λ

Lambda Calculus - Part III

Programmazione Funzionale
2023/2024
Università di Trento
Chiara Di Francescomarino

Today

- Recap
- Encodings
- Recursion

Recap

Beta-reduction

- Computation in the lambda calculus takes the form of beta-reduction

$$(\lambda x. e_1)e_2 \rightarrow e_1[e_2/x]$$

where $e_1[e_2/x]$ denotes the result of substituting e_2 for all free occurrences of x in e_1.

- A term of the form $(\lambda x. e_1)e_2$ (that is an application with an abstraction on the left) is called beta-redex (or β-redex).
- A (beta) normal form is a term containing no beta-redexes

Substitution

- $e_1[e_2/x]$: in expression e_1, replace every occurrence of x by e_2
- The result of the substitution is written with $\mapsto$
- A simple example

$$(\lambda x.\, x\ y\ x)\ z \ \mapsto z\ y\ z$$

- Three cases – the expression e_1 is a(n):
  1. value
  2. application and
  3. abstraction

1. substitution in case of a value

- In $(\lambda x. e_1)e_2 \mapsto e_1[e_2/x]$, where e_1 is a value
  - If $e_1 = x$, $x[e_2/x] = e_2$
  - If $e_1 = y (\neq x)$, $y[e_2/x] = y$

2. Substitution in case of application

- In $(\lambda x. e_1)e_2 \mapsto e_1[e_2/x]$, where e_1 is an application $e_{11}e_{12}$

$(e_{11}e_{12})[e_2/x]$=$(e_{11}[e_2/x]e_{12}[e_2/x])$

3. substitution in case of abstraction

- In $(\lambda x. e_1)e_2 \mapsto e_1[e_2/x]$, where e_1 is an abstraction $\lambda y. e$
  - If $y \neq x$ and $y \notin F_v(e_2)$, then
    $$(\lambda y. e)[e_2/x] = \lambda y. e[e_2/x]$$
  - If $y = x$, then
    $$(\lambda y. e)[e_2/x] = \lambda y. e$$
  - What happens instead if $y \in F_v(e_2)$?
    - We need to be careful!
    - We have to rename the name of the formal parameter (so that it does not depend anymore on e_2). Indeed:
      - $\lambda y. y = \lambda z. z$
      - $\lambda y. e = \lambda z. (e[z/y])$

There is no effect of the substitution

Equivalence

- Given two expressions e_1 and e_2, when should they be considered to be equivalent?
  - Natural answer: when they differ only in the names of the bound variables
- If y is not present in e,

$$\lambda x.\, e \equiv \lambda y.\, e[y/x]$$

- This is called α −equivalence
- Two expressions are α −equivalent if one can be obtained from the other by replacing part of one by an α −equivalent one

Termination

- β-reductions may terminate in a normal form
- Or they may run forever

$$(\lambda x.\, xx)(\lambda x.\, xx) \ \mapsto_\beta \ (xx)([(\lambda x.\, xx)/x])$$
$$= (\lambda x.\, xx)(\lambda x.\, xx)$$

- This is similar to infinite recursion or infinite loops

Confluence

- Basic theorem

  If e can be reduced to e_1 by a β-reduction and e can be reduced to e_2 by a β-reduction, then there exists an e_3 such that both e_1 and e_2 can be reduced to e_3 by β-reductions

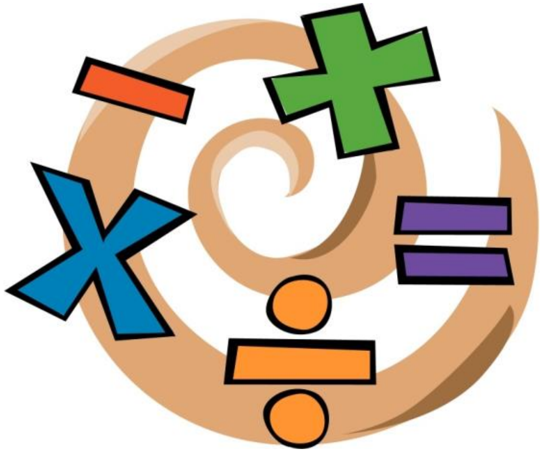
- This means that, if e can be reduced to a normal form, the order of the reductions does not matter

The λ-calculus

- We have seen at the beginning a version of λ-calculus including constants (0,1,2) and functions (+,*)
- The pure λ-calculus, however, seems to be a very limited language
  - Expressions: Only variables, application and abstraction
  - For example, $\lambda x.x + 2$ should be invalid, since 2 is not a variable
- Despite this, the λ-calculus is very expressive
  - It is Turing-complete: Any computation can be expressed in the λ-calculus
  - We can encode any computations …
  - booleans, pairs, constants and arithmetic can be expressed

Booleans

- $true = \lambda x. \lambda y. x$
- $false = \lambda x. \lambda y. y$
- If a then b else c = a b c
- Other Booleans operations
  - not = $\lambda x. x\ false\ true$
    - not x = if x then false else true
    - not true $\rightarrow (\lambda x. x\ false\ true) true \rightarrow (true\ false\ true) \rightarrow false$
  - and = $\lambda x. \lambda y. x\ y\ false$
    - and x y = if x then y else false
  - or = $\lambda x. \lambda y. x\ true\ y$
    - or x y = if x then true else y
  - xor = $\lambda x. x\ (\lambda y. y\ false\ true)\ y$
    - xor x y = if x then not y else y



Encodings

Pairs

- Encoding of a pair (a,b)
  - (a,b) = $\lambda x.if\ x\ then\ a\ else\ b$
  - fst $=\ \lambda f.f\ true$
  - snd = $\lambda f.f\ false$

- Examples
  - fst(a,b) = $(\lambda f.f\ true)(\lambda x.if\ x\ then\ a\ else\ b) \rightarrow$ $(\lambda x.if\ x\ then\ a\ else\ b)\ true \rightarrow if\ true\ then\ a\ else\ b$ $\rightarrow a$
  - snd(a,b) = $(\lambda f.f\ false)(\lambda x.if\ x\ then\ a\ else\ b) \rightarrow$ $(\lambda x.if\ x\ then\ a\ else\ b)\ false \rightarrow if\ false\ then\ a\ else\ b$ $\rightarrow b$

Coding natural numbers

- We base this on the Peano axioms:
  - 0 is a natural number
  - If n is a natural number, so is the successor of n, `succ(n)`
- Church's idea
  - 0 is coded as $\lambda f. \lambda x. x$
  - Intuitively, `f` applied 0 times to `x`
  - `succ(n)`: apply `f` to `x` `n` times

Natural numbers

- n is represented by the higher-order function that maps any function f to its n-fold composition
- In other words, the “value" of the numeral n is equivalent to the number of times the function is applied to its argument.
- More formally

$$f^n = \underbrace{f \circ f \circ \cdots \circ f}_{n\ times}$$

- That is $n = \lambda f. \lambda x.$<apply f n times to x>

Successor

- We write $n\ f$ to mean "apply $f\ n$ times"
- Then, n is $\lambda f.\lambda x.n\ f\ x$
- We define
$$succ(n) = \lambda n.\lambda f.\lambda x.f(n\ f\ x)$$
- Applied to the λ-definition of n, it should give us the λ-definition of $n + 1$
- $n + 1\ is\ \lambda f.\lambda x.f\ (n\ f\ x)$
- Every Church numeral is a function that takes two parameters

Natural numbers: function definition

Number	Function definition	Lambda-expression
0	$0\ f\ x =\ x$	$\lambda f.\lambda x.x$
1	$1\ f\ x =\ f\ x$	$\lambda f.\lambda x.f\ x$
2	$2\ f\ x =\ f(f\ x)$	$\lambda f.\lambda x.f(f\ x)$
3	$3\ f\ x =\ f(f(f\ x))$	$\lambda f.\lambda x.f(f(f\ x))$
…	…	
n	$n\ f\ x =\ f^n\ x$	$\lambda f.\lambda x.f^n\ x$

Church numeral example

- The Church numeral 3 represents the action of applying any given function three times to a value
- The function is first applied to the parameter and then successively to its own result
- If the function is the successor function, and the parameter is 0, the result is the numeral 3
- But note that the function itself, and not the result, is the Church numeral 3, which means simply to do anything three times

Question 9

What ML type can we give to a Church-encoded numeral?

A. (’a -> ‘b) -> ‘a -> ‘b
B. (‘a -> ‘a) -> ‘a ->’a
C. (‘a -> ‘a) -> ‘b -> int
D. (int -> int) -> int -> int

$n = \lambda f. \lambda x.$ <apply f n times to x>

Answer question 9

What ML type can we give to a Church-encoded numeral?

A. (’a -> ‘b) -> ‘a -> ‘b
B. (‘a -> ‘a) -> ‘a ->’a
C. (‘a -> ‘a) -> ‘b -> int
D. (int -> int) -> int -> int

$n = \lambda f . \lambda x .$ <apply f n times to x>

Let's have a look at 1= $succ(0)$

$$succ(0) =$$
$$(\lambda n.\lambda f.\lambda x.f(n\ f\ x))(\lambda f.\lambda x.x) \mapsto$$
$$\left(\lambda f.\lambda x.f\left((\lambda f.\lambda x.x)f\ x\right)\right) \mapsto$$
$$\left(\lambda f.\lambda x.f\left((\lambda x.x)\ x\right)\right) \mapsto$$
$$\lambda f.\lambda x.fx =$$
$$1$$

Let's have a look at $2 = succ(1)$

$succ(1) =$
$(\lambda n.\lambda f.\lambda x.f(n\ f\ x))(\lambda f.(\lambda x.fx)) \mapsto$

$(\lambda f.\lambda x.f((\lambda f.(\lambda x.fx))f\ x)) \mapsto$
$(\lambda f.\lambda x.f\big((\lambda x.fx)\ x\big)) \mapsto$

$(\lambda f.\lambda x.f(f\ x)) =$
2

In a similar way, $3\ =\ succ(2) =$
$\lambda f.\lambda x.f\big(f(fx)\big)$, …

Operations on Church numerals

- Iszero?
  - iszero = $\lambda z.z(\lambda y.false)true$
- Example
  - Iszero 0 =

$$(\lambda z.z(\lambda y.false)true)(\lambda f.\lambda x.x) \rightarrow$$
$$\big((\lambda f.\lambda x.x)(\lambda y.false)true\big) \rightarrow$$
$$\big((\lambda x.x)\ true\big) \rightarrow true$$

Addition

- n means: "f applied n times to x"
- So $2 + 3$ means: "apply f twice to the result of applying f three times to x"
- $n + m$: Apply f n times to m
- How to do this?
  - "Body" of m is mfx
  - Substitute the body of m in the body of n in in the place of x, i.e., $nf(mfx)$
- This gives us $\lambda n. \lambda m. \lambda f. \lambda x. nf(mfx)$

Let's see 2+3

$2+3=$

$(\lambda n.\lambda m.\lambda f.\lambda y.nf(mfy))(\lambda f.\lambda x.f(fx))\left(\lambda f.\lambda x.f\big(f(fx)\big)\right) \mapsto$

$(\lambda m.\lambda f.\lambda y.\big(\lambda f.\lambda x.f(fx)\big)f(mfy))\left(\lambda f.\lambda x.f\big(f(fx)\big)\right) \mapsto$

$(\lambda m.\lambda f.\lambda y.\lambda x.f(fx)(mfy))\left(\lambda f.\lambda x.f\big(f(fx)\big)\right) \mapsto$

$(\lambda f.\lambda y.\lambda x.f(fx)\left(\left(\lambda f.\lambda x.f\big(f(fx)\big)\right)fy\right)) \mapsto$

$(\lambda f.\lambda y.\lambda x.f(fx)\left(\left(\lambda x.f\big(f(fx)\big)\right)y\right)) \mapsto$

$(\lambda f.\lambda y.\lambda x.f(fx)\left(f\big(f(fy)\big)\right)) \mapsto$

$(\lambda f.\lambda y.f\left(f\left(f\big(f(fy)\big)\right)\right)) = 5$

- We have proved that $2+3=5$

Exercise 7.7

- Prove the following
  - $1 + 0 = 1$

Solution exercise 7.7

- Prove the following
  - $1 + 0 = 1$

$(\lambda n.\lambda m.\lambda f.\lambda y.nf(mfy))(\lambda f.\lambda x.fx)(\lambda f.\lambda x.x) \mapsto$

$(\lambda m.\lambda f.\lambda y.(\lambda f.\lambda x.fx)f(mfy))(\lambda f.\lambda x.x) \mapsto$

$(\lambda m.\lambda f.\lambda y.(\lambda x.fx)(mfy))\ (\lambda f.\lambda x.x) \mapsto$

$(\lambda f.\lambda y.(\lambda x.fx)\big((\lambda f.\lambda x.x)fy\big) \mapsto$

$(\lambda f.\lambda y.(\lambda x.fx)\big((\lambda x.x)y\big)) \mapsto$

$(\lambda f.\lambda y.(\lambda x.fx)y) \mapsto$

$\lambda f.\lambda y.fy = 1$

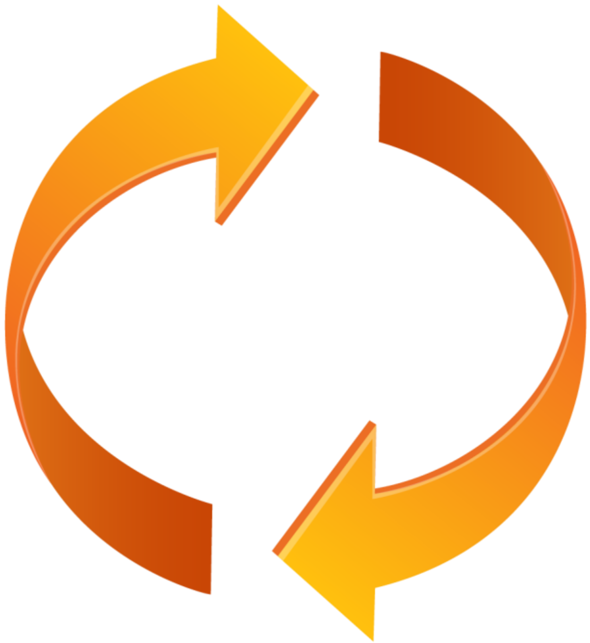
General remarks

- The notation can be very complicated (as in 2 + 3 = 5)
- Note that $\lambda x. x + 2$ is not a valid expression in the pure λ-calculus
  - but the following, equivalent expression, is

$$\lambda x.((\lambda n.\lambda m.\lambda f.\lambda x.(nf(mfx))x(\lambda f.\lambda x.f(f(x)))$$

Extensions of λ-calculus

- Slight abuse of notation: allow the use of numbers, operations and expressions
- We therefore allow expressions such as $\lambda x.(x\ +\ 2)$ or $\lambda x.\mathtt{if}\ x\ =\ 1\ \mathtt{then\ x\ else\ (x+2)}$
- These are used as abbreviations of expressions in the “pure" λ-calculus



Recursion

Recursion in λ-calculus

- We claimed that Lambda-calculus is powerful
- We saw how to define expressions:
  - Booleans and their operations
  - Pairs
  - Numbers and their operations

Recursion

- How to implement recursion in the λ-calculus?
  - Functional paradigm: using recursion
  - But how do we implement recursion?
- We cannot give a name to λx … ., but have to implement recursion using only abstraction and application
- Trivial example

```
fun f n = if n=0 then 1 else n*f(n-1);
```

- What is this function?

Implementing recursion

- Suppose we want to write the factorial function which takes a number n and computes n!

```
λn.if (n=0) then 1 else (n *(f (n-1)))
```

- This does not work. Because what is the unbound variable f?
- It would work if we could somehow make f be the function above

Eliminating recursion

- To give access to the function f, what about passing f as another parameter?
- Consider $f \ = \ \lambda n.\texttt{if} \ \ n = 0 \ \texttt{then 1 else n*f(n-1)}$ as a definition, not as an equation
- Making f a parameter, we get

$$\lambda f.\,\lambda n.\,if\ n = 0\ then\ 1\ else\ n * f(n-1)$$

- We have then eliminated the recursion

Recursion

- We can write the function as

  $G = \lambda f. \lambda n.$ `if n=0 then 1 else n * f(n-1)`

- In other words, we look for $f = G(f)$ where G is a higher-order function which takes a function as argument, and returns a function
- "Solving" this equation gives us f
- This means solving $h =_\beta Gh$
- In ML, this is equivalent to define

  ```
  fun g f n = if n=0 then 1 else n*f(n-1);
  ```

- But how do we solve this problem?

Y

The Y-combinator

The general problem

- Given a function G, find f such that $f \ =_\beta \ Gf$
- This means to find a fixpoint of the operator G
- The Y combinator is one of these fixpoints

$$Y \ = \ \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

The general problem

- We started from a function `fact`:

  $\lambda n.$if $n = 0$ `then 1 else n*f(n-1)`

- We wrote a function `ps_fact` G, which is no longer recursive

  $G = \lambda f.\lambda n.$ `if n=0 then 1 else n * f(n-1)`

- If we can pass to G this same logic (if n > 0 …) as `f`, then we have done
- This is what Y does!
- By applying the Y combinator to the pseudo-recursive function, we obtain our factorial function `fact`:

  `Y ps_fact = fact`

The Y combinator

$Y\ e =$

$(\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))e \mapsto$

$(\lambda x.e(xx))(\lambda x.e(xx)) \mapsto$

$e(\lambda x.e(xx))(\lambda x.e(xx)) =_\beta e(Y\ e)$

- Therefore, $Y\ e\ =\ e(Y\ e)$ and so $YG = G(YG)$, i.e., YG is a fixpoint for G
  - We can use Y to achieve recursion for G

Example

- `ps_fact` =
  $$\lambda f.\lambda n.if\ n\ =\ 0\ then\ 1\ else\ n\ *\ (f\ (n-1))$$
- The second argument of `ps_fact` is the integer
- The first argument is the function to call in the body
  - We'll use Y to make this recursively call fact

$$(Y\ ps\_fact)1\ =\ \big(ps\_fact\ (Y\ ps\_fact)\big)1\ \rightarrow$$
$$if\ 1\ =\ 0\ then\ 1\ else\ 1\ *\ ((Y\ ps\_fact)\ 0)\ \rightarrow$$
$$1\ *\ ((Y\ ps\_fact)\ 0)\ =$$
$$1\ *\ (ps\_fact\ (Y\ ps\_fact)\ 0)\ \rightarrow$$
$$1\ *\ (if\ 0\ =\ 0\ then\ 1\ else\ 0\ *\ ((Y\ ps\_fact)\ (-1))\ \rightarrow$$
$$1\ *\ 1\ \rightarrow\ 1$$

Exercise 7.8

- Reduce to normal form
  - $(\lambda x.\, yx)((\lambda y.\, \lambda t.\, yt)zx)$

Solution exercise 7.8

- Reduce to normal form
  - $(\lambda x.yx)\big((\lambda y.\lambda t.yt)zx\big)$

$$
\begin{aligned}
&(\lambda x.yx)\big((\lambda y.\lambda t.yt)zx\big) \mapsto \\
&(\lambda x.yx)((\lambda t.zt)x) \mapsto \\
&(\lambda x.yx)(zx) \mapsto \\
&y(zx)
\end{aligned}
$$

Exercise 7.9

- Reduce to normal form
  - $(\lambda x. xzx)((\lambda y. yyx)z)$

Solution exercise 7.9

- Reduce to normal form
  - $(\lambda x.\, xzx)((\lambda y.\, yyx)z)$

$$(\lambda x.\, xzx)((\lambda y.\, yyx)z) \mapsto$$
$$(\lambda x.\, xzx)(zzx) \mapsto$$
$$(zzx)z(zzx)$$

Exercise 7.10

- Reduce to normal form
  - $(\lambda x.xy)(\lambda t.tz)((\lambda x.\lambda z.xyz)yx)$

Solution exercise 7.10

- Reduce to normal form
  - $(\lambda x.xy)(\lambda t.tz)((\lambda x.\lambda z.xyz)yx)$

$$(\lambda x.xy)(\lambda t.tz)((\lambda x.\lambda z.xyz)yx) \mapsto$$
$$(\lambda x.xy)(\lambda t.tz)((\lambda z.yyz)x) \mapsto$$
$$(\lambda x.xy)(\lambda t.tz)(yyx) \mapsto$$
$$((\lambda t.tz)y)(yyx) \mapsto$$
$$(yz)(yyx)$$

Exercise 7.11

- Prove the following
  - $0 + 1 = 1$

Solution exercise 7.11

- Prove the following
  - $0 + 1 = 1$

$(\lambda n.\lambda m.\lambda f.\lambda y.nf(mfy))(\lambda f.\lambda x.x)(\lambda f.\lambda x.fx) \mapsto$
$(\lambda m.\lambda f.\lambda y.(\lambda f.\lambda x.x)f(mfy))\ (\lambda f.\lambda x.fx) \mapsto$
$(\lambda m.\lambda f.\lambda y.(\lambda x.x)(mfy))(\lambda f.\lambda x.fx) \mapsto$
$(\lambda m.\lambda f.\lambda y.(mfy))\ (\lambda f.\lambda x.fx) \mapsto$
$(\lambda f.\lambda y.\big((\lambda f.\lambda x.fx)fy\big)) \mapsto$
$(\lambda f.\lambda y.(\lambda x.fx)y) \mapsto$
$(\lambda f.\lambda y.y) \mapsto$
$= 1$

Exercise 7.12

- Prove the following
  - $1 + 1 = 1$

Solution exercise 7.12

- Prove the following
  - $1 + 1 = 1$

$(\lambda n.\lambda m.\lambda f.\lambda y.nf(mfy))(\lambda f.\lambda x.fx)(\lambda f.\lambda x.fx) \mapsto$
$(\lambda m.\lambda f.\lambda y.(\lambda f.\lambda x.fx)f(mfy))\ (\lambda f.\lambda x.fx) \mapsto$
$(\lambda m.\lambda f.\lambda y.(\lambda x.fx)(mfy))\ (\lambda f.\lambda x.fx) \mapsto$
$(\lambda m.\lambda f.\lambda y.f(mfy))\ (\lambda f.\lambda x.fx) \mapsto$
$(\lambda f.\lambda y.f\big((\lambda f.\lambda x.fx)fy\big))\ \mapsto$
$(\lambda f.\lambda y.f\big((\lambda x.fx)y\big))\ \mapsto$
$\lambda f.\lambda y.f(fy)$
$= 2$

Exercise 7.13 (*)

- Prove the following
  - $3 + 2 = 5$

Solution exercise 7.13

- Prove the following
  - $3 + 2 = 5$

$(\lambda n.\lambda m.\lambda f.\lambda y.nf(mfy))\big(\lambda f.\lambda x.f(f(f(x)))\big)(\lambda f.\lambda x.f(f(x))) \mapsto$

$(\lambda m.\lambda f.\lambda y.\big(\lambda f.\lambda x.f(f(f(x)))\big)f(mfy))(\lambda f.\lambda x.f(f(x))) \mapsto$

$(\lambda m.\lambda f.\lambda y.\big(\lambda x.f(f(f(x)))\big)(mfy))(\lambda f.\lambda x.f(f(x))) \mapsto$

$(\lambda m.\lambda f.\lambda y.f(f(f(mfy))))(\lambda f.\lambda x.f(f(x))) \mapsto$

$(\lambda f.\lambda y.f\big(f(f(\lambda x.f(f(x))y))\big)) \mapsto$

$(\lambda f.\lambda y.f(f(f(f(f(y))))))$

$\lambda f.\lambda x.f(f(f(f(f(x)))))$

$= 5$

Summary

- Encodings
- Recursion

Readings

- Chapter 11 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill
- Few slides from the University of Maryland

Next time

- Scala