

- Function that takes two other functions and produces their composition

$$\lambda f . \lambda g . (f \ g)$$

- Function that flips the order of two arguments that are passed to a function

$$\lambda f . \lambda x . \lambda y . (f \ y \ x)$$
• \( \lambda \)-calculus provides no built-in support for multi-argument functions
• Do we need an extension \( \lambda (x_1 \cdots x_n) \, e \) to support multiple arguments?
Multiple Arguments

- $\lambda$-calculus provides no built-in support for multi-argument functions
- Do we need an extension $\lambda(x_1 \cdots x_n)\ e$ to support multiple arguments?
- **Currying**: reducing a function with multiple arguments to functions with a single argument
  
  $$(\lambda(x_1 \cdots x_n)\ e) \Rightarrow (\lambda x_1(\lambda \cdots (\lambda x_n e) \cdots))$$

**Example.**

- Consider the function $\lambda(x, y).(x\ y)$
- Curried form: $f = \lambda x.\lambda y.(x\ y)$
- $g = (f\ \lambda x.x)$ is a function that takes a single argument and applies identity to it
Static Scoping

- Scope of a variable:
  portion of program where the identifier is accessible
- An abstraction $\lambda x. e$ binds the variable $x$ in $e$
  - $e$ is the scope of $x$
  - $x$ is bound in $\lambda x. e$
- $\lambda$-calculus uses static scoping
  - Scope of variable is fixed at compile-time to the smallest block containing the variable declaration
- $(\lambda x. x (\lambda x. x)) \ z$
  - Rightmost $x$ refers to the second binding
  - Takes its argument and applies it to the identity function
Free and Bound Variables

\[ \lambda \ x \ . \ e \]

- Occurrences of \( x \) in the body \( e \) are bound
- Nonbound variable occurrences are called free

**Free Variables**

\[
\begin{align*}
FV(x) &= \{x\} \\
FV(e_1 \ e_2) &= FV(e_1) \cup FV(e_2) \\
FV(\lambda x. e) &= FV(e) - \{x\}
\end{align*}
\]
Free and Bound Variables

\[ \lambda x. (x (\lambda y. y z) x) \ y \]
Free and Bound Variables

\[ \lambda x. (x (\lambda y. y \ z) \ x) \ y \]

bound
Free and Bound Variables

\[ \lambda x. (x \ (\lambda y. y \ z) \ x) \ y \]

• An expression is \textit{closed} if all variables are bound
• An expression is \textit{open} if it is not closed
Free and Bound Variables

\[ \lambda x. (x (\lambda y. y z) x) y \]

bound  boundfree
Free and Bound Variables

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

bound  boundfree  bound
Free and Bound Variables

• An expression is **closed** if all variables are bound
• An expression is **open** if it is not closed
Like in any language with statically nested scoping, we need to worry about variable shadowing.

Shadowing: a variable declared within a scope has the same name as a variable declared in an outer scope.

\[ \lambda x. \ x \ (\lambda x. x) \ x \]
Renaming Bound Variables

- **α-equivalence**: λ-expressions that can be obtained from one another by renaming the bound variables are identical.

- **Example**: \( \lambda x.x \) is identical to \( \lambda y.y \) and to \( \lambda z.z \).

- **Intuition**: Change the name of a formal argument and of all its occurrences in the function body: function behavior does not change.

- **α-conversion**: renaming the bound variables in the expression.

- **Always try to rename bound variables so that they are all unique**
  - Write \( \lambda x.x \ (\lambda y.y) \ x \) instead of \( \lambda x.x \ (\lambda x.x) \ x \)
  - Makes it easy to see the scope of bindings.
Substitution

$e[x \mapsto e_1]$  
\(\lambda\)-expression obtained by replacing each free occurrence of the variable \(x\) in \(e\) by the \(\lambda\)-expression \(e_1\)

- **Valid** Substitution: no free variable in \(e_1\) becomes bound as a result of the substitution \(e[x \mapsto e_1]\)
- **Invalid** Substitution: involves a variable capture or a name clash

**Example:**

- an invalid substitution:

\[
(\lambda x . x \ y)[y \mapsto x] ? = \lambda x . x \ x
\]

- first function: applies argument to \(y\)
- second function: identity
Substitution

- Substitution is defined by cases on $e[x \mapsto e']$

**Variable**

$$y[x \mapsto e'] = \begin{cases} e' & \text{if } x = y \\ y & \text{otherwise} \end{cases}$$

**Application**

$$(e_1 \ e_2)[x \mapsto e'] = (e_1[x \mapsto e'] \ e_2[x \mapsto e'])$$

**Abstraction**

$$(\lambda y.e_1)[x \mapsto e'] = \begin{cases} \lambda y.e_1 & \text{if } x = y \\ \lambda z.((e_1[y \mapsto z])[x \mapsto e']) & \text{otherwise} \end{cases}$$

where $z \not\in (\text{FV}(e_1) \cup \text{FV}(e') \cup \{x\})$
In λ-calculus there is one computation rule called β-reduction:

\[(\lambda x. e) \ e_1 \rightarrow_\beta e[x \mapsto e_1]\]

- A β-redex (reducible expression) is a term of form \((\lambda x. e) \ e_1\)
- λ-definable functions coincide with the Turing-computable functions (λ-calculus is Turing-complete)
- Any possible computation can be defined in terms of β-reduction rule
Apply the $\beta$-reductions in the following expression

$$((\lambda x.\lambda y.x) \ y) \ z$$
β-reduction Exercise

- Apply the β-reductions in the following expression

\[((\lambda x.\lambda y. x) \; y) \; z\]

\[\rightarrow_\beta \; ((\lambda y. x)[x \mapsto y]) \; z\] substitute \(y\) for \(x\) in the body of \(\lambda y.x\)

\[\rightarrow_\alpha \; ((\lambda y'.x)[x \mapsto y]) \; z\] after \(\alpha\)-conversion

\[\rightarrow \; (\lambda y'.y) \; z\] first \(\beta\)-reduction complete

\[\rightarrow_\beta \; y[y' \mapsto z]\] substitute \(z\) for \(y'\) in \(y\)

\[\rightarrow \; y\] second \(\beta\)-reduction complete
Apply the $\beta$-reductions in the following expression

$$(\lambda x.x\ x)(\lambda x.x\ x)$$
Apply the $\beta$-reductions in the following expression

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

Each time you beta-reduce you get the same expression back

$\lambda$-calculus is equivalent to Turing machine

Turing machine may fail to halt
\( \eta \)-conversion

- \( \eta \)-reduction is useful to eliminate redundant \( \lambda \)-abstractions

\[
\lambda x. (e\ x) \rightarrow_{\eta} e \quad \text{if } x \not\in \text{FV}(e)
\]

- Motivation for \( \eta \)-conversion:

- \( \lambda x. (e\ x) \) and \( e \) behave identically as functions

\[
(\lambda x. (e\ x))\ u \rightarrow_{\beta} (e\ u) \quad \text{if } x \not\in \text{FV}(e)
\]

**Example:**

\[
(\lambda x. (\lambda y.x\ y)\ (y\ w))
\]

\( \rightarrow_{\alpha} \)

\[
(\lambda x. (\lambda y'.x\ y')\ (y\ w))
\]

\( \rightarrow_{\beta} \)

\[
\lambda y'.((y\ w)\ y')
\]

\( \rightarrow_{\eta} \)

\[
(y\ w)
\]
Computing with $\lambda$-calculus

- $\lambda$-calculus is a **core** programming language: it is Turing complete
- How can we possibly compute with the $\lambda$-calculus when we have no data to manipulate?
  1. no numbers
  2. no data-structures
  3. no control structures (if-then-else, loops)
Computing with $\lambda$-calculus

- $\lambda$-calculus is a **core** programming language: it is Turing complete
- How can we possibly compute with the $\lambda$-calculus when we have no data to manipulate?
  1. no numbers
  2. no data-structures
  3. no control structures (if-then-else, loops)

**Church Encoding:**
Representation of data and operators in the lambda calculus

- Not intended as a practical implementation of primitive data types
- It shows that other primitive data types are not required to represent any calculation
• We need to define functions TRUE, FALSE, AND, NOT, IF that behave as expected
• For example:

\[
\begin{align*}
\text{AND TRUE FALSE} & = \text{FALSE} \\
\text{NOT FALSE} & = \text{TRUE} \\
\text{IF TRUE } e_1 e_2 & = e_1 \\
\text{IF FALSE } e_1 e_2 & = e_2
\end{align*}
\]
We can define TRUE and FALSE as e.g. following:

- TRUE returns the first
- FALSE returns the second
**Encoding Booleans**

- Conditional statement takes three arguments \( b, e_t, e_f \) where:
  - \( b \) is a Boolean value and \( e_t, e_f \) are arbitrary \( \lambda \)-expressions

\[
\text{IF} = \lambda b. \lambda e_t. \lambda e_f. \begin{cases} 
e_t & \text{if } b = \text{true} \\ e_f & \text{if } b = \text{false} \end{cases}
\]
• Conditional statement takes three arguments $b, e_t, e_f$ where: $b$ is a Boolean value and $e_t$, $e_f$ are arbitrary $\lambda$-expressions

\[
\text{IF} = \lambda b.\lambda e_t.\lambda e_f. \begin{cases} 
  e_t & \text{if } b = \text{true} \\
  e_f & \text{if } b = \text{false}
\end{cases}
\]

• Since $\text{TRUE } e_t e_f \rightarrow^\beta e_t$ and $\text{FALSE } e_t e_f \rightarrow^\beta e_f$ we define

\[
\text{IF} \triangleq \lambda b.\lambda e_t.\lambda e_f. b e_t e_f
\]
Encoding Booleans

\[
\text{NOT} \triangleq \lambda b. b \text{ FALSE TRUE}
\]

\[
\text{AND} \triangleq \lambda b_1. \lambda b_2. b_1 \ b_2 \text{ FALSE}
\]

\[
\text{OR} \triangleq \lambda b_1. \lambda b_2. b_1 \text{ TRUE } b_2
\]
Church Numerals

- Encode numbers with two-argument functions
- “Number $i$” composes the first argument $i$ times, starting with the second argument
- $z$ stands for “zero” and $s$ for “successor” (think unary)

$c_0 \equiv \lambda s.\lambda z.z$
$c_1 \equiv \lambda s.\lambda z.s \; z$
$c_2 \equiv \lambda s.\lambda z.s \; (s \; z)$
$c_3 \equiv \lambda s.\lambda z.s \; (s \; (s \; z))$

...
Church Numerals

\[ \begin{align*}
  c_0 & \quad \lambda s. \lambda z. z \\
  c_1 & \quad \lambda s. \lambda z. s \ z \\
  c_2 & \quad \lambda s. \lambda z. s \ (s \ z) \\
  c_3 & \quad \lambda s. \lambda z. s \ (s \ (s \ z)) \\
  \cdots & \\
  \text{SUCC} & \quad \lambda n. \lambda s. \lambda z. s \ (n \ s \ z)
\end{align*} \]

- successor: take “a number” and return “a number” that (when called) applies \( s \) one more time