λ

1

LINFO1104 – LSINC1104
Concepts, paradigms, and semantics of programming languages

Lecture 5
Lambda calculus

Peter Van Roy

ICTEAM Institute
Université catholique de Louvain

peter.vanroy@uclouvain.be

2

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 complete
  - All data types and control can be encoded in lambda calculus
  - Church-Rosser theorem: lambda calculus is confluent, i.e., a computation gives the 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

3

Definition of λ calculus

4

Introduction

- Lambda calculus is a formal system in mathematical logic for expressing computation
  - It contains only function definition and call, using variable binding and substitutions
  - 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

5

Functions of one argument

- Lambda calculus has only anonymous functions of one argument:

  $sum\_square(x,y) = x^2+y^2$

  becomes an anonymous function:

  $(x,y) \rightarrow x^2+y^2$

  of one argument:

  $x \rightarrow (y \rightarrow x^2+y^2)$

- In lambda notation, this is written:

  $\lambda x.\ \lambda y.\ x^2+y^2$

- Converting functions into nested one-argument functions is called currying (named after logician Haskell B. Curry)

6

Syntax of lambda expressions

- Lambda expressions are composed of:
  - Variables *x*, *y*, …
  - Abstraction symbols λ (lambda) and . (dot)
  - Parentheses ( )
- Lambda terms *t* are defined with the following syntax:

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

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

7

Lambda expressions in Oz

- Lambda expressions in Oz:
  - $\lambda x.\ t$
    fun {$ X} *T* **end**
  - $(t_1\ t_2)$
    $\{T_1\ T_2\}$
- Currying in Oz:
  - The definition
    F=**fun** {$ X Y} T **end**
    becomes
    F=**fun** {$ X} **fun** {$ Y} T **end** **end**
  - The call {F X Y} becomes {{F X} Y}

8

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:
  $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_{n-2} \rightarrow t_{n-1} \rightarrow \cdots$
  - We write $t_i \rightarrow^* t_j$ for zero or more reductions
  - We write $t_i \rightarrow_\beta t_j$ for a β-reduction

9

Free and bound variables

- Consider the term $\lambda x.t$
  - We say that the operator λx binds the variable x in t
  - The variables in t that are not bound by any λ are called free in t
  - If x is free in t then we say $\lambda x.t$ captures x
- 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
  $FV(\lambda x.t)=FV(t)\setminus\{x\}$
  $FV((t_1\ t_2))=FV(t_1) \cup FV(t_2)$

10

α-renaming (α-conversion)

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

11

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 $y \notin FV(t_2)$
- To do a substitution, α-renaming is sometimes needed
  - Substitution is not allowed to capture free variables
  - For example, (λx.y)[y:=x] can be done as (λz.y)[y:=x]

12

β-reduction

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

13

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
  - This is called extensionality: two functions are the same if they have the same external properties (even if definitions are different)
- Definition:
  $\lambda x.(t\ x) \rightarrow t$ if $x \notin FV(t)$
- Motivation:
  - Apply the term t to a: this gives (t a)
  - Apply the term (λx.(t x)) to a: this also gives (t a)
  - Therefore, t and (λx.(t x)) give the same result for any argument a
  - Therefore, we can say that t and (λx.(t x)) are the same function

14

Summary of reduction rules

- α-renaming

  $\lambda x.t_1[x] \rightarrow \lambda y.t_1[y]$

  (change bound vars without capture)
- β-reduction

  $(\lambda x.t_1)\ t_2 \ \rightarrow\ t_1[x:=t_2]$

  (replace free x of t_1 by t_2 without capture)
- η-reduction

  $\lambda x.(t\ x) \rightarrow t$ if $x \notin FV(t)$
- Examples on the board!

15

Computing with λ calculus

16

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: λx.λy.λz.t written as λxyz.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

17

Encoding datatypes

- Untyped lambda calculus can do all computations
  - It is Turing complete
- One way to show this is to encode datatypes and control operations as lambda terms
  - Numbers and arithmetic in lambda calculus
  - Boolean operations and conditional (if statement) in lambda calculus
  - Lists in lambda calculus (data structures)
  - Recursive functions in lambda calculus

18

Arithmetic: numbers

- Encoding natural numbers (Church numerals):
  - 0 ≜ λf.(λx.x)
  - 1 ≜ λf.(λx.(f x))
  - 2 ≜ λf.(λx.(f (f x)))
  - 3 ≜ λf.(λx.(f (f (f x))))
  - (we use the symbols 0, 1, 2, 3 as abbreviations for the lambda terms) (symbol "≜" 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"

19

Arithmetic: operations

- Successor function takes Church numeral n and returns n+1:
  SUCC ≜ λn.λf.λx.f ((n f) x)
  SUCC ≜ λn.λf.λx.f (n f x)
- Addition (plus) because the m-th composition of f composed with the n-th composition of f gives the (m+n)-th composition of f:
  PLUS ≜ λm.λn.λf.λx.(m f) ((n f) x)
  PLUS ≜ λm.λn.λf.λx.m f (n f x)

20

Example of successor

- SUCC 1 ≡
  (λn.λf.λx.f (n f x)) λf.(λx.(f x)) →
  λf.λx.f ((λf.(λx.(f x)) f x) →
  λf.λx.f (f x) ≡
  2
- We have incremented 1!
- The symbol "≡" means "is equivalent to"
  - We can always replace an abbreviation by the lambda expression it is equivalent to

Arithmetic: operations

- We can verify that PLUS 2 3 is equivalent to 5
- Multiplication can be defined as:
  MULT ≜ λm.λn.λf.m (n f)
  or else:
  MULT ≜ λm.λn.m (PLUS n) 0
  "repeat m times the 'PLUS n' starting with 0"
- Exponentiation can be defined as:
  POW ≜ λb.λe.e b

Arithmetic: operations

- The predecessor function, defined as n-1 for a positive integer n, and PRED 0 = 0, is much harder:

  PRED ≜ λn.λf.λx.n (λg.λh.h (g f)) (λu.x) (λu.u)

- With PRED we can define subtraction:
  SUB ≜ λm.λn.n PRED m
- The term SUB m n yields m-n when m>n and 0 otherwise

23

Logical operations

- By convention, we express true and false as follows:
  TRUE ≜ λx.λy.x
  FALSE ≜ λx.λy.y

- We can define logic operators:
  AND ≜ λp.λq.p q p
  OR ≜ λp.λq.p p q
  NOT ≜ λp.p FALSE TRUE
  IFTHENELSE ≜ λp.λa.λb.p a b

- For example: (example on the board)
  AND TRUE FALSE
  ≡ (λp.λq.p q p) TRUE FALSE $\rightarrow_\beta$ TRUE FALSE TRUE
  ≡ (λx.λy.x) FALSE TRUE $\rightarrow_\beta$ FALSE

24

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:
  ISZERO ≜ λn.n (λx.FALSE) TRUE
- Less-than-or-equal:
  LEQ ≜ λm.λn.ISZERO (SUB m n)

25

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
- PAIR ≜ λx.λy.λf.f x y
  FIRST ≜ λp.p TRUE
  SECOND ≜ λp.p FALSE
  NIL ≜ λx.TRUE
  NULL ≜ λp.p (λx.λy.FALSE) (test if nil, return TRUE or FALSE)

26

Using pairs

- A list is either NIL (empty list) or the PAIR of an element and a smaller list

- Example of use of pairs:
  SHIFTINC ≜ λx.PAIR (SECOND x) (SUCC (SECOND x))
  - Maps (m,n) to (n,n+1)
- With SHIFTINC we can define predecessor in a simpler way:
  PRED ≜ λn.FIRST (n SHIFTINC (PAIR 0 0))

27

Recursive functions

- 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 an argument
  - Kind of like this in Oz:

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

28

Recursive functions: adding an extra argument

- Let us do recursive factorial in λ calculus
- We start with a factorial written using the data and control structures we already defined:
  Fact(n) ≜ **if** n=0 **then** 1 **else** n × Fact(n-1)
- We rewrite this so it becomes a function of two arguments, where the first argument is the function itself:
  G ≜ λf. λn. (**if** n=0 **then** 1 **else** n × (f n-1))

29

Recursive functions: the Y combinator

- We define a helper called "Y combinator":
  Y ≜ λg.(λx.g (x x)) (λx.g (x x))
- We can show: Y g $\rightarrow^*$ g (Y g)
  (Y g) = λg.(λx.g (x x)) (λx.g (x x)) g →
  (λx.g (x x)) (λx.g (x x)) →
  g ((λx.g (x x)) (λx.g (x x))) = g (Y g)
- We say that (Y g) is a fixed point of g
  (Y g) $\rightarrow^*$ g (Y g)
- This extracts the g and calls it with argument (Y g)
  - We use the g right away
  - The (Y g) remains, so we can use it again later

30

Recursive functions: using the Y combinator

- Now we can do the recursive factorial:

  ((Y G) 4) $\rightarrow^*$
  (G (Y G) 4) =
  (λf. λn.(**if** n=0 **then** 1 **else** n × (f n-1)) (Y G) 4 $\rightarrow$
  if 4=0 **then** 1 **else** 4 × ((Y G) 4-1) $\rightarrow^*$
  4 × (G (Y G) 4-1) $\rightarrow^*$
  4 × 3 × (G (Y G) 3-1) $\rightarrow^*$
  4 × 3 × 2 × 1 × 1 $\rightarrow^*$
  24

- G is a function of two arguments
  - The first argument is (Y G) for the recursion
  - The second argument is N to do the computation

31

λ calculus and programming languages

32

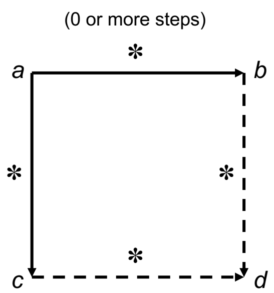
λ calculus and programming languages

- Functional programming languages are just syntactic sugar for lambda calculus
  - Procedure values (lexically scoped closures) are lambda functions
  - Some languages: Haskell, Scheme, OCaml, Oz, Scala
- Languages can use different reduction strategies
  - Reduction strategy = in what order an expression is reduced
  - All reduction strategies give the same result (if they terminate!)
    - Church-Rosser theorem: important theoretical result for λ calculus
  - It's possible for some reduction strategies to give errors or infinite loops
    - Choosing a good reduction strategy is important!
- The two main possibilities are lazy and eager evaluation
  - Lazy evaluation: only evaluate arguments if they are needed
  - Eager evaluation: always evaluate the arguments

33

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



34

Eager evaluation

- Evaluate arguments before the function
  - Innermost first (innermost = arguments)

- **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

Many popular languages use eager evaluation (Java, Python, C++, etc.)

35

Normal order (a form of lazy)

- Evaluate function before the arguments
  - Outermost first (outermost = function)

- **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

In contrast to eager evaluation, normal order only evaluates a function if its result is needed for the computation

36

Lazy evaluation

- Like normal order but shares functions
  - The Average function is only evaluated once

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

  {Double {Average 5 7}} →
  local X={Average 5 7} **in** X+X **end** →
  local X=((5+7)/2) **in** X+X **end** →
  local X=(12/2) **in** X+X **end** →
  local X=6 **in** X+X **end** →
  6+6 →
  12

The functional language Haskell uses lazy evaluation by default; some functional languages use eager evaluation by default but allow declaring lazy evaluation (OCaml, Oz)

37

If statement evaluation

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

- Lazy evaluation:

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

- Eager evaluation:

  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 evaluate **if** statements with eager evaluation for the condition and with lazy evaluation for the **then** and **else** parts

38

Summary of reduction strategies

- Reduction strategies define in what order expressions are computed
- Applicative order: leftmost innermost first (arguments before function)
  - Call by value (eager evaluation): similar to applicative order, but no reductions inside function definitions (compiled functions are not changed)
    - Traditional languages (Java, Python, C++, etc.)
    - Eager functional languages (Scheme, Oz, OCaml, …)
  - Deterministic dataflow (concurrency): execution with threads where each thread does eager evaluation

LINFO1104

- Normal order: leftmost outermost first (function before arguments)
  - Call by name: similar to normal order except no reductions inside function definitions (compiled functions are not changed)
  - Call by need (lazy evaluation): similar to call by name except that functions are shared, not copied (evaluated at most once)
    - Lazy functional languages (Haskell)
    - Declared lazy evaluation (Oz, OCaml)

LINFO1131

39

Variations and extensions

- Lambda calculus is a fundamental part of theoretical computer science research
- Theoretical work on programming languages defines many extensions of the lambda calculus:
  - Typed lambda calculus: lambda calculus with typed variables and functions
  - System F: typed lambda calculus with type variables (variables ranging over types, not the same as "typed variables"!)
  - 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, uses S, K, and I combinators
  - Oz kernel language: a lambda calculus with dataflow variables (single assignment), dataflow execution, threads, and explicit lazy evaluation

40

Conclusions

41

Conclusions

- 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 was introduced by Alonzo Church (1936)
  - Its relationship to programming was first recognized by Peter Landin (1965)
    - This is why languages from the 1950s (Fortran and Lisp) did not do functions right!
- The lambda calculus has strong properties
  - Lambda calculus is Turing complete (all computations can be expressed)
  - Church-Rosser theorem: The result of a computation is independent of the reduction strategy (this is also known as confluence)
    - Important reduction strategies are eager evaluation, lazy evaluation, and dataflow concurrency
  - Higher-order programming is based on the lambda calculus
    - It is the foundation of data abstraction (objects, classes, ADTs, components, agents, etc.)
  - Functional programming languages (Haskell, Scheme, Scala, OCaml, Oz, etc.) are designed to take advantage of these properties

42