Lambda Calculus

higher-order programming, eta-conversion, recursion combinator, numbers, booleans

Carlos Varela
Rennselaer Polytechnic Institute

February 19, 2013

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 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**.

Normal vs Applicative 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

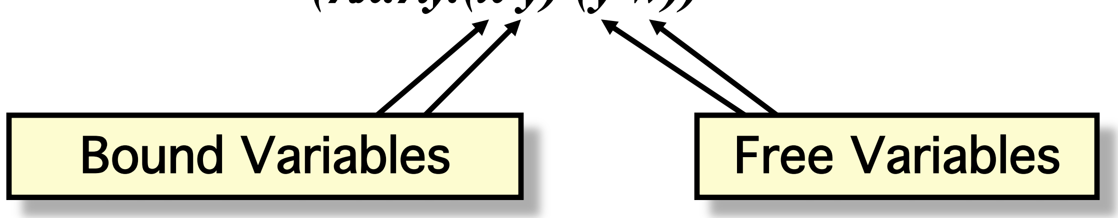
The lambda functional abstraction is the only syntactic construct that *binds* variables. That is, in an expression of the form:

$$\lambda \mathbf{v}.\mathbf{e}$$

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

E.g.,

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



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.

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

Suppose we want to express a factorial function in the λ calculus.

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

We may try to write it as:

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

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:

```
X:    (λx.(λg.λn.(if (= n 0)
                      1
                      (* n (g (- n 1))))
           λy.((x x) y))
       λx.(λg.λn.(if (= n 0)
                      1
                      (* n (g (- n 1))))
           λy.((x x) y)))
```

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 η-expansion (η-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:	$\lambda n.\lambda x.n$	(Succesor)

(s 0)

$\underline{(\lambda n.\lambda x.n\ \lambda x.x)}$

$\Rightarrow \lambda x.\lambda x.x$

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

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

24. PDCS Exercise 2.11.7 (page 31).
25. PDCS Exercise 2.11.9 (page 31).
26. PDCS Exercise 2.11.10 (page 31).
27. Prove that your addition operation is correct using induction.
28. PDCS Exercise 2.11.11 (page 31).
29. PDCS Exercise 2.11.12 (page 31).