Lambda Calculus (PDCS 2)
alpha-renaming, beta reduction, applicative and normal evaluation orders, Church-Rosser theorem, combinators

Carlos Varela
Rennselaer Polytechnic Institute

February 19, 2015
Mathematical Functions

Take the mathematical function:

\[ f(x) = x^2 \]

\( f \) is a function that maps integers to integers:

\[ f: \mathbb{Z} \rightarrow \mathbb{Z} \]

We apply the function \( f \) to numbers in its domain to obtain a number in its range, e.g.:

\[ f(-2) = 4 \]
Given the mathematical functions:
\[ f(x) = x^2, \quad g(x) = x+1 \]

\( f \circ g \) is the composition of \( f \) and \( g \):

\[
(f \circ g)(x) = f(g(x)) = f(x+1) = (x+1)^2 = x^2 + 2x + 1
\]

\[
g \circ f (x) = g(f(x)) = g(x^2) = x^2 + 1
\]

Function composition is therefore not commutative. Function composition can be regarded as a (higher-order) function with the following type:

\[ \circ : (Z \rightarrow Z) \times (Z \rightarrow Z) \rightarrow (Z \rightarrow Z) \]
Lambda Calculus (Church and Kleene 1930’s)

A unified language to manipulate and reason about functions.

Given
\( f(x) = x^2 \)

\( \lambda x. x^2 \)
represents the same \( f \) function, except it is anonymous.

To represent the function evaluation \( f(2) = 4 \),
we use the following \( \lambda \)-calculus syntax:

\( (\lambda x. x^2 \ 2) \Rightarrow 2^2 \Rightarrow 4 \)
Lambda Calculus Syntax and Semantics

The syntax of a $\lambda$-calculus expression is as follows:

$$e ::= v \quad \text{variable}$$
$$| \quad \lambda v. e \quad \text{functional abstraction}$$
$$| \quad (e e) \quad \text{function application}$$

The semantics of a $\lambda$-calculus expression is as follows:

$$(\lambda x. E M) \Rightarrow E\{M/x\}$$

where we alpha-rename the lambda abstraction $E$ if necessary to avoid capturing free variables in $M$. 
Currying

The lambda calculus can only represent functions of one variable. It turns out that one-variable functions are sufficient to represent multiple-variable functions, using a strategy called *currying*.

E.g., given the mathematical function: \( h(x,y) = x+y \)
of type \( h: \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \)

We can represent \( h \) as \( h' \) of type: \( h': \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \)
Such that \( h(x,y) = h'(x)(y) = x+y \)
For example, \( h'(2) = g \), where \( g(y) = 2+y \)

We say that \( h' \) is the *curried* version of \( h \).
Function Composition in Lambda Calculus

S: $\lambda x. (s \, x)$    (Square)
I: $\lambda x. (i \, x)$    (Increment)

C: $\lambda f. \lambda g. \lambda x. (f \, (g \, x))$   (Function Composition)

Recall semantics rule:

$$((\lambda x. E \, M) \Rightarrow E\{M/x\})$$

$$(\lambda f. \lambda g. \lambda x. (f \, (g \, x)) \, \lambda x. (i \, x))$$
$$\Rightarrow (\lambda g. \lambda x. (\lambda x. (s \, x) \, (g \, x)) \, \lambda x. (i \, x))$$
$$\Rightarrow \lambda x. (\lambda x. (s \, x) \, (\lambda x. (i \, x) \, x))$$
$$\Rightarrow \lambda x. (\lambda x. (s \, x) \, (i \, x))$$
$$\Rightarrow \lambda x. (s \, (i \, x))$$
Free and Bound Variables

The lambda functional abstraction is the only syntactic construct that *binds* variables. That is, in an expression of the form:

$$\lambda v. e$$

we say that free occurrences of variable $v$ in expression $e$ are *bound*. All other variable occurrences are said to be *free*.

E.g.,

$$(\lambda x. \lambda y. (x \, y) \, (y \, w))$$

Bound Variables

Free Variables
α-renaming

Alpha renaming is used to prevent capturing free occurrences of variables when reducing a lambda calculus expression, e.g.,

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

This reduction **erroneously** captures the free occurrence of \(y\).

A correct reduction first renames \(y\) to \(z\), (or any other *fresh* variable) e.g.,

\[(\lambda x. \lambda y. (x \, y) \, (y \, w)) \Rightarrow (\lambda x. \lambda z. (x \, z) \, (y \, w)) \Rightarrow \lambda z. ((y \, w) \, z)\]

where \(y\) remains *free*.

C. Varela
Does the order of evaluation change the final result? Consider:

\[ \lambda x. (\lambda x. (s x) (\lambda x. (i x) x)) \]

There are two possible evaluation orders:

Applicative Order

\[ \Rightarrow \lambda x. (\lambda x. (s x) (i x)) \]
\[ \Rightarrow \lambda x. (s (i x)) \]

Normal Order

\[ \Rightarrow \lambda x. (\lambda x. (s x) (i x)) \]
\[ \Rightarrow \lambda x. (s (\lambda x. (i x) x)) \]
\[ \Rightarrow \lambda x. (s (i x)) \]

Is the final result always the same?

Recall semantics rule:

\[ (\lambda x. E \ M) \Rightarrow E\{M/x\} \]

C. Varela
Church-Rosser Theorem

If a lambda calculus expression can be evaluated in two different ways and both ways terminate, both ways will yield the same result.

Also called the diamond or confluence property.

Furthermore, if there is a way for an expression evaluation to terminate, using normal order will cause termination.
Order of Evaluation and Termination

Consider:

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

There are two possible evaluation orders:

\[
\begin{align*}
  (\lambda x. y (\lambda x. (x x)) \lambda x. (x x)) \\
  \Rightarrow (\lambda x. y (\lambda x. (x x)) \lambda x. (x x))
\end{align*}
\]

and:

\[
\begin{align*}
  (\lambda x. y (\lambda x. (x x)) \lambda x. (x x)) \\
  \Rightarrow y
\end{align*}
\]

In this example, normal order terminates whereas applicative order does not.

Recall semantics rule:

\[ (\lambda x. E M) \Rightarrow E\{M/x\} \]

Applicative Order

Normal Order
A lambda calculus expression with *no free variables* is called a *combinator*. For example:

I: \( \lambda x.x \) (Identity)

App: \( \lambda f.\lambda x.(f\ x) \) (Application)

C: \( \lambda f.\lambda g.\lambda x.(f\ (g\ x)) \) (Composition)

L: \( (\lambda x.(x\ x)\ \lambda x.(x\ x)) \) (Loop)

Cur: \( \lambda f.\lambda x.\lambda y.((f\ x)\ y) \) (Currying)

Seq: \( \lambda x.\lambda y.(\lambda z.y\ x) \) (Sequencing--normal order)

ASeq: \( \lambda x.\lambda y.(y\ x) \) (Sequencing--applicative order)

where \( y \) denotes a *thunk*, *i.e.*, a lambda abstraction wrapping the second expression to evaluate.

The meaning of a combinator is always the same independently of its context.
Combinators in Functional Programming Languages

Most functional programming languages have a syntactic form for lambda abstractions. For example the identity combinator:

\[ \lambda x.x \]

can be written in Oz as follows:

```
fun ${ X} X end
```

and it can be written in Scheme as follows:

```
(lambda(x) x)
```
Currying Combinator in Oz

The currying combinator can be written in Oz as follows:

```
fun {F}
  fun {X}
    fun {Y}
      {F X Y}
    end
  end
end
```

It takes a function of two arguments, F, and returns its curried version, e.g.,

```
{{Curry Plus} 2} 3} ⇒ 5
```
Exercises

20. PDCS Exercise 2.11.1 (page 31).
21. PDCS Exercise 2.11.2 (page 31).
22. PDCS Exercise 2.11.5 (page 31).
23. PDCS Exercise 2.11.6 (page 31).