

LINFO1104

Concepts, paradigms, and semantics of programming languages

Lecture 5

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be



1

Overview of lecture 5

- Refresher of higher-order programming
 - Higher-order programming is a key technique
 - Many powerful techniques are possible, and we will use them to build abstractions
- Lambda calculus
 - A very simple model of computation that is Turing equivalent
 - All data types can be encoded in lambda calculus
 - Church-Rosser theorem: lambda calculus is confluent, i.e., same results for all reduction orders
 - The key reason why functional programming is an important paradigm
 - The foundation of higher-order programming and functional programming languages



2



Introduction

- Lambda calculus is a formal system in mathematical logic for expressing computation
 - It is based on function abstraction and application, using variable binding and substitution
 - It was introduced by logician Alonzo Church in the 1930s as part of research into the foundations of mathematics
- Lambda calculus is a universal model of computation that can be used to simulate a Turing machine
 - Untyped lambda calculus, introduced by Church in 1936
 - In the 1960s, the relation to programming languages was clarified (Peter Landin 1965) and it has since been used as a foundation for understanding and designing programming languages

3



Basic concepts

- The lambda calculus provides a simple semantics for computation
- The lambda calculus has only anonymous functions of one argument:
 $\text{sum_square}(x,y) = x^2 + y^2$
becomes an anonymous function:
 $(x,y) \rightarrow x^2 + y^2$
of one argument:
 $x \rightarrow (y \rightarrow x^2 + y^2)$
- Converting functions into nested one-argument functions is called **currying**

4

Syntax of lambda expressions



- Lambda expressions are composed of:
 - Variables x, y, \dots
 - Abstraction symbols λ (lambda) and $.$ (dot)
 - Parentheses
- Lambda terms t are defined with the following rule:

$$t ::= x \mid (\lambda x. t) \mid (t_1 t_2)$$

- Terminology:
 - $(\lambda x. t)$ is called an *abstraction*
 - $(t_1 t_2)$ is called an *application*

5

Lambda expressions in Oz



- Lambda expressions in Oz:
 - $\lambda x. t$
`fun {$ X} T end`
 - $(t_1 t_2)$
`{T1 T2}`
- Currying in Oz:
`F=fun {$ X Y} T end`
becomes:
`F=fun {$ X} fun {$ Y} T end end`
- The call `{F X Y}` becomes `{{F X} Y}`

6

Free and bound variables



- The abstraction operator λ binds its variable when it occurs in the body of the abstraction
- In a term $\lambda x.t$, the part λx is called the *binder*, and it *binds* the variable x in t
- The set of free variables in a term t is denoted as $FV(t)$ and is defined as follows:
 - $FV(x) = \{x\}$ where x is a variable
 - $FV(\lambda x.t) = FV(t) \setminus \{x\}$
 - $FV((t_1 t_2)) = FV(t_1) \cup FV(t_2)$

7

Semantics of lambda expressions



- The meaning of lambda terms is defined by how they can be reduced
- There are three possible reductions:
 - **α -renaming**: change bound variable names
 - **β -reduction**: apply functions to arguments
 - **η -reduction**: remove unused variables (extensionality)
- We show reduction steps with arrows:
 - $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_{n-2} \rightarrow t_{n-1} \rightarrow \dots$
 - We write $t_i \rightarrow^* t_j$ for zero or more reductions
 - We write $t_i \rightarrow_\beta t_j$ for a β -reduction

8

α -renaming (α -conversion)



- Allows bound variable names to be changed:
 - Example: $\lambda x.x \rightarrow \lambda y.y$
- A name can be changed only if it does not introduce a name conflict (in $\lambda x.t$, the relationship between λx and the bound variables in t must be unchanged):
 - It cannot change a bound variable in a subterm
 - Allowed $\lambda x.\lambda x.x \rightarrow \lambda y.\lambda x.x$ but not allowed $\lambda x.\lambda x.x \rightarrow \lambda y.\lambda x.y$
 - It cannot do capturing
 - Not allowed $\lambda x.\lambda y.x \rightarrow \lambda y.\lambda y.y$ because y is captured
- Terms differing by α -renaming are called α -equivalent
 - Terms that are α -equivalent form an equivalence class

9

Substitution



- Substitution $t_1[x:=t_2]$ replaces all free occurrences of x in t_1 by t_2
- Definition:
 - $x[x:=t] = t$
 - $y[x:=t] = y$, if $x \neq y$
 - $(t_1 t_2)[x:=t] = (t_1[x:=t]) (t_2[x:=t])$
 - $(\lambda x.t_1)[x:=t_2] = \lambda x.t_1$
 - $(\lambda y.t_1)[x:=t_2] = \lambda y.(t_1[x:=t_2])$, if $x \neq y$ and $y \notin FV(t_2)$
- α -renaming is allowed to make substitution possible
 - Substitution defined up to α -equivalence

10

β-reduction



- β-reduction is the idea of function application
 - It is defined in terms of substitution
- Definition:
 $(\lambda x. t_1) t_2 \rightarrow t_2[x := t_1]$
- Example: $(\lambda x. (x x)) y \rightarrow (y y)$

11

η-reduction



- η-reduction expresses the idea that two functions are the same if and only if they have the same results for all arguments
- Definition:
 $\lambda x. (t x) \rightarrow t$ if $x \notin FV(t)$
- Example: $(\lambda x. (t x)) a \rightarrow (t a)$ (if x is not free in t)
It means that $\lambda x. (t x)$ is the same function as t

12

Summary of reduction rules



- α -renaming
 $\lambda x. t_1[x] \rightarrow \lambda y. t_2[y]$
(change bound vars without capture)
- β -reduction
 $(\lambda x. t_1) t_2 \rightarrow t_2[x:=t_1]$
- η -reduction
 $\lambda x. (t x) \rightarrow t$ if $x \notin FV(t)$
- Examples on the board!

13

Notation conventions



- Important when writing and manipulating lambda expressions!
 - Drop outermost parentheses: $(t_1 t_2) \rightarrow t_1 t_2$
 - Applications are left-associative: $t_1 t_2 t_3 \rightarrow (t_1 t_2) t_3$
 - Abstraction body extends right: $\lambda x. t_1 t_2$ means $\lambda x. (t_1 t_2)$
 - Sequence of abstractions: $\lambda x. \lambda y. \lambda z. t$ written as $\lambda xyz. t$
- You will see why this is important when we start manipulating big lambda expressions
 - We will also use **abbreviations** a lot, it really helps when doing lambda computations by hand

14



Encoding datatypes

- The untyped lambda calculus can be used to do all computations
 - It is actually Turing equivalent
- One way to show this is to encode datatypes as lambda terms
 - Numbers and arithmetic in lambda calculus
 - Boolean operations in lambda calculus
 - Lists in lambda calculus
 - Recursive functions in lambda calculus

15



Arithmetic: numbers

- Encoding natural numbers (**Church numerals**):
 - $0 \triangleq \lambda f.(\lambda x.x)$
 - $1 \triangleq \lambda f.(\lambda x.(f\ x))$
 - $2 \triangleq \lambda f.(\lambda x.(f\ (f\ x)))$
 - $3 \triangleq \lambda f.(\lambda x.(f\ (f\ (f\ x))))$
 - (we use the symbols 0, 1, 2, 3 as short-cuts for the lambda terms)
(symbol " \triangleq " means "is defined as")
- A Church numeral is a higher-order function: it takes a single-argument function f and returns another single-argument function
- The Church numeral n is a function that takes a function f as argument and returns the n -th composition of f , i.e., f composed with itself n times: it is like saying " f is applied n times"

16

Arithmetic: operations



- **Successor function** takes Church numeral n and returns $n+1$:

$$\text{SUCC} \triangleq \lambda n. \lambda f. \lambda x. f ((n f) x)$$

$$\text{SUCC} \triangleq \lambda n. \lambda f. \lambda x. f (n f x)$$
- **Addition** (plus) uses the fact that the m -th composition of f composed with the n -th composition of f gives the $(m+n)$ -th composition of f :

$$\text{PLUS} \triangleq \lambda m. \lambda n. \lambda f. \lambda x. (m f) ((n f) x)$$

$$\text{PLUS} \triangleq \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)$$

17

Example of successor



- $\text{SUCC } 1 \equiv$
 $(\lambda n. \lambda f. \lambda x. f (n f x)) \lambda f. (\lambda x. (f x)) \rightarrow$
 $\lambda f. \lambda x. f ((\lambda f. (\lambda x. (f x))) f x) \rightarrow$
 $\lambda f. \lambda x. f (f x) \equiv$
 2
- We have incremented 1!
- The symbol “ \equiv ” means “is equivalent to”
 - We can always replace an abbreviation by the lambda expression it is equivalent to

18

Arithmetic: operations



- We can verify that PLUS 2 3 is equivalent to 5
- **Multiplication** can be defined as:
$$\text{MULT} \triangleq \lambda m. \lambda n. \lambda f. m \ (n \ f)$$

or else:
$$\text{MULT} \triangleq \lambda m. \lambda n. m \ (\text{PLUS } n) \ 0$$

“repeat m times the ‘PLUS n’ starting with 0”
- **Exponentiation** can be defined as:
$$\text{POW} \triangleq \lambda b. \lambda e. e \ b$$

19

Arithmetic: operations



- The predecessor function, defined as n-1 for a positive integer n, and PRED 0 = 0, is much harder:
$$\text{PRED} \triangleq \lambda n. \lambda f. \lambda x. n \ (\lambda g. \lambda h. h \ (g \ f)) \ (\lambda u. x) \ (\lambda u. u)$$
- With PRED we can define subtraction:
$$\text{SUB} \triangleq \lambda m. \lambda n. n \ \text{PRED } m$$
- The term SUB m n yields m-n when m>n and 0 otherwise

20

Logical operations



- By convention, we express **true** and **false** as follows:
 $\text{TRUE} \triangleq \lambda x. \lambda y. x$
 $\text{FALSE} \triangleq \lambda x. \lambda y. y$
- We can define **logic operators**:
 $\text{AND} \triangleq \lambda p. \lambda q. p \ q \ p$
 $\text{OR} \triangleq \lambda p. \lambda q. p \ p \ q$
 $\text{NOT} \triangleq \lambda p. p \ \text{FALSE} \ \text{TRUE}$
 $\text{IFTHENELSE} \triangleq \lambda p. \lambda a. \lambda b. p \ a \ b$
- For example: (example on the board)
 $\text{AND} \ \text{TRUE} \ \text{FALSE}$
 $\equiv (\lambda p. \lambda q. p \ q \ p) \ \text{TRUE} \ \text{FALSE} \rightarrow_{\beta} \text{TRUE} \ \text{FALSE} \ \text{TRUE}$
 $\equiv (\lambda x. \lambda y. x) \ \text{FALSE} \ \text{TRUE} \rightarrow_{\beta} \text{FALSE}$

21

Predicates



- A **predicate** is a function that returns a boolean value
- **Compare with zero**:
Return TRUE if the argument is 0 and FALSE if the argument is any other number:
 $\text{ISZERO} \triangleq \lambda n. n \ (\lambda x. \text{FALSE}) \ \text{TRUE}$
- **Less-than-or-equal**:
 $\text{LEQ} \triangleq \lambda m. \lambda n. \text{ISZERO} \ (\text{SUB} \ m \ n)$

22

Pairs (cons cells)



- A **pair** is a 2-tuple, it can be defined in terms of TRUE and FALSE
 - PAIR encapsulates the pair (x,y), FIRST returns the first element, and SECOND returns the 2nd
- $\text{PAIR} \triangleq \lambda x.\lambda y.\lambda f.f\ x\ y$
 $\text{FIRST} \triangleq \lambda p.p\ \text{TRUE}$
 $\text{SECOND} \triangleq \lambda p.p\ \text{FALSE}$
 $\text{NIL} \triangleq \lambda x.\text{TRUE}$
 $\text{NULL} \triangleq \lambda p.p\ (\lambda x.\lambda y.\text{FALSE})$ (test if nil, return TRUE or FALSE)

23

Using pairs



- A **list** is either NIL (empty list) or the PAIR of an element and a smaller list
- Example of use of pairs:
 $\text{SHIFTINC} \triangleq \lambda x.\text{PAIR}\ (\text{SECOND}\ x)\ (\text{SUCC}\ (\text{SECOND}\ x))$
 - Maps (m,n) to (n,n+1)
- With SHIFTINC we can define **predecessor** in a simpler way:
 $\text{PRED} \triangleq \lambda n.\text{FIRST}\ (n\ \text{SHIFTINC}\ (\text{PAIR}\ 0\ 0))$

24

Recursive functions (1)



- Since functions are anonymous, we can't do recursion directly in lambda calculus
 - However, we can do recursion by arranging for a lambda term to get itself as its argument
 - Kind of like this in Oz:

```
Fact=fun {$ F N}  
    if N==0 then 1  
    else N*{F F N-1} end  
end  
{Browse {Fact Fact 5}} % Displays 120
```

25

Recursive functions (2)



- We show how to do recursive factorial in lambda calculus
 - We start with a factorial written using the data and control structures we already defined:
$$\text{Fact}(n) = \text{if } n=0 \text{ then } 1 \text{ else } n \times \text{Fact}(n-1)$$
- We rewrite this so it becomes a function of two arguments, where the first argument is the function itself:
$$G := \lambda r. \lambda n. (\text{if } n=0 \text{ then } 1 \text{ else } n \times (r \ n-1))$$

26

Recursive functions (3)



- We define a helper called “Y combinator”:

$$Y \triangleq \lambda g. (\lambda x. g \ (x \ x)) \ (\lambda x. g \ (x \ x))$$
 - We can show: $Y \ g \rightarrow^* g \ (Y \ g)$
 - This extracts the g and calls it with argument $(Y \ g)$
 - Remember that g has two arguments, so $g \ (Y \ g)$ is a one-arg function!
- Now we can do the recursive factorial:

$$(Y \ G) \ 4 \rightarrow^* G \ (Y \ G) \ 4 \rightarrow$$

$$(\lambda r. \lambda n. (\text{if } n=0 \text{ then } 1 \text{ else } n \times (r \ n-1))) \ (Y \ G) \ 4 \rightarrow$$

$$\text{if } 4=0 \text{ then } 1 \text{ else } 4 \times ((Y \ G) \ 4-1) \rightarrow$$

$$4 \times (G \ (Y \ G) \ 4-1) \rightarrow \dots \rightarrow$$

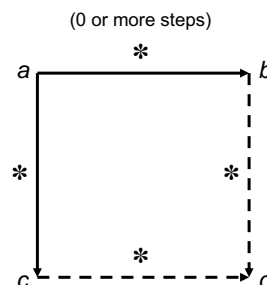
$$4 \times 3 \times 2 \times 1 \times 1 \rightarrow 24$$

27

Church-Rosser theorem



- An amazing property of lambda calculus is that the order of reduction makes no difference
- **Church-Rosser theorem:**
 If a reduces to b (in 0 or more steps), and a reduces to c (in 0 or more steps), then there exists a term d such that b and c can reduce to d
- We say that the lambda calculus is *confluent* or that it has the *Church-Rosser property*
- It means that the result of a computation is the same no matter in what order the reductions are done



28

Lambda calculus and programming languages



- Functional programming languages can be understood in terms of the lambda calculus
 - Procedure values (lexically scoped closures) are lambda functions
 - Eager versus lazy evaluation is just a different reduction strategy
- Reduction strategies define in what order expressions are computed:

- **Applicative order**: leftmost **innermost** first (arguments evaluated before function)

Traditional langs.
(Java, Python)

- **Call by value (eager evaluation)**: similar to applicative order, but no reductions inside function definitions (compiled functions are not changed)

- **Normal order**: leftmost **outermost** first (function evaluated before arguments)

- **Call by name**: similar to normal order except no reductions inside function definitions (compiled functions are not changed)

Lazy functional
langs (Haskell)

- **Call by need (lazy evaluation)**: similar to normal order except that functions are shared, not copied (evaluated at most once)

29

Applicative order example



- **fun** {Double X} X+X **end**
fun {Average X Y} (X+Y)/2 **end**

{Double {Average 5 7}} →
{Double ((5+7)/2)} →
{Double (12/2)} →
{Double 6} →
6+6 →
12

30



Normal order example

- **fun** {Double X} X+X **end**
fun {Average X Y} (X+Y)/2 **end**

{Double {Average 5 7}} →
{Average 5 7}+{Average 5 7} →
((5+7)/2)+{Average 5 7} →
(12/2)+{Average 5 7} →
6+{Average 5 7} →
6+((5+7)/2) →
6+(12/2) →
6+6 →
12

Lazy evaluation is similar except that the Average function would only be evaluated once (the two copies would be shared)

31



If statement evaluation

- **if** 3<4 **then** 5+5 **else** 1/0 **end**

- Normal order:

if 3<4 **then** 5+5 **else** 1/0 **end** →
if true then 5+5 **else** 1/0 **end** →
5+5 →
10

- Applicative order:

if 3<4 **then** 5+5 **else** 1/0 **end** →
if true then 5+5 **else** 1/0 **end** →
if true then 10 **else** 1/0 **end** →
[Error: Division by zero]

- Most languages do **if** statements with applicative order for the condition and with normal order for the **then** and **else** parts!

32

Variations and extensions



- Theoretical work on programming languages uses lambda calculus as a foundation
 - **Typed lambda calculus**: lambda calculus with typed variables and functions
 - **System F**: typed lambda calculus with type variables (variables ranging over types)
 - **Calculus of constructions**: typed lambda calculus with types as first-class values
 - **Combinatory logic**: logic without variables
 - **SKI combinator calculus**: equivalent to lambda calculus but without variable substitutions
 - **Oz kernel language**: a lambda calculus with futures (single-assignment variables), dataflow synchronization, threads, and lazy evaluation

33

Summary



- The lambda calculus is part of the theoretical foundation of almost all programming languages
 - All except for logic and constraint languages, which are based on formal logic
 - Lambda calculus is equivalent in expressiveness to Turing machine
 - Introduced by Alonzo Church (1936)
 - Relationship to programming recognized by Peter Landin (1965)
- The lambda calculus has strong properties
 - **Higher-order programming** is a consequence of the lambda calculus
 - It is the foundation of language abstraction (objects, classes, ADTs, components, etc.)
 - The result of a computation is independent of the reduction strategy (**confluence**) as stated in the **Church-Rosser theorem**
 - Important reduction strategies are eager evaluation, lazy evaluation, and dataflow concurrency
 - **Functional programming languages** (Haskell, Scheme, OCaml, Oz, etc.) are designed to take advantage of these properties

34