Lambda Calculus (PDCS 2)

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

Carlos Varela
Rennselaer Polytechnic Institute

September 6, 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**.

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.

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:

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

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

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

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} $\Rightarrow$ 5

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

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: $(\lambda x.(\lambda g.\lambda n.(\mathit{if}\ (= n\ 0)\ 1\ (*\ n\ (g\ (-\ n\ 1))))\ \lambda y.((x\ x)\ y))\ \lambda x.(\lambda g.\lambda n.(\mathit{if}\ (= n\ 0)\ 1\ (*\ n\ (g\ (-\ n\ 1))))\ \lambda y.((x\ x)\ y)))$

```
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: $\lambda f.(\lambda x.(f\ \lambda y.((x\ x)\ y))$
$\quad\lambda x.(f\ \lambda y.((x\ x)\ y)))$

Applicative Order

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

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:	$\lambda n.\lambda x.n$	(Successor)

$$(s\ 0)$$

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

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

Recall semantics rule:

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

Church Numerals

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

Recall semantics rule:

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

$(s\ 0)$

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

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?)

$$(isZero?\ 0)$$
$$\underline{(\lambda} n.((n\ \lambda x.false)\ true)\ \lambda f.\lambda x.\underline{x)}$$
$$\Rightarrow (\underline{(\lambda f.\lambda x.x\ \lambda x.false)}\ true)$$
$$\Rightarrow (\lambda x.x\ true)$$
$$\Rightarrow true$$

$$(isZero?\ 1)$$
$$\underline{(\lambda} n.((n\ \lambda x.false)\ true)\ \lambda f.\lambda x.(f\ \underline{x))}$$
$$\Rightarrow (\underline{(\lambda f.\lambda x.(f\ x)\ \lambda x.false)}\ true)$$
$$\Rightarrow (\underline{\lambda x.(\lambda x.false\ x)}\ true)$$
$$\Rightarrow (\lambda x.false\ true)$$
$$\Rightarrow false$$

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.