Lambda Calculus (PDCS 2)

alpha-renaming, beta reduction, eta conversion, applicative and normal evaluation orders, Church-Rosser theorem, combinators, booleans

Carlos Varela
Rennselaer Polytechnic Institute

September 2, 2016

Lambda Calculus Syntax and Semantics

The syntax of a λ-calculus expression is as follows:

e	**::=**	**v**	variable
	|	***λ*v.e**	functional abstraction
	|	***(*e e*)***	function application

The semantics of a λ-calculus expression is called beta-reduction:

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

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

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:

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

((C S) I)

$$((\underline{\lambda f.\lambda g.\lambda x.(f\ (g\ x))\ \lambda x.(s\ x))}\ \lambda x.(i\ x))$$
$$\Rightarrow \underline{(\lambda g.\lambda x.(\lambda x.(s\ x)\ (g\ x))\ \lambda x.(i\ x))}$$
$$\Rightarrow \lambda x.(\lambda x.(s\ x)\ \underline{(\lambda x.(i\ x)\ x))}$$
$$\Rightarrow \lambda x.\underline{(\lambda x.(s\ x)\ (i\ x))}$$
$$\Rightarrow \lambda x.(s\ (i\ x))$$

Order of Evaluation in the Lambda Calculus

Does the order of evaluation change the final result?
Consider:

$$\lambda x.(\lambda x.(s\ x)\ (\lambda x.(i\ x)\ x))$$

> ***Recall semantics rule:***
>
> $$(\lambda x.E\ M) \Rightarrow E\{M/x\}$$

There are two possible evaluation orders:

$$\begin{aligned} & \lambda x.(\lambda x.(s\ x)\ \underline{(\lambda x.(i\ x)\ x)}) \\ \Rightarrow\ & \lambda x.\underline{(\lambda x.(s\ x)\ (i\ x))} \\ \Rightarrow\ & \lambda x.(s\ (i\ x)) \end{aligned}$$

Applicative Order

and:

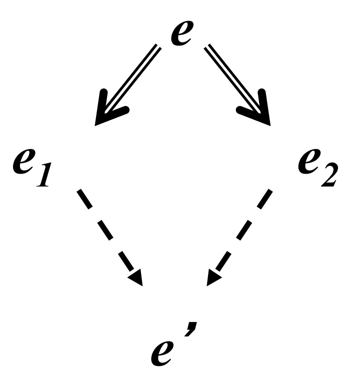
$$\begin{aligned} & \lambda x.\underline{(\lambda x.(s\ x)\ (\lambda x.(i\ x)\ x))} \\ \Rightarrow\ & \lambda x.(s\ \underline{(\lambda x.(i\ x)\ x)}) \\ \Rightarrow\ & \lambda x.(s\ (i\ x)) \end{aligned}$$

Normal Order

Is the final result always the same?

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:

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

Recall semantics rule:

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

There are two possible evaluation orders:

$$(\lambda x.y\ \underline{(\lambda x.(x\ x)\ \lambda x.(x\ x))})$$
$$\Rightarrow (\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))$$

Applicative Order

and:

$$\underline{(\lambda x.y\ (\lambda x.(x\ x)\ \lambda x.(x\ x)))}$$
$$\Rightarrow\ y$$

Normal Order

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

Free and Bound Variables

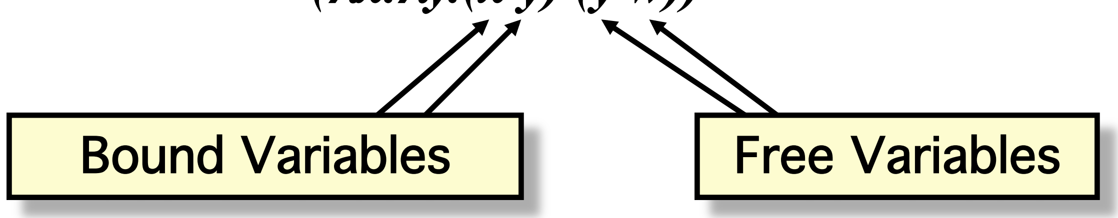
The lambda functional abstraction is the only syntactic construct that *binds* variables. That is, in an expression of the form:

$$\lambda v.e$$

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

E.g.,

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



Why α-renaming?

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

$$\underline{(\lambda x.\lambda y.(x\ y)\ (y\ w))}$$
$$\Rightarrow \lambda y.((y\ w)\ y)$$

This reduction **erroneously** captures the free occurrence of ***y***.

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

$$(\lambda x.\lambda y.(x\ y)\ (y\ w))$$
$$\Rightarrow \underline{(\lambda x.\lambda z.(x\ z)\ (y\ w))}$$
$$\Rightarrow \lambda z.((y\ w)\ z)$$

where ***y*** remains *free*.

α-renaming

Alpha renaming is used to prevent capturing free occurrences of variables when beta-reducing a lambda calculus expression.

In the following, we rename ***x*** to ***z***, (or any other *fresh* variable):

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

$$\xrightarrow{\alpha} \quad (\lambda z.(y\ z)\ x)$$

Only *bound* variables can be renamed. No *free* variables can be captured (become bound) in the process. For example, we *cannot* alpha-rename ***x*** to ***y***.

β-reduction

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

Beta-reduction may require alpha renaming to prevent capturing free variable occurrences. For example:

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

$$\xrightarrow{\alpha} (\lambda x.\lambda z.(x\ z)\ (y\ w))$$

$$\xrightarrow{\beta} \lambda z.((y\ w)\ z)$$

Where the *free* ***y*** remains free.

η-conversion

$$\lambda x.(E\ x) \xrightarrow{\eta} E$$

if ***x*** is *not* free in ***E***.

For example:

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

$$\xrightarrow{\alpha} (\lambda x.\lambda z.(x\ z)\ (y\ w))$$

$$\xrightarrow{\beta} \lambda z.((y\ w)\ z)$$

$$\xrightarrow{\eta} (y\ w)$$

Combinators

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

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
```

in Haskell as follows: `\x -> x`

and 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

Booleans and Branching (*if*) in λ Calculus

|*true*|:	$\lambda x.\lambda y.x$	(True)
|*false*|:	$\lambda x.\lambda y.y$	(False)
|*if*|:	$\lambda b.\lambda t.\lambda e.((b\ t)\ e)$	(If)

Recall semantics rule:

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

$(((if\ true)\ a)\ b)$

$$(((\underline{(\lambda b.\lambda t.\lambda e.((b\ t)\ e)\ \lambda x.\lambda y.x)}\ a)\ b)$$
$$\Rightarrow (\underline{(\lambda t.\lambda e.((\lambda x.\lambda y.x\ t)\ e)\ a)}\ b)$$
$$\Rightarrow \underline{(\lambda e.((\lambda x.\lambda y.x\ a)\ e)\ b)}$$
$$\Rightarrow (\underline{(\lambda x.\lambda y.x\ a)}\ b)$$
$$\Rightarrow \underline{(\lambda y.a\ b)}$$
$$\Rightarrow a$$

Exercises

1. PDCS Exercise 2.11.1 (page 31).
2. PDCS Exercise 2.11.2 (page 31).
3. PDCS Exercise 2.11.5 (page 31).
4. PDCS Exercise 2.11.6 (page 31).
5. Define Compose in Haskell. Demonstrate the use of curried Compose using an example.
6. PDCS Exercise 2.11.7 (page 31).
7. PDCS Exercise 2.11.9 (page 31).
8. PDCS Exercise 2.11.12 (page 31). Test your representation of booleans in Haskell.