

# Pure (Untyped) $\lambda$ -Calculus

©Andrey Kruglyak, 2010

Wednesday, April 21, 2010

1

# Pure (untyped) $\lambda$ -calculus

- ◀ An example of a simple formal language
- ◀ Invented by Alonzo Church (1936)
- ◀ Used as a “core language” (a calculus capturing the essential mechanisms of the language, with other features understood by way of translating them into the core) by Peter Landin (1964)
- ◀ Has seen widespread use for specification of programming language features, in language design and implementation, and in the study of type systems
- ◀ Is the underlying “computational substrate” for many type systems

Wednesday, April 21, 2010

2

# Pure (untyped) $\lambda$ -calculus

- ◀ Can be viewed both as a programming language in which computations can be described and as a mathematical object about which rigorous statements can be proved
- ◀ In its original form, all computation in  $\lambda$ -calculus is reduced to function definition and function application. But it can be easily enriched in a variety of ways, by adding special concrete syntax for features like numbers, booleans, tuples, etc.
- ◀ Inspired other core calculi used for similar purposes ( $\pi$ -calculus by Milner, Parrow and Walker, 1991, for defining semantics of message-based concurrent languages; object calculus by Abadi and Cardelli, 1996, for core features of object-oriented languages)

Wednesday, April 21, 2010

3

# Pure (untyped) $\lambda$ -calculus

- ◀ We will start by giving syntactical rules – the rules that define the structure of terms of  $\lambda$ -calculus
- ◀ These rules can be used to generate new terms, or they can be used to verify whether a particular sequence of words is a valid term in  $\lambda$ -calculus
- ◀ We will discuss the meaning of the terms a bit later!

Wednesday, April 21, 2010

4

## Syntax of $\lambda$ -calculus

- ◀ We will define the syntax using a BNF grammar
- ◀  $t ::= x \mid v \mid t t$   
Each term is either a variable ( $x$  stands for any variable), or a value, or an application (two terms separated by a space)
- ◀  $v ::= \lambda x.t$   
All values are  $\lambda$ -abstractions, which are built up by ' $\lambda$ ', followed by some variable name, followed by a dot, followed by some other term. For example, these are valid values:  
 $\lambda x.x \quad \lambda x.y \quad \lambda x.(x y) \quad \lambda y.(\lambda z.(y z))$   
The parentheses are usually omitted as an abstraction always stretches as far to the right as possible

Wednesday, April 21, 2010

5

## Syntax of $\lambda$ -calculus

- ◀ The same BNF grammar can also be written like this:

```
t ::=
  x
  v
  t t
v ::=
  λx.t
```

Wednesday, April 21, 2010

6

## Syntax of $\lambda$ -calculus

- ◀ Let us now extend the calculus by adding integers to values and addition and subtraction to terms
- ◀  $t ::= x \mid v \mid t t \mid t + t \mid t - t$
- ◀  $v ::= \lambda x.t \mid n$
- ◀ We can now give some intuition behind  $\lambda$ -calculus

Wednesday, April 21, 2010

7

## Intuition

- ◀ We can think of each  $\lambda$ -abstraction as a function:  
 $\lambda x.x+4$  takes  $x$  and returns " $x+4$ "
- ◀ Then each application becomes an application of the function:  
 $(\lambda x.x+4) 5$  becomes  $5+4$
- ◀ The beauty of  $\lambda$ -calculus is that a function can equally well be applied on any term, for example, another function:  
 $(\lambda x.x y) (\lambda z.z+3)$  becomes  $(\lambda z.z+3) y$  becomes  $y+3$
- ◀ And we can combine several  $\lambda$ -abstractions to define functions of several arguments:  
 $\lambda x.\lambda y.x+y$  takes  $x$  and  $y$  and returns " $x+y$ "  
(This is sometimes written as  $\lambda x y.x+y$ )

Wednesday, April 21, 2010

8

## Examples of terms

| A term in $\lambda$ -calculus     | A mathematical expression         |
|-----------------------------------|-----------------------------------|
| 5+x                               | 5+x                               |
| a-b                               | a-b                               |
| $\lambda x.x+4$                   | f(x), where f(x)=x+4              |
| $\lambda x.\lambda y.x+y+z$       | g(x,y), where g(x,y)=x+y+z        |
| $(\lambda x.x+4) 12$              | f(12), where f(x)=x+4             |
| $(\lambda x.\lambda y.x+y+z) 1 3$ | g(1,3), where g(x,y)=x+y+z        |
| $(\lambda x.\lambda y.x+y+z) 2$   | k(y) = g(2,y), where g(x,y)=x+y+z |
| $\lambda x.\lambda y.x (y z)$     | h(x,y), where h(x,y)=x(y(z))      |

- \* We say that an abstraction binds a variable
- \* Each occurrence of a variable in a term is either bound or free
- \* A term with no free variables is called closed, or a combinator

Wednesday, April 21, 2010

9

## Examples of terms

| A term in $\lambda$ -calculus     | A mathematical expression                       |
|-----------------------------------|---|
| 5+x                               | 5+x   |
| a-b                               | a-b   |
| $\lambda x.x+4$                   | f(x), where f(x)=x+4                            |
| $\lambda x.\lambda y.x+y+z$       | g(x,y), where g(x,y)=x+y+z                      |
| $(\lambda x.x+4) 12$              | f(12), where f(x)=x+4                           |
| $(\lambda x.\lambda y.x+y+z) 1 3$ | g(1,3), where g(x,y)=x+y+z                      |
| $(\lambda x.\lambda y.x+y+z) 2$   | g <sub>1</sub> (y) = g(2,y), where g(x,y)=x+y+z |
| $\lambda x.\lambda y.x (y z)$     | h(x,y), where h(x,y)=x(y(z))                    |

application is left-associative, so we leave out parentheses from  $((\lambda x.\lambda y.x+y+z) 1) 3$

- \* We say that an abstraction binds a variable
- \* Each occurrence of a variable in a term is either bound or free
- \* A term with no free variables is called closed, or a combinator

Wednesday, April 21, 2010

10

## $\beta$ -reduction

- ◁  $\beta$ -reduction of an application:  
 $(\lambda x.t_1) t_2 \rightarrow [x \mapsto t_2] t_1$   
substitution of all free occurrences of x in  $t_1$  by  $t_2$
- ◁ It is important that only free occurrences are substituted!

For example:

$(\lambda x. x + (\lambda x. x - 3) 4) 10 \rightarrow [x \mapsto 10] x + (\lambda x. x - 3) 4 \rightarrow 10 + (\lambda x. x - 3) 4$  which should clearly give us 11

If we replaced all occurrences of x with 10, we would have  $10 + (\lambda x. 10 - 3) 4$  which would give us 17

Wednesday, April 21, 2010

11

## $\alpha$ -conversion

- ◁  $\alpha$ -conversion:  
 $\lambda x.t_1 \rightarrow \lambda y.[x \mapsto y] t_1$   
change of the name of a bound variable
- ◁ Two terms are normally considered equivalent "up to  $\alpha$ -conversion":  
 $\lambda x.x + 4 \equiv \lambda y.y + 4$
- ◁ It is often handy (but not strictly required) to perform  $\alpha$ -conversion to distinguish between free and bound occurrences of the same variable:  
 $x - (\lambda x.x + 4) 7 \equiv x - (\lambda y.y + 4) 7$

Wednesday, April 21, 2010

12

## Evaluation

- Operational semantics of  $\lambda$ -calculus is given by defining an evaluation relation, denoted by " $\rightarrow$ "
  - In general, evaluation may not be deterministic, i.e. there may be more than one place in the term that can be reduced
  - Even when evaluation is not deterministic, it is possible that any evaluation path ultimately leads to the same value
  - Evaluation function is typically a partial function, i.e. some terms may be stuck, for example  $3 + (\lambda x.x + 4)$

## Operational Semantics

$$\frac{t_2 \rightarrow t_2'}{t_1 t_2 \rightarrow t_1 t_2'} \quad (\text{App1})$$

$$\frac{}{(\lambda x.t_1) v \rightarrow [x \mapsto v] t_1} \quad (\text{App2})$$

$$\frac{t_1 \rightarrow t_1'}{t_1 + t_2 \rightarrow t_1' + t_2} \quad (\text{Add1})$$

$$\frac{t_1 \rightarrow t_1'}{t_1 - t_2 \rightarrow t_1' - t_2} \quad (\text{Sub1})$$

$$\frac{t_2 \rightarrow t_2'}{v + t_2 \rightarrow v + t_2'} \quad (\text{Add2})$$

$$\frac{t_2 \rightarrow t_2'}{v - t_2 \rightarrow v - t_2'} \quad (\text{Sub2})$$

$$\frac{v_1 + v_2 = v_3}{v_1 + v_2 \rightarrow v_3} \quad (\text{Add3})$$

$$\frac{v_1 - v_2 = v_3}{v_1 - v_2 \rightarrow v_3} \quad (\text{Sub3})$$

## Operational Semantics

$$\frac{t_2 \rightarrow t_2'}{t_1 t_2 \rightarrow t_1 t_2'} \quad (\text{App1})$$

$$\frac{}{(\lambda x.t_1) v \rightarrow [x \mapsto v] t_1} \quad (\text{App2})$$

$$\frac{t_1 \rightarrow t_1'}{t_1 + t_2 \rightarrow t_1' + t_2} \quad (\text{Add1})$$

$$\frac{t_2 \rightarrow t_2'}{v_1 + t_2 \rightarrow v_1 + t_2'} \quad (\text{Add2})$$

$$\frac{v_1 + v_2 = v_3}{v_1 + v_2 \rightarrow v_3} \quad (\text{Add3})$$

Is this evaluation relation deterministic?

$$\frac{t_1 \rightarrow t_1'}{t_1 - t_2 \rightarrow t_1' - t_2} \quad (\text{Sub1})$$

$$\frac{t_2 \rightarrow t_2'}{v_1 - t_2 \rightarrow v_1 - t_2'} \quad (\text{Sub2})$$

$$\frac{v_1 - v_2 = v_3}{v_1 - v_2 \rightarrow v_3} \quad (\text{Sub3})$$

## Encoding booleans

- The original  $\lambda$ -calculus is in fact sufficiently expressive in itself. Practically any other language constructs can be encoded in it, including natural numbers and booleans.
- However, this encoding is often cumbersome and difficult to work with, so it is common to add other constructs as primitives (as we did with integers and operations on integers).
- At the same time, people normally try to give added primitives the same semantics as their encodings would have in the original  $\lambda$ -calculus.

## Encoding booleans

- Booleans can be encoded as follows:

```
True =  $\lambda x.\lambda y.x$   
False =  $\lambda x.\lambda y.y$   
if z then x else y = z x y
```

- For example:

```
if True then 5 else 6 becomes  
 $(\lambda x.\lambda y.x) 5 6 \rightarrow ([x \mapsto 5](\lambda y.x)) 6 = (\lambda y.5) 6 \rightarrow$   
 $[y \mapsto 6]5 = 5$   
and if False then 5 else 6 becomes  
 $(\lambda x.\lambda y.y) 5 6 \rightarrow ([x \mapsto 5](\lambda y.y)) 6 = (\lambda y.y) 6 \rightarrow$   
 $[y \mapsto 6]y = 6$ 
```