λ

Lambda Calculus - Part II

Programmazione Funzionale

2024/2025

Università di Trento

Chiara Di Francescomarino

Mini-challenge on Thursday

- ML Challenge on Thursday May 15th 10:30-12:30 during the laboratory class.
- Please register your group in the form by today (23:59) https://docs.google.com/forms/d/e/1FAIpQLSdjVknRl1y4hB3ojIUzkFcDr6TRkzR8RarMMitvfRTazIZZjQ/viewform
    - Groups can be at most composed of three students
    - For those of you who cannot attend next Thu lecture, you can participate to the mini challenge *alone* and you will have time until 23:59 of May 15th
- Please be aware that you cannot use chatgpt (or similar) to solve the mini challenge exercise.

When you have time

You can find the link also in Moodle!

Join this Wooclap event

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
DQDQRX

Today

- Beta-reductions
- Encodings
- Recursion

β

Beta-reduction

Lambda expression evaluation

The intuition

- Consider this lambda expression:

$$(\lambda x. x + 1)4$$

It means that we apply the lambda abstraction to the argument 4, as if we apply the increment function to the argument 4.

- How do we do it?

  The result of applying a lambda abstraction to an argument is an instance of the body of the lambda abstraction in which bound occurrences of the formal parameter in the body are replaced with copies of the argument.

- This means: $(\lambda x. x + 1)4 \underset{\beta}{\rightarrow} 4 + 1$

β-reduction examples

- $(\lambda x. x + x)5 \rightarrow 5 + 5 \longrightarrow 10$
- $(\lambda x. 3)5 \rightarrow 3$

Parameters

- formal
- formal occurrence
- actual

It looks like we instantiate the formal parameter (i.e., the occurrences of the bound variable) with the actual parameter (the expression to which we are applying the function)

Higher-Order Functions

- Beta-reductions can be applied with higher-order functions
- For instance, let us consider a function that, given a function f, return function f ∘ f

  $\lambda f.\ \lambda x.\ f\ (f\ x)$
- How does this work? Let us apply it to the successor function

$$(\lambda f.\ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+1) \mapsto_\beta$$

$$\lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ \ x) \mapsto_\beta$$

$$\lambda x.\ (\lambda y.\ y+1)\ (x+1) \mapsto_\beta$$

$$\lambda x.\ (x+1)+1$$

Same result if executing first the first λy

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* 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 $e_1 = y(\neq x), y[e_2/x] = y$

Identity function

Constant function

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
    $$(\lambda y.e)[e_2/x] = \lambda y.e$$

There is no effect of the substitution

  - What happens instead if $y \in F_v(e_2)$?
    - We need to be careful!

Variable capture

- What happens when $y \in F_v(e_2)$?
- For instance what happens with $(\lambda x.\lambda y.x\ y)y$?
- When we replace y inside the expression, we do not want to be captured by the inner binding of y (it would violate the static scoping), that is, if we apply $(\lambda y.e)[e_2/x]$=$\lambda y.e[e_2/x]$, we would get
  $(\lambda y.x\ y)[y/x] \longmapsto \lambda y.(x\ y[y/x]) = \lambda y.yy$ but
  $(\lambda x.\lambda y.x\ y)y \neq \lambda y.yy$
- Solution: rename y in v, that is change $\lambda y.x\ y$ to $\lambda v.x\ v$

$$(\lambda v.x\ v)[y/x] \longmapsto \lambda v.(x\ v[y/x]) = \lambda v.yv$$

An example

```
int x=0;
int foo (name int y){
     int x = 2;
     return x + y;
}
...
int a = foo(x+1);
```

- Blindly applying the copy rule would lead us to a result of x+x+1=5
- Incorrect result as it would depend on the name of the local variable
- With a body `{int z = 2; return z + y;}` the result would have been z+x+1=3

- When the body contains the same name of the actual parameter, we say that it is captured by the local declaration
- In order to avoid substitutions in which the actual parameter is captured by the local declaration, we impose that the formal parameter – even after the substitution – is evaluated in the environment of the caller and not of the callee

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

α-Conversion

- α-conversion can be used to avoid having variable capture during substitution
- Examples

$$\lambda x. x =_\alpha \lambda y. y$$
$$\lambda x. xy =_\alpha \lambda z. zy$$

- But NOT

$$\lambda y. xy =_\alpha \lambda y. zy$$

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

Let’s try a test

Join this Wooclap event

1. Go to wooclap.com
2. Enter the event code in the top banner

Event code
PCEXEL

Few rules/guidelines … to remember for β-reduction

1. Associativity of applications is on the left: $M\ N\ L\ \equiv\ (M\ N)\ L$
2. The body of a lambda extends as far as possible to the right, e.g.,
   $\lambda x.\, x\ \lambda z.\, x\ z\ x$ corresponds to $\lambda x.\left(x\ \lambda z.\,(x\ z\ x)\right)$ and not to $\cancel{(\lambda x.\, x)\ (\lambda z.\,(x\ z\ x))}$
3. Consider the precedence rules imposed by parentheses – when they are used
4. Otherwise, precedence is given to the leftmost and innermost precedence, e.g.,
   $\left((\lambda x.\, x)x\right)(\lambda x.\, xy) \mapsto x(\lambda x.\, xy)$, while $\left((\lambda x.\, x)x\right)(\lambda x.\, xy) \mapsto \cancel{(\lambda x.\, xy)x}$ is incorrect!

Few rules/guidelines … to remember for β-reduction

5. Be careful when a variable is captured (i.e., when a free variable becomes bound): this is an error! E.g.,

   $(\lambda y.(\lambda x.yx))x \nrightarrow (\lambda x.xx)$ as the free variable y becomes bound after the application … we need to rename the bound x with a different name, e.g., t: $(\lambda y.(\lambda t.yt))x$, so as to avoid that variables are captured

You can find lambda functions …

- In ML

```
val square = fn x => x*x;
```

- In Python:

```
square = lambda x: x*x
```

Same Procedure (ML)

- Given function f, return function f ∘ f

```
fn f => fn x => f(f(x));
val it = fn: ('a -> 'a) -> 'a -> 'a
```

- How does this work?

```
(fn f => fn x => f(f(x))) (fn y => y + 1)

= fn x => ((fn y => y + 1) ((fn y => y + 1) x))

= fn x => ((fn y => y + 1) (x + 1))

= fn x => ((x + 1) + 1)
```

Same Procedure (JavaScript)

- Given function f, return function f ∘ f

```
function (f) { return function (x) { return f(f(x)); } ; }
```

- How does this work?

```
(function (f) { return function (x) { return f(f(x)); } ;  })
                    (function (y) { return y + 1; })


function (x) { return (function (y) { return y + 1; })
                            ((function (y) { return y + 1; }) (x)); }


function (x) { return (function (y) { return y +1; }) (x + 1); }


function (x) { return ((x + 1) + 1); }
```

Same Procedure (Python)

- Given function f, return function f ∘ f

```
def g(x): return (lambda f,x: f(f(x)))(lambda y:y+1,x)
```

- How does this work?

```
def g(x): return (lambda f,x: f(f(x)))(lambda y:y+1,x)

def g(x): return (lambda y:y+1,(lambda y:y+1,x))

def g(x): return (lambda y:y+1,(x + 1))

def g(x): return ((x + 1) + 1)
```

β −reductions

- β-reductions are not symmetric
- $e_1 \mapsto_\beta e_2$ does not imply $e_2 \mapsto_\beta e_1$
  - So this is not an equivalence relation
  - A notion of β-equivalence can be defined as the reflexive and transitive closure of $\mapsto_\beta$

Normal form

- Expressions with no redex, have no β-reductions
  - This is called normal form
  - $\lambda x. \lambda y. x$ is in normal form
  - $\lambda x. ((\lambda y. y)x)$ is not in normal form
  - $(\lambda y. y)x \mapsto_\beta x$ and therefore $\lambda x. (\lambda y. y)x \mapsto_\beta \lambda x. x$

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

Exercise 10.4

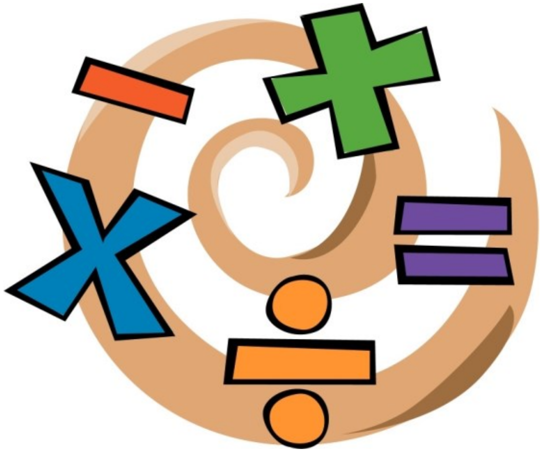
- Reduce to normal form
  - $(\lambda x. x(xy))(\lambda z. zx)$

Exercise 10.5

- Reduce to normal form
  - $(\lambda x. xy)(\lambda z. zx)(\lambda z. zx)$

Exercise 10.6

- Reduce to normal form
  - $(\lambda t. tx)((\lambda z. xz)(xz))$



Encodings

The λ-calculus

- We have seen so far 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

- Examples
  - If true then b else c = $(\lambda x. \lambda y. x) b\ c \rightarrow (\lambda y. b) c \rightarrow b$
  - If false then b else c = $(\lambda x. \lambda y. y) b\ c \rightarrow (\lambda y. y) c \rightarrow c$

Booleans

- 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
- Given these operations
  - Can build up a logical inference system

Let’s try a test

Join this Wooclap event

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
PCEXEL

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)`
- The following idea is by Church
  - 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

$$succ(n) = \lambda n.\lambda f.\lambda x.f(n\ f\ x)$$

- Why is this the successor function?
- Applied to the λ-definition of n, it should give us the λ-definition of $n + 1$
- Formally: $n + 1 = \lambda f.\lambda x.f\ (n\ f\ x)$
- 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 numerals

- 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

Let’s try a test

Join this Wooclap event

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
PCEXEL

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

$succ(0) =$

$\left(\lambda n.\lambda f.\lambda x.f(n\ f\ x)\right)(\lambda f.\lambda x.x) \mapsto$

$\left(\lambda f.\lambda x.f\left((\lambda f.\lambda x.x)f\ x\right)\right) \mapsto$

$\left(\lambda f.\lambda x.f\left((\lambda x.x)\ x\right)\right) \mapsto$

$\lambda f.\lambda x.fx =$

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

$2 + 3 =$

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

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

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

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

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

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

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

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

Exercise 10.7

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

Exercise 10.8

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

Exercise 10.9

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

Exercise 10.10

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

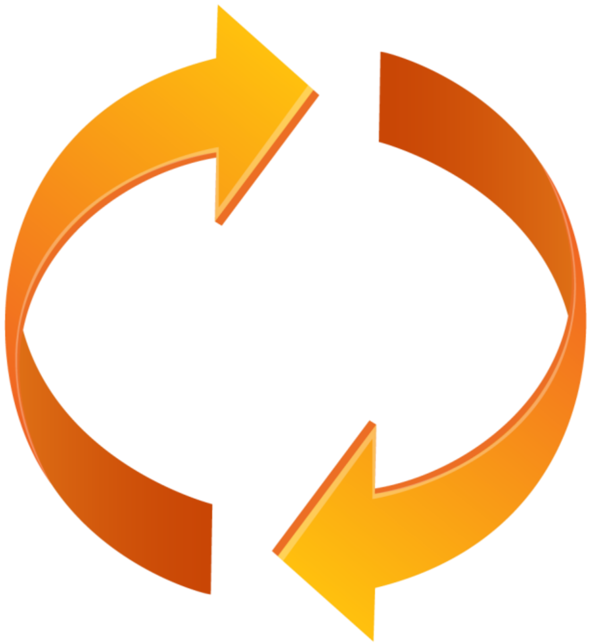
General remarks

- 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 $\lambda x.\mathtt{if}\ x\ =\ 1\ \mathtt{then}$
- 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?
- 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
- G is a function that if we give it a function f able to compute the next step, then it returns the factorial function, that is G is a description of the factorial function but we need the application
- 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 way to compute such a fixpoint

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

- The Y combinator is the solution to our problem: it is a function that applied to G returns the function f we were looking for, that is Y is a function that allows us to call again G

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)`
- We need a function that allows us to compute the fixpoint
- 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`
- `ps_fact` describes what the recursion does (given the next step), while `Y ps_fact` is the application of the recursive function, that is the factorial function

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 10.11

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

Exercise 10.12

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

Exercise 10.13

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

Summary

- Beta-reductions
- Encodings
- Recursion

Next time

- Logic Programming