

# Lambda Calculus (PDCS 2)

combinators, higher-order programming, recursion  
combinator, numbers, Church numerals

Carlos Varela  
Rensselaer Polytechnic Institute

September 6, 2016

# Lambda Calculus Syntax and Semantics

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

<b>e</b>	<b>::=</b>	<b>v</b>	variable
		<b><math>\lambda v.e</math></b>	functional abstraction
		<b>(e e)</b>	function application

The semantics of a  $\lambda$ -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**.

# $\alpha$ -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):

$$\begin{array}{l} (\lambda x. (y x) x) \\ \xrightarrow{\alpha} (\lambda z. (y z) x) \end{array}$$

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  $y$ .

# $\beta$ -reduction

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

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

$$\begin{aligned} & (\lambda x. \lambda y. (x y) (y w)) \\ & \xrightarrow{\alpha} (\lambda x. \lambda z. (x z) (y w)) \\ & \xrightarrow{\beta} \lambda z. ((y w) z) \end{aligned}$$

Where the *free*  $y$  remains free.

# Booleans and Branching (*if*) in $\lambda$ 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:*

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

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

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

# $\eta$ -conversion

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

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

For example:

$$\begin{aligned} & (\lambda x. \lambda y. (x \ y) \ (y \ w)) \\ & \xrightarrow{\alpha} (\lambda x. \lambda z. (x \ z) \ (y \ w)) \\ & \xrightarrow{\beta} \lambda z. ((y \ w) \ z) \\ & \xrightarrow{\eta} (y \ w) \end{aligned}$$

# 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
end
```

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

$$\{\{\{\text{Curry Plus}\} 2\} 3\} \Rightarrow 5$$

## Recursion Combinator (*Y* or *rec*)

Suppose we want to express a factorial function in the  $\lambda$  calculus.

$$f(n) = n! = \begin{cases} 1 & n=0 \\ n \cdot (n-1)! & n>0 \end{cases}$$

We may try to write it as:

$$f: \quad \lambda n. (if (= n 0) \\ \quad 1 \\ \quad (* n (f (- n 1))))$$

But  $f$  is a free variable that should represent our factorial function.

# Recursion Combinator ( $Y$ or *rec*)

We may try to pass  $f$  as an argument ( $g$ ) as follows:

$$f: \quad \lambda g. \lambda n. (\text{if } (= n 0) \\ \quad 1 \\ \quad (* n (g (- n 1))))$$

The *type* of  $f$  is:

$$f: (Z \rightarrow Z) \rightarrow (Z \rightarrow Z)$$

So, what argument  $g$  can we pass to  $f$  to get the factorial function?

# Recursion Combinator ( $Y$ or *rec*)

$$f: (Z \rightarrow Z) \rightarrow (Z \rightarrow Z)$$

$(f f)$  is not well-typed.

$(f I)$  corresponds to:

$$f(n) = \begin{cases} 1 & n=0 \\ n*(n-1) & n>0 \end{cases}$$

We need to solve the fixpoint equation:

$$(f X) = X$$

# Recursion Combinator ( $Y$ or *rec*)

$$(f X) = X$$

The  $X$  that solves this equation is the following:

$$\begin{aligned} X: & \quad (\lambda x. (\lambda g. \lambda n. (\text{if } (= n 0) \\ & \quad \quad \quad 1 \\ & \quad \quad \quad (* n (g (- n 1)))))) \\ & \quad \quad \lambda y. ((x x) y)) \\ & \quad \lambda x. (\lambda g. \lambda n. (\text{if } (= n 0) \\ & \quad \quad \quad 1 \\ & \quad \quad \quad (* n (g (- n 1)))))) \\ & \quad \quad \lambda y. ((x x) y)) \end{aligned}$$

# Recursion Combinator ( $Y$ or *rec*)

$X$  can be defined as  $(Y f)$ , where  $Y$  is the *recursion combinator*.

$Y$ :  $\lambda f. (\lambda x. (f \lambda y. ((x x) y)))$   
 $\lambda x. (f \lambda y. ((x x) y)))$

Applicative  
Order

$Y$ :  $\lambda f. (\lambda x. (f (x x)))$   
 $\lambda x. (f (x x))$

Normal Order

You get from the normal order to the applicative order recursion combinator by  $\eta$ -expansion ( $\eta$ -conversion from right to left).

# Natural Numbers in Lambda Calculus

$ 0 :$	$\lambda x.x$	(Zero)
$ 1 :$	$\lambda x.\lambda x.x$	(One)
...		
$ n+1 :$	$\lambda x. n $	(N+1)
$s:$	$\lambda n.\lambda x.n$	(Successor)

$$\begin{aligned} & (s\ 0) \\ & (\lambda n.\lambda x.n\ \lambda x.x) \\ & \Rightarrow \lambda x.\lambda x.x \end{aligned}$$

***Recall semantics rule:***

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

# Church Numerals

$ 0 $ :	$\lambda f. \lambda x. x$	(Zero)
$ 1 $ :	$\lambda f. \lambda x. (f x)$	(One)
...		
$ n $ :	$\lambda f. \lambda x. (f \dots (f x) \dots)$	(N applications of f to x)
$s$ :	$\lambda n. \lambda f. \lambda x. (f ((n f) x))$	(Successor)

$(s\ 0)$

*Recall semantics rule:*

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

$$\begin{aligned}
 & (\lambda n. \lambda f. \lambda x. (f ((n f) x))\ \lambda f. \lambda x. x) \\
 & \Rightarrow \lambda f. \lambda x. (f ((\lambda f. \lambda x. x\ f)\ x)) \\
 & \Rightarrow \lambda f. \lambda x. (f (\lambda x. x\ x)) \\
 & \Rightarrow \lambda f. \lambda x. (f\ x)
 \end{aligned}$$



# Church Numerals: isZero?

*Recall semantics rule:*

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

*isZero?:*                     $\lambda n.((n \lambda x.false) true)$                     (Is n=0?)

*(isZero? 0)*  
 $(\lambda n.((n \lambda x.false) true) \lambda f.\lambda x.x)$   
 $\Rightarrow ((\lambda f.\lambda x.x \lambda x.false) true)$   
 $\Rightarrow (\lambda x.x true)$   
 $\Rightarrow true$

*(isZero? 1)*  
 $(\lambda n.((n \lambda x.false) true) \lambda f.\lambda x.(f x))$   
 $\Rightarrow ((\lambda f.\lambda x.(f x) \lambda x.false) true)$   
 $\Rightarrow (\lambda x.(\lambda x.false x) true)$   
 $\Rightarrow (\lambda x.false true)$   
 $\Rightarrow false$

# Exercises

9. PDCS Exercise 2.11.10 (page 31). Test your representation of numbers in Haskell.
10. PDCS Exercise 2.11.11 (page 31).
11. Prove that your addition operation is correct using induction.