

# $\lambda$ calculus, formally

Assume given an infinite set  $\mathcal{V}$  of variables, denoted by  $x, y, z$ , etc. The set of lambda terms  $\Lambda$  is given by

Lambda terms:  $M, N ::= x \mid (MN) \mid (\lambda x. M)$

The following are some examples of lambda terms:

$(\lambda x. x)$        $((\lambda x. (xx))(\lambda y. (yy)))$        $(\lambda f. (\lambda x. (f(fx))))$

```

type rec t =
  | Var (string)
  | App (t,t)
  | Fun (string, t)
  
```

# Where is the computation happening ?

- $\beta$ -reduction (function application in an informal sense)
- $\beta$ -redex is a term of the form  $(\lambda x. M)N$
- A  $\beta$ -redex **reduces** to  $M[N/x]$

For example,

$$\begin{aligned}
 (\lambda x. y) \left( \underline{(\lambda z. zz)(\lambda w. w)} \right) &\rightarrow_{\beta} (\lambda x. y) \left( \underline{(\lambda w. w)(\lambda w. w)} \right) \\
 &\rightarrow_{\beta} \underline{(\lambda x. y)(\lambda w. w)} \\
 &\rightarrow_{\beta} y.
 \end{aligned}$$

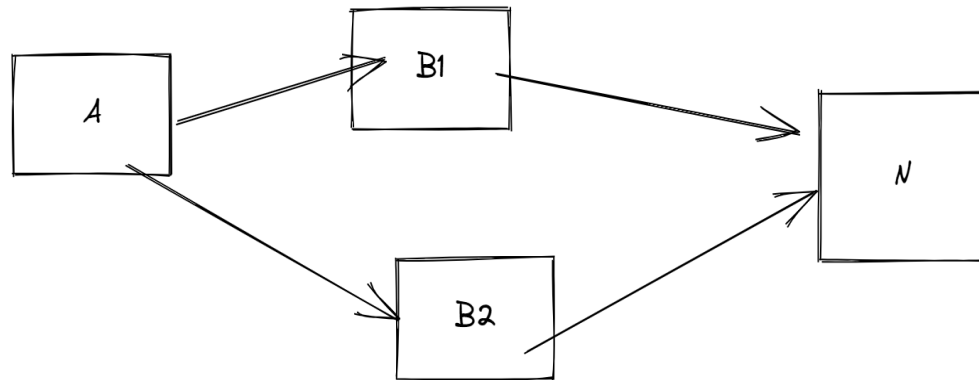
The same term can be reduced differently,

$$\underline{(\lambda x. y) \left( (\lambda z. zz)(\lambda w. w) \right)} \rightarrow_{\beta} y$$

# Confluence of untyped lambda calculus

- Church-Rosser theorem

If there are reduction sequences from any term  $A$  to two different terms  $B_1$  and  $B_2$ , then there exist reduction sequences from those two terms to some common term  $N$ .



- Result of computation is independent of the evaluation order
- Languages can choose different evaluation order, e.g, Haskell, ReScript

# Interpreter (natural semantics)

- Evaluate closed term, call by value

```
let rec eval = (t: lambda) => {
  switch t {
  | Var(_) => assert false
  | Fn(_, _) => t
  | App(f, arg) => {
    let Fn(x, body) = eval(f)
    let va = eval(arg)
    eval(subst(x, va, body)) // substitution explained later
  }
  }
}
```

- Example

# A formal view of the natural semantics

- Left to right

$$\frac{a \rightarrow a'}{a b \rightarrow a' b} \quad \frac{b \rightarrow b'}{v b \rightarrow v b'}$$

- Call by value:

$$\frac{}{(\lambda x. a) v \rightarrow a[v/x]}$$

- What are values?
  - functions
  - can we use functions to represent constant numbers, boolean value, etc?

# Two Interpreters

## Eval using substitution

```
let rec eval = (t: lambda) => {
  switch t {
  | Var(_) => assert false
  | Fn(_, _) => t
  | App(f, arg) => {
    let Fn(x, body) = eval(f)
    let va = eval(arg)
    eval(subst(x, va, body))
  }
}
```

## Eval using env map

```
let eval = (t: lambda) => {
  let rec go = (e, t) => {
    switch t {
    | Var(x) => List.assoc(x, e)
    | Fn(x, body) => Vclosure(e, x, body)
    | App(f, arg) => {
      let Vclosure(e', x, body) = go(e, f)
      let va = go(e, arg)
      go(list{(x, va), ...e'}, body)
    }
  }
  go (list{}, t)
}
```

## Two kinds of interpreters

- Substitution: **eagerly** replace the bound variables with the argument
- Environment: save the argument and **lazily** replace the bound variables
- Evaluate to the equivalent results
- The former used in optimizations, the latter used in efficient interpreters

For lambda terms  $M$  without free variables

$$\text{eval1}(M) = \text{Fn}(x, N) \Leftrightarrow \text{eval2}(M) = \text{Vclosure}([], x, N)$$



# Primitives

- Definition of boolean

$$\text{if\_then\_else } \bar{T} \ M \ N \ \rightarrow_{\beta} \ M$$

$$\text{if\_then\_else } \bar{F} \ M \ N \ \rightarrow_{\beta} \ N$$

- Boolean values

$$\bar{T} \quad = \quad \lambda xy. x$$

$$\bar{F} \quad = \quad \lambda xy. y$$

- If-Then-Else

$$\text{if\_then\_else} = \lambda x. x$$

- Church Pair

$$\text{pair} = \lambda x y z. z x y$$
$$\text{fst} = \lambda p. p (\lambda x y. x)$$
$$\text{snd} = \lambda p. p (\lambda x y. y)$$

# Church numerals

- The Church numerals  $\bar{0}, \bar{1}, \bar{2}, \dots$  are defined by

$$\bar{n} = \lambda f x. f^n(x).$$

- Here are the first few Church numerals:

$$\bar{0} = \lambda f x. x$$

$$\bar{1} = \lambda f x. f x$$

$$\bar{2} = \lambda f x. f(f x)$$

$$\bar{3} = \lambda f x. f(f(f x))$$

...

# Constructors

- Iso-morphism, used in Coq for number theory, called Peano number

```
type rec nat = Z | S(nat)
let three = S (S (S Z))
```

- Compare with the church numeral:  $\bar{3} = \lambda fx. f(f(fx))$
- There is correspondence between constructors  $S$  and  $Z$  and bound variables  $f$  and  $x$
- Recommended reading: Church encoding and Scott encoding

# Arithmetic functions

$$\text{succ} = \lambda n f x. f(n f x)$$

$$\text{add} = \lambda n m f x. n f(m f x)$$

## Example

$$\text{succ } \bar{n} = (\lambda n f x. f(n f x))(\lambda f x. f^n x)$$

$$\rightarrow_{\beta} \lambda f x. f((\lambda f x. f^n x) f x)$$

$$\rightarrow_{\beta} \lambda f x. f(f^n x)$$

$$= \lambda f x. f^{n+1} x$$

$$= \overline{n + 1}$$

## More primitives

- Test whether a number is zero

$$\text{iszero} = \lambda n. n (\lambda z. \bar{F}) \bar{T}$$

- Predecessor (simple version using pair)

$$\text{pred} = \lambda n. \text{fst} (n (\lambda p. \text{pair}(\text{snd } p)(\text{succ}(\text{snd } p))))(\text{pair } \bar{0} \bar{0})$$

# Pred for church numerals

$$f = \lambda p. \text{pair } (\text{second } p) (\text{succ } (\text{second } p))$$
$$\text{zero} = (\lambda f. \lambda x. x)$$
$$\text{pc0} = \text{pair } \text{zero } \text{zero}$$
$$\text{pred} = \lambda n. \text{first } (n \ f \ \text{pc0})$$
$$\begin{aligned} \text{pred three} &= \text{first } (f \ (f \ (f \ (\text{pair } \text{zero } \text{zero})))) \\ &= \text{first } (f \ (f \ (\text{pair } \text{zero } \text{one}))) \\ &= \text{first } (f \ (\text{pair } \text{one } \text{two})) \\ &= \text{first } (\text{pair } \text{two } \text{three}) \\ &= \text{two} \end{aligned}$$

## How to define multiplication?

Recall that we have  $n + m$ , we want to define  $n \times m$  recursively as

$$(\times) = \lambda n m. \text{if}(n = 0)\text{then } 0 \text{ else } (m + (n - 1) \times m)$$

we replaced `ite`, `iszero`, `add`, `pred` with syntactic sugar



# Recursive function

Note that  $(\times)$  is a free variable on the right-hand side. So we rewrite the term

$$(\times) = (\lambda f n m. \text{if}(n = 0)\text{then } 0 \text{ else } (m + f (n - 1) m))(\times)$$

The right-hand side still contains the free variable  $f$ . But we get a closed term, which we will abbreviate as  $F$ ,

$$F = \lambda f n m. \text{if}(n = 0)\text{then } 0 \text{ else } (m + f (n - 1) m)$$

now we have

$$(\times) = F (\times)$$

Now what can we do with  $F$ ?

# Iteration method

- First attempt: apply  $F$  with  $\perp$

$$F(\perp) = \lambda n m. \text{if}(n = 0)\text{then } 0 \text{ else } (m + \perp)$$

Far away from what we want. But at least it is correct when  $n = 0$

- Next attempt: apply  $F$  with  $F(\perp)$

$$F(F(\perp)) = \lambda n m. \text{if}(n = 0)\text{then } 0 \text{ else } (m + F(\perp) (n - 1) m)$$

$F(\perp)$  is correct when  $n = 0$ , so  $F(F(\perp))$  is correct when  $n = 0$  or  $n = 1$

- Actually,  $F^i(\perp)$  correctly calculates  $n \times m$  for  $n < i$
- how can we iterate this process?

# Infinite reduction

Consider the following term

$$\omega = \lambda x. xx$$

Then we define the  $\Omega$  combinator

$$\Omega = \omega\omega$$

which reduces to itself

$$(\lambda x. xx)(\lambda x. xx) \rightarrow_{\beta} (\lambda x. xx)(\lambda x. xx) \rightarrow_{\beta} \dots$$

- Turing machine can also loop forever

# Y combinator

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

Repeatedly applying this equality gives:

$$Y F =_{\beta} F (Y F) =_{\beta} F (F (Y F)) =_{\beta} F (\dots F (Y F) \dots)$$

$Y F$  is also known as a fixed-point of  $F$

# Y combinator

Then we can define the multiplication function to be

$$(\times) = Y F$$

where

$$F = \lambda f n m. \text{if}(n = 0)\text{then } 0 \text{ else } (m + f (n - 1) m)$$

This is correct, because  $Y$  combinator iterates  $F(F(\dots))$  infinitely many times

Actually, applying the property  $Y F = F (Y F)$

$$(\times) = \lambda n m. \text{if}(n = 0)\text{then } 0 \text{ else } (m + (n - 1) \times m)$$

# Fixed-point

In math,  $x$  is a fixed-point of a function  $f$  if

$$x = f(x)$$

for example, 2 is a fixed point of  $f(x) = x^2 - 3x + 4$

Similarly, the fixed-point of  $F$  is the lambda term  $X$  such that

$$X = F X$$

# Memoization

- [y-combinator](#)
- Consider the code for calculating fibonacci numbers

```
let rec fib = n => {  
  switch n {  
    | 0 | 1 => 1  
    | _ => fib(n-1) + fib(n-2)  
  }  
}
```

- `rec` Tying the knot and semantics: fixed recursion



# Memoization

```
let memofib = {  
  let cache = Hashtbl.create(100)  
  (n) => {  
    switch Hashtbl.find_opt(cache, n) {  
    | Some(x) => x  
    | None => {  
      let x = fib(n)  
      Hashtbl.replace(cache, n, x)  
      x  
    }  
    }  
  }  
}
```

# Untying the knot

```
let myfib = (myfib,n)=>{  
  switch n {  
    | 0 | 1 => 1  
    | _ => myfib(n-1)+myfib(n-2)  
  }  
}
```

- not recursive
- open recursion

# Memoization

```
let memo = anyFunc => {  
  let cache = Hashtbl.create(100)  
  let rec fix = (n) => {  
    switch Hashtbl.find_opt(cache, n) {  
    | Some(x) => x  
    | None => {  
      let x = anyFunc(fix,n)  
      Hashtbl.replace(cache, n, x)  
      x  
    }  
    }  
  }  
  fix  
}  
let memofib = memo(myfib)
```

## Homework

- Implement the substitution function  $N[v/x]$  :  
`subst (N: lambda x: string, v: value) : lambda`
- Think about how substitution works on arbitrary terms, i.e.  $N[M/x]$  where  $M$  could contain free variables.
- Implement Church numerals and arithmetic functions using lambda calculus