CS 476 – Programming Language Design

William Mansky
Functional Programming

• *Functions* are the basic unit of computation
• Functions are values! (“first-class functions”)
  — Functions can take functions as arguments
• No mutable variables*
• Usually contrasted with imperative languages
• Examples: F#, OCaml, Lisp, Haskell, lambda-expressions
The First Functional Language

- Functional languages are older than computers!
- The *lambda calculus* was invented as a mathematical model of “what can be computed”, and it consists entirely of functions.

<table>
<thead>
<tr>
<th>Math notation</th>
<th>Lambda calculus</th>
<th>OCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = x + 1$</td>
<td>$\lambda x. x + 1$</td>
<td><code>fun x -&gt; x + 1</code></td>
</tr>
<tr>
<td>$g(x, y) = y$</td>
<td>$\lambda x. (\lambda y. y)$</td>
<td><code>fun x y -&gt; y</code></td>
</tr>
</tbody>
</table>

Currying: to make a two-argument function, return a function!
The First Functional Language

• Functional languages are older than computers!
• The *lambda calculus* was invented as a mathematical model of “what can be computed”, and it consists entirely of functions

<table>
<thead>
<tr>
<th>Math notation</th>
<th>Lambda calculus</th>
<th>OCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) = x + 1 )</td>
<td>( \lambda x. x + 1 )</td>
<td>( \text{fun } x \rightarrow x + 1 )</td>
</tr>
<tr>
<td>( g(x, y) = y )</td>
<td>( \lambda x. \lambda y. y )</td>
<td>( \text{fun } x \ y \rightarrow y )</td>
</tr>
</tbody>
</table>

\[ f(1) = 1 + 1 = 2 \]
The First Functional Language

• Functional languages are older than computers!
• The lambda calculus was invented as a mathematical model of “what can be computed”, and it consists entirely of functions

<table>
<thead>
<tr>
<th>Math notation</th>
<th>Lambda calculus</th>
<th>OCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(x) = x + 1$</td>
<td>$\lambda x. x + 1$</td>
<td><code>fun x -&gt; x + 1</code></td>
</tr>
<tr>
<td>$g(x, y) = y$</td>
<td>$\lambda x. \lambda y. y$</td>
<td><code>fun x y -&gt; y</code></td>
</tr>
</tbody>
</table>

$f(1) = 1 + 1 = 2$  \hspace{1cm} (fun x -> x + 1) 1 = 1 + 1 = 2
The First Functional Language

• Functional languages are older than computers!
• The \textit{lambda calculus} was invented as a mathematical model of “what can be computed”, and it consists entirely of functions

<table>
<thead>
<tr>
<th>Math notation</th>
<th>Lambda calculus</th>
<th>OCaml</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(x) = x + 1 )</td>
<td>( \lambda x. x + 1 )</td>
<td>fun x -&gt; x + 1</td>
</tr>
<tr>
<td>( g(x, y) = y )</td>
<td>( \lambda x. \lambda y. y )</td>
<td>fun x y -&gt; y</td>
</tr>
</tbody>
</table>

\[ f(1) = 1 + 1 = 2 \quad (\lambda x. x + 1) 1 = 1 + 1 = 2 \]
Lambda Calculus Basics

• Functions are values, and functions are the only values!
• No declarations, no lets, just anonymous functions
• A function has two parts: $\lambda x. B$
  - argument name
  - body (any term, can contain $x$)
  - “bound variable”

• Functions can be applied to other terms (also functions)
• Application is evaluated by replacing the bound variable with the argument in the body
  $$(\lambda x. (\lambda y. x)) z$$
Lambda Calculus Basics

• Functions are values, and functions are the only values!
• No declarations, no lets, just anonymous functions
• A function has two parts: $\lambda x. B$

  argument name    body (any term, can contain $x$)
  “bound variable”

• Functions can be applied to other terms
• Application is evaluated by replacing the bound variable with
  the argument in the body

  $(\lambda x. (\lambda y. x)) \, z \rightarrow (\lambda y. x) \text{ with } x \text{ replaced by } z$
Lambda Calculus Basics

• Functions are values, and functions are the only values!
• No declarations, no lets, just anonymous functions
• A function has two parts: \( \lambda x. B \)
  
  argument name  body (any term, can contain \( x \))  “bound variable”

• Functions can be applied to other terms
• Application is evaluated by replacing the bound variable with the argument in the body

\[
(\lambda x. (\lambda y. x)) z \rightarrow \lambda y. z
\]
Variable Binding

int f(int x){ return x + 1; }

int x = 5;
f(x + 2);
Lambda Calculus: Binding and Scope

• $\lambda x. B$ binds $x$ in $B$

• In other words, wherever $x$ appears in $B$, it means “the argument passed to this function”

• Each variable refers to the innermost $\lambda$-binding around it

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

• A variable that is not bound is free, like $y$ in $\lambda x. y x$
Lambda Calculus: Renaming

• The name of the argument to a function doesn’t really matter
• \( \lambda x. x \) is the same as \( \lambda y. y \)
• We can always rename a bound variable

\[
\lambda x. (\lambda x. x (\lambda x. x x)) x
\]
Lambda Calculus: Renaming

• The name of the argument to a function doesn’t really matter
• \( \lambda x. x \) is the same as \( \lambda y. y \)
• We can always rename a bound variable

\[
\lambda x. (\lambda x. x (\lambda y. y y)) x
\]
Lambda Calculus: Renaming

- The name of the argument to a function doesn’t really matter
- $\lambda x. x$ is the same as $\lambda y. y$
- We can always rename a bound variable

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

- Renaming (sometimes called “alpha-conversion”) shouldn’t change the behavior of a function
Lambda Calculus: Substitution

• In general, $(\lambda x. l) \ l_2$ evaluates to $[x \mapsto l_2]l$ ("$l$ with $l_2$ substituted for $x$"")

• $(\lambda x. x) \ z$ evaluates to
Lambda Calculus: Substitution

- In general, $(\lambda x. l) l_2$ evaluates to $[x \mapsto l_2]l$ (“$l$ with $l_2$ substituted for $x$”)
- $(\lambda x. x) z$ evaluates to $z$
- $(\lambda x. \lambda y. x) z$ evaluates to
Lambda Calculus: Substitution

• In general, \((\lambda x. l) \, l_2\) evaluates to \([x \mapsto l_2]l\) ("\(l\) with \(l_2\) substituted for \(x\")

• \((\lambda x. x) \, z\) evaluates to \(z\)

• \((\lambda x. \lambda y. x) \, z\) evaluates to \(\lambda y. z\)

• \((\lambda x. \lambda y. y) \, z\) evaluates to
Lambda Calculus: Substitution

• In general, \((\lambda x. l) \ l_2\) evaluates to \([x \mapsto l_2]l\) ("\(l\) with \(l_2\) substituted for \(x\)")

• \((\lambda x. x) \ z\) evaluates to \(z\)
• \((\lambda x. \lambda y. x) \ z\) evaluates to \(\lambda y. z\)
• \((\lambda x. \lambda y. y) \ z\) evaluates to \(\lambda y. y\)
• \((\lambda x. x (\lambda y. y)) \ z\) evaluates to
Lambda Calculus: Substitution

• In general, \((\lambda x. l) \ l_2\) evaluates to \([x \mapsto l_2]l\) ("\(l\) with \(l_2\) substituted for \(x\))

• \((\lambda x. x) \ z\) evaluates to \(z\)
• \((\lambda x. \lambda y. x) \ z\) evaluates to \(\lambda y. z\)
• \((\lambda x. \lambda y. y) \ z\) evaluates to \(\lambda y. y\)
• \((\lambda x. x (\lambda y. y)) \ z\) evaluates to \(z \ (\lambda y. y)\)
• \((\lambda x. x (\lambda x. x)) \ z\) evaluates to
Lambda Calculus: Substitution

- In general, \((\lambda x. l) l_2\) evaluates to \([x \mapsto l_2]l\) ("\(l\) with \(l_2\) substituted for \(x\")

- \((\lambda x. x) z\) evaluates to \(z\)
- \((\lambda x. \lambda y. x) z\) evaluates to \(\lambda y. z\)
- \((\lambda x. \lambda y. y) z\) evaluates to \(\lambda y. y\)
- \((\lambda x. x (\lambda y. y)) z\) evaluates to \(z (\lambda y. y)\)
- \((\lambda x. x (\lambda x. x)) z\) evaluates to \(z (\lambda x. x)\)
Lambda Calculus: Substitution

How do we define substitution? \([x \mapsto l_2]l\)

match \(l\) with
| \(y\) \(\rightarrow\)
| \((l_a \ l_b)\) \(\rightarrow\)
| \((\lambda y. \ b)\) \(\rightarrow\)
Lambda Calculus: Substitution

How do we define substitution? \([x \mapsto l_2]l\)

match \(l\) with
| \(y\) \(\rightarrow\) if \(y = x\) then \(l_2\) else \(y\)
| \((l_a\ l_b)\) \(\rightarrow\)
| \((\lambda y.\ b)\) \(\rightarrow\)
Lambda Calculus: Substitution

How do we define substitution? \([x \mapsto l_2]l\)

match \(l\) with
| \(y\) → if \(y = x\) then \(l_2\) else \(y\)
| \((l_a \ l_b)\) → \(([x \mapsto l_2]l_a) (\[x \mapsto l_2]l_b)\)
| \((\lambda y. \ b)\) →
Lambda Calculus: Substitution

How do we define substitution? \([x \mapsto l_2]l\)

match \(l\) with
  \(y \rightarrow \text{if } y = x \text{ then } l_2 \text{ else } y\)
  \((l_a \ l_b) \rightarrow ([x \mapsto l_2]l_a) ([x \mapsto l_2]l_b)\)
  \((\lambda y. b) \rightarrow \text{if } y = x \text{ then } \lambda y. b \text{ (\(\lambda x. (\lambda x. x \ y))\)} z \rightarrow \lambda x. x \ y\)
Lambda Calculus: Substitution

How do we define substitution? \([x \mapsto l_2]l\)

match \(l\) with
| \(y\) \to \text{if } y = x \text{ then } l_2 \text{ else } y
| \((l_a \ l_b)\) \to ([x \mapsto l_2]l_a) \ ([x \mapsto l_2]l_b)
| \((\lambda y. \ b)\) \to \text{if } y = x \text{ then } \lambda y. \ b \ (\lambda x. (\lambda y. x \ y)) \ z \to \lambda y. z \ y
Lambda Calculus: Substitution

How do we define substitution? \([x \mapsto l_2]l\)

match \(l\) with
| \(y\) \rightarrow \text{if } y = x \text{ then } l_2 \text{ else } y \\
| (l_a \, l_b) \rightarrow ([x \mapsto l_2]l_a) \, ([x \mapsto l_2]l_b) \\
| (\lambda y. b) \rightarrow \text{if } y = x \text{ then } \lambda y. b \text{ else } \lambda y. ([x \mapsto l_2]b) \quad \text{except...} \\
(\lambda x. \lambda y. x \, y) \, y \text{ evaluates to } ?
Lambda Calculus: Substitution

How do we define substitution? \([x \mapsto l_2]l\)

match \(l\) with
| \(y\) → if \(y = x\) then \(l_2\) else \(y\)
| \((l_a \ l_b)\) → \([(x \mapsto l_2]l_a) \ (][x \mapsto l_2]l_b)\)
| \((\lambda y. \ b)\) → if \(y = x\) then \(\lambda y. \ b\) else \(\lambda y. \ ([x \mapsto l_2]b)\)  except...

\((\lambda x. \ \lambda y. \ x \ y)\) \(y\) evaluates to \(\lambda y. \ ([x \mapsto y](x \ y))\)

which is \(\lambda y. \ y\ \ y\) ... but that’s not right! “variable capture”
Lambda Calculus: Substitution

In OCaml:

let f x y = x + y;; (* f = fun x -> fun y -> x + y *)
let y = 5;;
let g = f y;;
(* g = fun y -> y + y would be wrong *)
(* g = fun z -> 5 + z would be right *)
Lambda Calculus: Substitution

How do we define substitution? \([x \mapsto l_2]l\)

match \(l\) with
| \(y\)  \(\rightarrow\) if \(y = x\) then \(l_2\) else \(y\)
| \((l_a \ l_b)\) \(\rightarrow\) \(([x \mapsto l_2]l_a)([x \mapsto l_2]l_b)\)
| \((\lambda y. \ b)\) \(\rightarrow\) if \(y = x\) then \(\lambda y. \ b\) else \(\lambda y. ([x \mapsto l_2]b)\) except...

\((\lambda x. \lambda y. \ x \ y)\) \(y\) is the same as \((\lambda x. \lambda z. \ x \ z)\) \(y\)

which evaluates to \(\lambda z. ([x \mapsto y](x \ z)) = \lambda z. y \ z\)
Lambda Calculus: Substitution

How do we define substitution? \([x \mapsto l_2]l\)

match \(l\) with

<table>
<thead>
<tr>
<th>(y)</th>
<th>(\text{if } y = x \text{ then } l_2 \text{ else } y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>((l_a \ l_b))</td>
<td>(((x \mapsto l_2)l_a) (x \mapsto l_2)l_b)</td>
</tr>
<tr>
<td>((\lambda y. b))</td>
<td>(\text{if } y = x \text{ then } \lambda y. b \text{ else})</td>
</tr>
<tr>
<td></td>
<td>(\text{let } z = \text{fresh } l_2 \text{ in } \lambda z. ([x \mapsto l_2] (\text{rename } y \ z \ b)))</td>
</tr>
</tbody>
</table>

“Capture-avoiding substitution”
Lambda Calculus: Syntax

\[ L ::= \text{<ident>} \mid \lambda \text{<ident>}. L \mid LL \]
Lambda Calculus: Semantics

$L ::= <ident> \mid \lambda <ident>. L \mid LL$

- Functions are values
- Application is evaluated by substitution
Lambda Calculus: Semantics

\[ L ::= <\text{ident}> \mid \lambda<\text{ident}>. L \mid L L \]

- \( \lambda x. l \) is a value
- Application is evaluated by substitution
Lambda Calculus: Semantics

$L ::= <\text{ident}> \mid \lambda<\text{ident}>. L \mid LL$

• $\lambda x. l$ is a value

\[
\frac{l_1 \rightarrow l'_1}{l_1 l_2 \rightarrow l'_1 l_2}
\]

\[
(\lambda x. l) l_2 \rightarrow [x \mapsto l_2] l
\]

• “Call by name”
Lambda Calculus: Semantics

\[ L ::= \text{<ident>} \mid \lambda\text{<ident>}. L \mid LL \]

\[ \frac{l_1 \rightarrow l'_1}{l_1 \; l_2 \rightarrow l'_1 \; l_2} \]

\[ \frac{l_2 \rightarrow l'_2}{\nu \; l_2 \rightarrow \nu \; l'_2} \quad (\lambda x. l) \; \nu \rightarrow [x \leftrightarrow \nu]l \]

• “Call by value”
Call-By-Name vs. Call-By-Value

\((\lambda x. \lambda y. y) \ l\) where \(l\) becomes a value in 10 steps

**Call-by-name:**
\[
\begin{align*}
\rightarrow (\lambda y. y)
\end{align*}
\]

**Call-by-value:**
\[
\begin{align*}
\rightarrow (\lambda x. \lambda y. y) \ l_1 \rightarrow (\lambda x. \lambda y. y) \ l_2 \rightarrow \cdots \rightarrow (\lambda x. \lambda y. y) \ v \\
\rightarrow (\lambda y. y)
\end{align*}
\]
Call-By-Name vs. Call-By-Value

$$(\lambda x. \lambda y. y) \ l \text{ where } l \text{ runs forever}$$

$$(\lambda x. x \ x) \ (\lambda x. x \ x) \ \rightarrow [x \mapsto \lambda x. x \ x](x \ x)$$

which is $$(\lambda x. x \ x) \ (\lambda x. x \ x)!$$

Call-by-name:
$$\rightarrow (\lambda y. y)$$

Call-by-value:
$$\rightarrow (\lambda x. \lambda y. y) \ l \rightarrow (\lambda x. \lambda y. y) \ l \rightarrow \cdots$$
Call-By-Name vs. Call-By-Value

$(\lambda x. ... x ... x ...) \; l$ where $l$ becomes a value in 10 steps

Call-by-name:
\[
\rightarrow ... l ... l ... \rightarrow ... l_1 ... l ... \rightarrow ... v ... l ... \rightarrow ... v ... l_1 ... \rightarrow ...
\]

Call-by-value:
\[
\rightarrow (\lambda x. ... x ... x ...) \; l_1 \rightarrow ... \rightarrow (\lambda x. ... x ... x ...) \; v \rightarrow ... v ... v ...
\]
Lambda Calculus and Computability

What can be computed?

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

“Turing-complete”

Church, 1936

Turing, 1936