Lambda Calculus

WANG Hanfei

School of Computer
Wuhan University

March 20, 2022

Contents

Functions

named function

- $f : X \to Y, x \mapsto f(x)$, ex. $s : \mathbb{N} \to \mathbb{N}, n \mapsto n \cup \{n\}$.
- $f : X \to Y, f(x) =$ expression of x, ex. $s : \mathbb{N} \to \mathbb{N}, f(n) = n \cup \{n\}$.
- infix notation: $\Delta : X \times X \to X, \langle x, y \rangle \mapsto x \Delta y$, ex. the compostion: $(A \to A) \times (A \to A) \to (A \to A), \langle R, S \rangle \mapsto R \circ S$.
- C: `int add (int x, int y) { return x + y; }`.

anonymous function

- Haskell: `\x -> x + 2`, `(\f -> f 3) (\x-> x + 2)`
- C#: `x => x + 2`, `(f => f(3))(x => x + 2)` (?)
- Coq: `fun x => x + 2` , `(fun f => f 3)(fun x => x + 2)`
- OCaml: `fun x -> x + 2`, `(fun f -> f 3)(fun x -> x + 2)`

Example in OCaml (Review)

```
# let rec len l = match l with
    [] -> 0
  | a::l1 ->  1 + (len l1);;
val len : 'a list -> int = <fun>
# len [1; 2; 3];;
- : int = 3
# let rec sum l = match l with
    [] -> 0
  | a::l1 ->  a + (sum l1);;
val sum : int list -> int = <fun>
# sum [1; 2; 3];;
- : int = 6
# let rec rev l = match l with
    [] -> []
  | a::l1 -> (rev l1) @ [a];;
val rev : 'a list -> 'a list = <fun>
# rev [1; 2; 3];;
- : int list = [3; 2; 1]
```

Evaluation Processus of len

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
= 1 + (1 + (1 + (len [])))
= 1 + (1 + (1 + (  0   )))
```

Abstraction `1 + (len l)` with function `f(a, len l)`, we have

```
  len [1; 2; 3]
= f(1, len [2; 3])
= f(1, f(2, len [3]))
= f(1, f(2, f(3, len [])))
= f(1, f(2, f(3,   0   )))
```

Evaluation Processus of sum

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> a + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

Abstraction `a + (sum l)` with function `f(a, sum l)`, we have

```
  sum [1; 2; 3]
= f(1, sum [2; 3])
= f(1, f(2, sum [3]))
= f(1, f(2, f(3, sum [])))
= f(1, f(2, f(3,   0   )))
```

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> 1 + rev l1;;
```

```
  rev [1; 2; 3]
= (rev [2; 3]) @ [1]
= ((rev [3]) @ [2]) @ [1]
= (((rev []) @ [3])@ [2]) @ [1]
= (((  []   ) @ [3])@ [2]) @ [1]
```

Abstraction `(rev l) @ a` with function `f(a, rev l)`, we have

```
  rev [1; 2; 3]
= f(1, rev [2; 3])
= f(1, f(2, rev [3]))
= f(1, f(2, f(3, rev [])))
= f(1, f(2, f(3,   []   )))
```

Function as data

- The above 3 functions have the same behaviors: applying consecutively the every list element from right to left to a function f:
$$f(a_1, f(a_2, f(a_3, f(\cdots f(a_n, b)\cdots)))).$$
where $(a_1, a_2, \ldots a_n)$ is the list, and $f : X \times Y \to Y$ is the abstract function which operates on a list element & the result of the application f to the rest of the list. The intial element b will correspond the result of the empty list.
- for `len`, f can be taked $(x, y) \mapsto 1 + y$, $b = 0$.
- for `sum`, f can be taked $(x, y) \mapsto x + y$, $b = 0$.
- for `rev`, f can be taked $(x, y) \mapsto y@[x]$, $b = []$.

Evaluation Processus of sum

define new function fold_right, take sum as an argument

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> a + sum l1;;
```

```
let rec fold_right f l b = match l with
| [] -> b
| a::l1 ->f a (fold_right f l1 b);;
```

```
  fold_right f [1; 2; 3] b
= f(1, fold_right f [2; 3] b)
= f(1, f(2, fold_right f [3] b))
= f(1, f(2, f(3, fold_right f [] b)))
= f(1, f(2, f(3,         b        )))
```

b is the initial element.

Example in OCaml

```
# let rec fold_right f l b = match l with
    [] -> b
  | a::l1 -> f a (fold_right f l1 b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# let len l = fold_right (fun x y -> 1 + y)  l 0;;
val len : 'a list -> int = <fun>
# len [1; 2; 3];;
- : int = 3
# let sum l = fold_right (+) l 0;;
val sum : int list -> int = <fun>
# sum [1; 2; 3];;
- : int = 6
# let rev l = fold_right (fun a l1 -> l1 @ [a]) l [];;
val rev : 'a list -> 'a list = <fun>
# reve [1; 2; 3];;
- : int list = [3; 2; 1]
```

Change to tail recursion

- fold_right is not tail recursive, so the execution is not efficient.
- Because compiler can transform the tail recursion to while-loop, the more efficient way is define the function as tail recursion.
- The tips is change the recursion result to recursion argument, so called "accumulator":
```
let rec sum l = match l with
  [] -> 0
| a::l1 -> a + (sum l1);;
```
could transform to:
```
let rec sum a l = match l with
  [] -> a
| b::l1 -> sum (a + b) l1;;
```
- The same way, define fold_left as
$$f(f(\cdots f(f(f(a, b_1), b_2), b_3), \ldots, b_{n-1}), b_n).$$
where $(b_1, b_2, \ldots b_n)$ is the list, and $f : X \times Y \to X$ is the abstract function which operates on an intial element a and list element, produces the element of same type of the initial element.

Evaluation Processus of sum

define new function fold_left, take sum as an argument f

```
  sum  0 [1; 2; 3]
= sum (0 + 1) [2; 3]
= sum (0 + 1 + 2) [3]
= sum (0 + 1 + 2 + 3) []
=      0 + 1 + 2 + 3
```

```
let rec sum a l = match l with
| [] -> a
| b::l1 -> sum (a + b) l1;;
```

```
let rec fold_left f a l = match l with
| [] -> a
| b::l1 -> fold_left (f a b) l1;;
```

```
  fold_left f a [1; 2; 3]
= fold_left (f(a, 1)) [2; 3]
= fold_left (f(f(a, 1), 2)) [3]
= fold_left (f(f(f(a, 1), 2), 3)) []
=           (f(f(f(a, 1), 2), 3))
```

a is the initial element.

Example in OCaml

```
# let rec fold_left f a l = match l with
    [] -> a
  | b::l1 -> fold_left f (f a b) l1;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# let len l = fold_left (fun x y -> x + 1) 0 l;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
# let sum l = fold_left (+) 0 l;;
val sum : int list -> int = <fun>
# sum [1;2;3];;
- : int = 6
# let rev l = fold_left (fun l1 a -> a::l1) [] l;;
val rev : 'a list -> 'a list = <fun>
# reve [1;2;3];;
- : int list = [3; 2; 1]
```

fold_left is an iterator

```
# let rec aux l a = match l with
| [] -> [a]
| b :: l1 -> if a <= b then a::l else b::(aux l1 a);;
val insert_sort : 'a list -> 'a list = <fun>
# let insert_sort l = fold_left aux [] l;;
val insert_sort : 'a list -> 'a list = <fun>
# insert_sort [3; 1; 6; 2; 4; 5];;
- : int list = [1; 2; 3; 4; 5; 6]
# insert_sort [3; 1; 6; 2; 4; 5; 1; 2]
- : int list = [1; 1; 2; 2; 3; 4; 5; 6]
```

Functional Programming

- the different paradigms from our general imperative programming.
- based on Church computation model: λ-calculus. A program in FP is just λ expression.
- First-class and higher-order functions: functions that can either take other functions as arguments or return them as results.
- No control structures, recursion instead.
- type inference, parametric polymorphism.
- no memory or I/O side effects in pure FP.

λ-calculus

- anonymous function, and function as first citizen, must have a supported formal system which can express the function as data.
- λ-calculus, is just the needed formal system for express function definition, function application and recursion.
- λ-calculus was introduced by Alonzo Church in the 1930s as part of an investigation into the foundations of mathematics.
- it provides a simple mechanism of substitution which is our ordinary meaning of computation.
- it's the base of combinatory logic, type theory, domain theory (for the denotational semantics). so it plays an important role in the development of the theory of programming languages
- it's the computation model of FP (ISWIM, Lisp, Mercury, Miranda, SML, OCaml, Haskell, Erlang...)
- FP is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state.

Terms

Definition

The terms of the λ-calculus, known as, λ-terms, are constructed recursively from a given set of variables $x, y, z, \ldots$. They are inductively defined as: following forms:

- all variables are terms (called atoms);
- if M and N are any terms, then (MN) is a term (called an application);
- if M is any term and x is any variable, then $(\lambda x.M)$ is a term (called an abstraction).

the set of all terms is denoted by Λ.

Example

- $(\lambda v0.(v0v00))$;
- $(\lambda x.(xy))$, $((\lambda y.y)(\lambda x.(xy)))$ (N can be any term);
- $(x(\lambda x.(\lambda x.x)))$ (two occurrences of λx in one term);
- $(\lambda x.(yz))$ (x does not occur in M, vacuous abstraction).

Remarks

- the terms are the 2 binary operator expression system.
- Abstraction λ introduces the argument of function, like the prototype of function definition in PL C, ex: `int add (int x, int y)` will express as $\lambda x.(\lambda y.M)$.
- the body M of abstraction $\lambda x.M$ is like the body of function definition in C, but without any program controls.
- Abstraction is like the quantifiers (universal $\forall$ or existential $\exists$) in first order logic which introduces the well-formed formula (the anonymous boolean function).
- the difference is the application, for the term, it can apply any term. but for logic, it can't apply the predicate itself (with this unlimitation, the function becomes the first citizen). ex: $(\forall x P(f(x)))(3)$ is correct, but $(\forall x P(f(x)))(\forall x P(f(x)))$ is wrong. $((\lambda x.(xx))(\lambda x.(xx)))$.
- please refer our lambda reducer source in Coq : `Lambda.v`.

Conventions

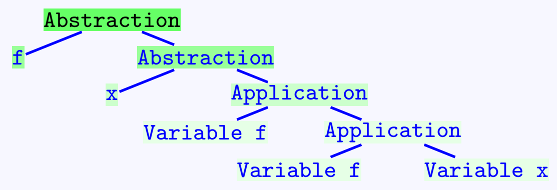
- Application has precedence level higher than the abstraction, ex $(\lambda x.(MN))$ can be simply written $\lambda x.MN$.
- Appliaction is left associative. $N_1 N_2 \cdots N_n$ means $(\cdots(N_1 N_2)\cdots N_n)$.
- Abstraction is right associative and the consecutive abstraction can be intrduced with a single λ. so $\lambda x_1 x_2 \ldots x_n.M$ denotes $(\lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.M)\cdots)))$.
- Syntactic identity of terms will be denoted by '$\equiv$' which means two term are the same alphabetic string (after add the omitted parentheses). so $MP \equiv NQ$ iff $M \equiv P$ and $N \equiv Q$. $(\lambda x.(MN)) \equiv \lambda x.MN$.
- We will use Knuth's Literate programming in our next lectures.

Data type of terms

```
type lambdaExpression =
    Variable of string
  | Abstraction of string * lambdaExpression
  | Apply of lambdaExpression * lambdaExpression;;

#lambda "@fx.f(fx)";;
- : lambdaExpression =
Abstraction ("f", Abstraction ("x", Apply
  (Variable "f", Apply (Variable "f", Variable "x"))))
```

Abstract Syntax Three



Length of the terms

```
let rec lgh = function
| (Variable var) ->  1
| (Abstraction (var, body)) -> 1 + lgh body
| (Apply (func, arg)) ->  lgh func + lgh arg;;

#lgh (lambda "(@x.(@f.(f (f (f (f (f (f (f x)))))))))");;
- : int = 10
```

the length is very useful for induction on the terms.

Free and bound variables

```
let bounds term = let rec bv = function
  | (Variable var) -> []
  | (Abstraction (var, body)) -> union [var] (bv body)
  | (Apply (func, arg)) -> union (bv func) (bv arg)
  in   bv (lambda term);;

let rec fv = function
  | (Variable var) -> [var]
  | (Abstraction (var, body)) -> exclude var (fv body)
  | (Apply (func, arg)) -> union (fv func) (fv arg)
and free term =  fv (lambda term) ;;

#bounds "(@y.yx(@x.y(@y.z)x))vw";;
- : string list = ["x"; "y"]
#free "(@y.yx(@x.y(@y.z)x))vw";;
- : string list = ["v"; "w"; "x"; "z"]
```

Remarks

- the notions of bound and free are the same of the first order formulas, or
- the integral $\int_y^z f(x)dx$ where x is bound, and y, z are free.
- x occurs both bound and free in $(\lambda y.yx(\lambda x.y(\lambda y.z)x))vw$, just like the global and argument with the same name in PL. It's better to avoid this name conflict in practice.
- A closed term is a term without any free variables. ex. $\lambda fx.f(f(fx))$. and we will concentrate only the close terms.

Substitution

the substitution L for every every free occurrence y in the term M, denoted by $M[L/y]$ is inductively defined as

$$x[L/y] \equiv \begin{cases} L & \text{if } x \equiv y \\ x & \text{otherwise} \end{cases}$$

$$(\lambda x.M)[L/y] \equiv \begin{cases} \lambda x.M & \text{if } y \notin \mathrm{FV}(M) \\ \lambda x.(M[L/y]) & x \notin \mathrm{FV}(L) \wedge y \in \mathrm{FV}(M) \\ \lambda z.(M[z/x][L/y]) & x \in \mathrm{FV}(L) \wedge y \in \mathrm{FV}(M) \wedge \\ & z \text{ is new variable not in } \mathrm{FV}(LM) \end{cases}$$

$$(MN)[L/y] \equiv (M[L/y])(N[L/y])$$

Examples

the substitution is similar to the substution rule of first order logic, and the substituted term L in the result $M[L/y]$ must not introduce the new binding. so the free variables in L, should not be the bound variable in M.

- $(\lambda fx.f(fx))[\lambda fx.f(fx)/f] \equiv \lambda fx.f(fx)$ (no free occurrence of f in M).
- $(\lambda fx.f(yx))[\lambda fx.f(fx)/y] \equiv \lambda fx.f((\lambda fx.f(fx))x)$ (L is closed term).
- $(\lambda fx.f(yx))[\lambda f.f(fx)/y] \not\equiv \lambda fx.f((\lambda f.f(fx))x)$ (the free x in L is binding in the result).
- $(\lambda fx.f(yx))[\lambda f.f(fx)/y] \equiv \lambda fv.f((\lambda f.f(fx))v)$.

implementation of substitution

```
let var_counter = ref 0 ;;

let uniqueVar () = var_counter := !var_counter + 1 ;
  "v" ^ (string_of_int !var_counter);;

let rec substitution e x t = match e with
  | (Variable v) ->
      if v = x then t else e
  | (Abstraction (v, b)) ->
      if v = x then e     (* e has no free ocurrences of x *)
      else if not (belongs v (fv t)) then
        (* no free ocurrences of v in t, so no capture *)
        Abstraction (v, substitution b x t)
      else   (* there are free ocurrences of v in t and they
                are all captured -> use alpha equivalence *)
        let z = uniqueVar () in
        let newBody = substitution b v (Variable z) in
          Abstraction (z, substitution newBody x t)
  | (Apply (f,n)) ->
      Apply (substitution f x t, substitution n x t)
  and subst e x t =
    print (substitution (lambda e) x (lambda t));;

#subst "@fx.f(yx)" "y" "@f.f(fx)";;
(@f.(@v1.(f ((@f.(f (f x))) v1))))
```

α-conversions

Definition

Let a term P has an subterm $\lambda x.M$, and let $y \notin FV(M)$.
The act of replacing $\lambda x.M$ by $\lambda y.M[y/x]$ is called a change of bound variable or an α-conversion in P. If P can be changed to Q by a finite (perhaps empty) series of α-conversions, we shall say P α-converts to Q, and denoted by $P \equiv_\alpha Q$.

Example

$$\begin{aligned}\lambda xy.x(xy) &\equiv \lambda x.(\lambda y.x(xy))\\ &\equiv_\alpha \lambda x.(\lambda v.x(xv))\\ &\equiv_\alpha \lambda u.(\lambda v.u(uv))\\ &\equiv \lambda uv.u(uv).\end{aligned}$$

just like changing formal parameter name of subroutine in PL.

Properties of α-conversions

Theorem

- The relation $\equiv_\alpha$ is reflexive, transitive and symmetric (equivalent). That is, for all P, Q, R, we have:
  – reflexivity: $P \equiv_\alpha P$,
  – transitivity: $P \equiv_\alpha Q \wedge Q \equiv_\alpha R \Rightarrow P \equiv_\alpha R$,
  – symmetry: $P \equiv_\alpha Q \Rightarrow Q \equiv_\alpha P$.
- $M \equiv_\alpha M' \wedge N \equiv_\alpha N' \Rightarrow M[N/x] \equiv_\alpha M'[N'/x]$.

- By the symmetry, α-conversion is reversible.
- the α-conversion is congruent relation under the substitution. the α-conversion guarantees that the substitution works correctly. e.g. $\lambda x.y[x/y] = \lambda z.y$, if z is new introduced variable; but $\lambda x.y[x/y] = \lambda w.y$, if w is the new one. and $\lambda z.y \equiv_\alpha \lambda w.y$. if no α-conversion, we get two different term.

β-conversion

Definition

let P a term, any subterm of form

$$(\lambda x.M)N$$

is called a β-redex and the corresponding term

$$M[N/x]$$

is called its contractum. if P' is the result of replacing that occurrence by $M[N/x]$, we say we have contracted the redex-occurrence in P, and P β-converts (reduces) to P' and denoted by

$$P \rhd_{1\beta} P'.$$

the reflexive and transitive closure of $\rhd_{1\beta}$ is denoted by $\rhd_\beta$.

"$\rhd_\beta$" plays the similar role of the TM "$\vdash$", but with the difference.

Examples

- $(\lambda x.x(xy))N \equiv \underline{(\lambda x.x(xy))}\underline{N} \rhd_{1\beta} x(xy)[N/x] \equiv_\alpha N(Ny)$.
- $(\lambda x.x)N \equiv \underline{(\lambda x.x)}\underline{N} \rhd_{1\beta} x[N/x] \equiv_\alpha N$ (identity $\mathbb{1}$).
- $(\lambda x.y)N \equiv \underline{(\lambda x.y)}\underline{N} \rhd_{1\beta} y[N/x] \equiv_\alpha y$ (Constant function).
- $(\lambda x.(\lambda y.yx)z)v \equiv \underline{(\lambda x.(\lambda y.yx)z)}\underline{v} \rhd_{1\beta} (\lambda y.yx)z[v/x] \equiv_\alpha (\lambda y.yv)z \equiv \underline{(\lambda y.yv)}\underline{z} \rhd_{1\beta} yv[z/y] \equiv_\alpha zv$.
- $(\lambda x.(\lambda y.yx)z)v \equiv (\lambda x.\underline{(\lambda y.yx)}\underline{z})v \rhd_{1\beta} (\lambda x.(yx[z/y]))v \equiv_\alpha (\lambda x.(zx))v \equiv \underline{(\lambda x.(zx))}\underline{v} \rhd_{1\beta} zx[v/x] \equiv_\alpha zv$.

Remark

- β-conversion is just like subroutine call in PL which replacing the formal parameter in function $\lambda x.M$ with the actual parameter N.
- Unlike TM "$\vdash$", $\rhd_\beta$ is not functional relationship, for $P \in \Lambda$, $\exists P', P'', P' \not\equiv P'' \wedge P \rhd_\beta P' \wedge P \rhd_\beta P''$.
- We simply denote $\equiv_\alpha$ by $\equiv$.

Nontermination

$$\begin{aligned}(\lambda x.xx)(\lambda x.xx) &\rhd_{1\beta} xx[\lambda x.xx/x] \equiv (\lambda x.xx)(\lambda x.xx)\\ &\rhd_{1\beta} xx[\lambda x.xx/x] \equiv (\lambda x.xx)(\lambda x.xx)\\ &\rhd_{1\beta} xx[\lambda x.xx/x] \equiv (\lambda x.xx)(\lambda x.xx)\\ &\ldots\end{aligned}$$

$$\begin{aligned}(\lambda x.xxy)(\lambda x.xxy) &\rhd_{1\beta} xxy[\lambda x.xxy/x] \equiv (\lambda x.xx)(\lambda x.xx)y\\ &\rhd_{1\beta} (xxy[\lambda x.xxy/x])y \equiv (\lambda x.xxy)(\lambda x.xxy)yy\\ &\rhd_{1\beta} (xxy[\lambda x.xxy/y])yy \equiv (\lambda x.xxy)(\lambda x.xxy)yyy\\ &\ldots\end{aligned}$$

Remark

- like TM "$\vdash$", there exists P and if $P \rhd_\beta P'$, P' always has a redex, and $P' \rhd_{1\beta} P''$. the computation never stops.
- reduction does not always simplify the terms.

β-normal form

Definition

A term Q which contains no β-redexes is called a β-normal form (or simply nf).
If $P \rhd_\beta Q$ and Q is nf, then Q is called nf of P.
If Q is nf, there is not Q' such that $Q \rhd_{1\beta} Q'$.

Examples

- $(\lambda x.(\lambda y.yx)z)v \rhd_\beta zv$ and zv has no redex, so is nf of $(\lambda x.(\lambda y.yx)z)v$.
- $(\lambda x.y)N \rhd_\beta y$ for any $N \in \Lambda$.
- $(\lambda x.xx)(\lambda x.xx)$ has not nf form.

Strategy of reduction

- $\rhd_{1\beta}$ is multivalued relationship.
- if a term has more than one redexes, we must choose one to do the β-conversion.
- so the different strategy of reduction maybe results the different nf. this nondeterminism doesn't conformed with the notion of computability.
- in fact, the nf is unique, if there is. and it's independent of the strategies of reduction.

Examples

- $(\lambda x.(\lambda y.yx)z)v \equiv \underline{(\lambda x.(\lambda y.yx)z)}\underline{v} \rhd_{1\beta} (\lambda y.yx)z[v/x] \equiv_\alpha (\lambda y.yv)z \equiv \underline{(\lambda y.yv)}\underline{z} \rhd_{1\beta} yv[z/y] \equiv_\alpha zv.$
- $(\lambda x.(\lambda y.yx)z)v \equiv (\lambda x.\underline{(\lambda y.yx)}\underline{z})v \rhd_{1\beta} (\lambda x.(yx[z/y]))v \equiv_\alpha (\lambda x.(zx))v \equiv \underline{(\lambda x.(zx))}\underline{v} \rhd_{1\beta} zx[v/x] \equiv_\alpha zv.$

Church Rosser theorem

If $P \rhd_\beta M$ and $P \rhd_\beta N$, then there exists a term T such that
$$M \rhd_\beta T \wedge N \rhd_\beta T.$$

$$\begin{tikzcd}
 & P \arrow[dl] \arrow[dr] & \\
M \arrow[dr, dashed] & & N \arrow[dl, dashed] \\
 & T &
\end{tikzcd}$$

The theorem garantees the uniqueness of nf, and the term can be reduced to two different terms then these two terms can be further reduced to one term, is called confluence.

Different cases of reductions

- disjoint: $\cdots(\lambda x.M)N\cdots(\lambda y.P)Q\cdots$

$$\begin{tikzcd}
 & a((\lambda x.x)b) \arrow[dr] & \\
((\lambda x.x)a)((\lambda x.x)b) \arrow[ur] \arrow[dr] & & ab \\
 & ((\lambda x.x)a)b \arrow[ur] &
\end{tikzcd}$$

  reduction one of the redexes will not effect the another.

- substitution: $\cdots(\lambda x.(\cdots(\lambda y.M)N\cdots))Q\cdots$

$$\begin{tikzcd}
 & (\lambda y.yv)z \arrow[dr] & \\
(\lambda x.(\lambda y.yx)z)v \arrow[ur] \arrow[dr] & & zv \\
 & (\lambda x.(zx)) \arrow[ur] &
\end{tikzcd}$$

- duplication: $\cdots(\lambda x.M)(\cdots(\lambda y.N)P\cdots)\cdots$

$$\begin{tikzcd}
 & (\lambda x.xx)a \arrow[rr] & & aa \\
(\lambda x.xx)(\mathbf{I}\,a) \arrow[ur] \arrow[dr] & & & \\
 & (\mathbf{I}\,a)\mathbf{I}\,a \arrow[r] & a(\mathbf{I}\,a) \arrow[uur] &
\end{tikzcd}$$

  where $\mathbf{I} = \lambda x.x$.

Discussions

- we can transform the substitution case to duplication case by
  $\cdots(\lambda z.(\lambda x.(\cdots z\cdots))((\lambda y.M)N))Q\cdots$
  $\rhd_{1\beta}\cdots(\lambda x.(\cdots(\lambda y.M)N\cdots))Q\cdots$
- this case corresponds the local declarations in OCaml:
```
let x = e in f;;
```
  the compiler will transform it to $(\lambda x.f)e$. e.g.
```
let s = fun x -> x + 1 in s 2;;
```
  it is just (called syntactic sugar)
```
(fun s -> s 2) (fun x -> x +1);;
```
- normally, reduce first the outside redex is more efficient than the inside. but it is not always true.
- the Church-Rosser theorem can be proved by using the strip lemma: if $M \rhd_{1\beta} P$ and $M \rhd_\beta Q$, then there is T such that $P \rhd_\beta T \wedge Q \rhd_\beta T$.

β-equality

- β-reduction is not inversible, so $\rhd_\beta$ is not symmetric relation.
- the symmetric and transitive closure of $\rhd_\beta$ is equivalent relation, called β-equality, denoted by $=_\beta$.
- $P =_\beta Q$ iff Q can be obtained from P by a finite (perhaps empty) series of β-reduction, reversed β-reduction and α-conversion.

Example

$(\lambda xyz.xzy)(\lambda xy.x) =_\beta (\lambda xy.x)(\lambda x.x)$ En fact

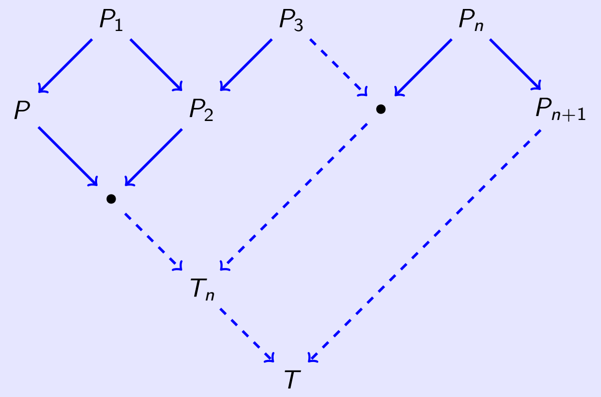
$$\begin{aligned}(\lambda xyz.xzy)(\lambda xy.x) &\rhd_\beta \lambda yz.z\\ (\lambda xy.x)(\lambda x.x) &\rhd_\beta \lambda yx.x\\ &\equiv \lambda yz.z\end{aligned}$$

Church-Rosser theorem for $=_\beta$

If $P =_\beta Q$, then there exists a term T such that $P \rhd_\beta T \wedge Q \rhd_\beta T$.

Illustration of proof by induction

if $P =_\beta Q$ by 0 step $\rhd_{1\beta}$ or the dual, it's $P \equiv Q$.
suppose $P =_\beta P_n$ by n steps of $\rhd_{1\beta}$ or the dual, there is T. then for $n+1$



Undecidability of λ-calculus

- There is no algorithm that takes as input two λ terms and outputs TRUE or FALSE depending on whether or not the two term are β-equal.
- The problem can be reduce to determining whether a given term has a nf. it's the halting problem for λ-calculus. Church assumes that's decidable. then there is term e based on the Gödel numbering, and if e is applied to its own Gödel number, a contradiction results.
- This was historically the first problem for which undecidability could be proved.

Different reduction strategies

let $\Omega = (\lambda x.xx)(\lambda x.xx)$, then

$$\begin{tikzcd}
 & a & & \text{call by name} \\
(\lambda y.a)\Omega \arrow[ru] \arrow[rd] & & & \\
 & (\lambda y.a)\Omega \arrow[r] & (\lambda y.a)\Omega \arrow[r, dotted, no head] & \text{call by value}
\end{tikzcd}$$

- it can be obtained by call-by-name strategy of reduction: function argument (Ω) is not reduced but substituted 'as is' into the body of the abstraction (a). so the substitution erases the argument.
- if reduce the argument (Ω) first (call-by-value), then reductions are trapped into Ω without termination and never reach the nf.
- whether a term has nf or not, and how much work needs to be done in reaching it if there is, depends to a large extent on the reduction strategy used.
- the compiler of PL must choose the reduction strategies for it works as determinstic program.

Applicative order

- The rightmost, innermost redex is always reduced first. Intuitively this means a function's arguments are always reduced before the function itself. Applicative order always attempts to apply functions to normal forms, even when this is not possible.
- most FP (including Lisp, ML) use this strategy, it also called "eager (strict) evaluation"
- because a redex is reduced only when its right hand side (function argument) has reduced to nf. It is also called call-by-value. most imperative languages like C and Java use this convention for function call. e.g.
  `(x -> x + x) (3 * 4) => (x -> x + x) 7 => 7 + 7`
- it's efficient, but it's not the normalising strategy (which always obtains the nf if there is).
- it can be implemented by post-order tree traversal (from right to left).

Examples

$(\lambda x.a((\lambda y.by)x))((\lambda z.(\lambda u.ud)z)c)$

$$\begin{aligned} & (\lambda x.a((\lambda y.by)x))((\lambda z.(\lambda u.ud)z)c) \\ \rhd_{1\beta} & (\lambda x.a((\lambda y.by)x))((\lambda z.zd)c) \\ \rhd_{1\beta} & (\lambda x.a((\lambda y.by)x))(cd) \\ \rhd_{1\beta} & (\lambda x.a(bx))(cd) \\ \rhd_{1\beta} & a(b(cd)) \end{aligned}$$

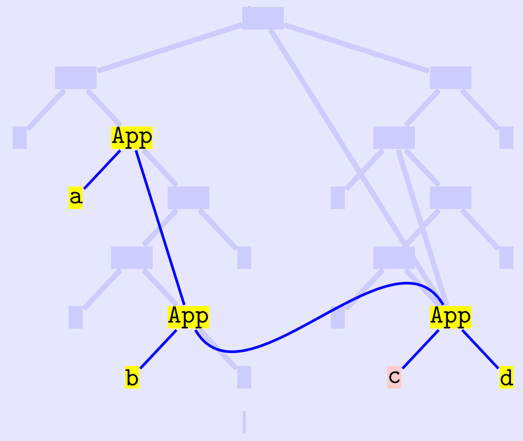
OCaml Example

OCaml use eager evaluation as default reduction strategy:

```
# let f = (fun x -> let y = print_string "a"; x + 2
    in print_string "b"; y + 3);;
# f (let y = print_string "c"; 3
    in print_string "d"; y + 3);;
cdab- :  int = 11
```

Example: applicative order animation

$(\lambda x.a((\lambda y.by)x))((\lambda z.(\lambda u.ud)z)c)$



implementation of reduction of applicative order

```
let rec reductionStepInnerRightOrder = function
  | (Variable var) -> raise Lfail
  | (Abstraction (var, body)) ->
    Abstraction (var, reductionStepInnerRightOrder body)
  | (Apply (func, arg)) ->
     try Apply (func, reductionStepInnerRightOrder arg)
     with Lfail ->
       try Apply (reductionStepInnerRightOrder func, arg)
       with Lfail ->
         match func with
          | Abstraction (var, body) ->
              (* beta reduction *)
              substitution body var arg
          | _ -> raise Lfail
;;
```

Normal order

- the leftmost outermost redex is always reduced first, applying functions before evaluating function arguments.
- it correspond preorder traversal of abstract syntax tree.
- because the reduction of the right hand side of the redex (function argument) is delayed. It is also called call-by-name. ALGOL 60 uses this convention. e.g.
  `(x -> x + x) (3 * 4) => (3 * 4) + (3 * 4) => 7 + (3 * 4) => 7 + 7`
- it isn't efficient, but it's the normalising strategy (which always obtains the nf if there is).

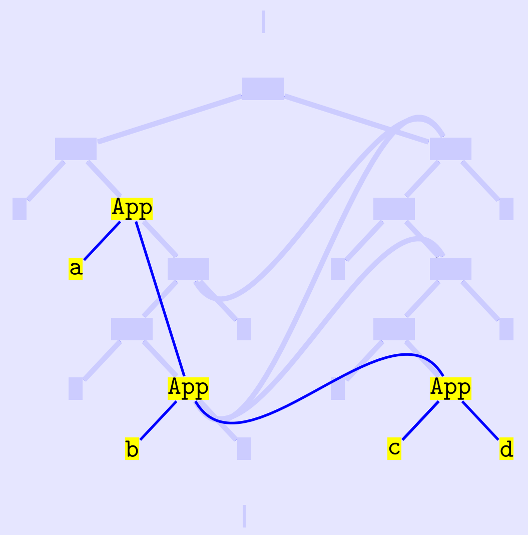
Examples

$(\lambda x.a((\lambda y.by)x))((\lambda z.(\lambda u.ud)z)c)$

$$
\begin{aligned}
&(\lambda x.a((\lambda y.by)x))((\lambda z.(\lambda u.ud)z)c)\\
\rhd_{1\beta}\,&a((\lambda y.by)((\lambda z.(\lambda u.ud)z)c))\\
\rhd_{1\beta}\,&a((b((\lambda z.(\lambda u.ud)z)c)))\\
\rhd_{1\beta}\,&a(b((\lambda u.ud)c))\\
\rhd_{1\beta}\,&a(b(cd))
\end{aligned}
$$

Example: normal order animation

$(\lambda x.a((\lambda y.by)x))((\lambda z.(\lambda u.ud)z)c)$



implementation of reduction of applicative order

```
let rec reductionStepOutLeftOrder = function
  | (Variable var) -> raise Lfail
  | (Abstraction (var, body)) ->
     Abstraction (var, reductionStepOutLeftOrder body)
  | (Apply (func, arg)) ->
     match func with
       | Abstraction (var, body) -> (* beta reduction *)
          substitution body var arg
       | _ ->
          try Apply (reductionStepOutLeftOrder func, arg)
          with Lfail ->
            Apply (func, reductionStepOutLeftOrder arg)
;;
```

Lazy evaluation

- if there are multiple occurrences of bound variable in function body, the normal reduction must evaluate the same function argument multiple times if the argument has redexes. So it's inefficient.
  $(\lambda x.xx)((\lambda x.x)y) \rhd_{1\beta} ((\lambda x.x)y)((\lambda x.x)y) \rhd_{1\beta} y((\lambda x.x)y) \rhd_{1\beta} yy$
- Lazy evaluation (or call-by-need) is an improved normal reduction. which never evaluates an argument more than once. it evaluate the argument until its value is actually required and the next occurrence of the argument will share the result of the first one. So it's optimal. e.g.
  $(\lambda x.xx)((\lambda x.x)y) \rhd_{1\beta} ((\lambda x.x)y)((\lambda x.x)y) \rhd_{\beta} yy$
- it also called non-strict evaluation.
- it can be implemented by representing the term by a graph rather than a tree.

Lazy evaluation (cont'd)

- most purely functional programming languages (Miranda, Haskell) use lazy evaluation as default reduction strategy.
- OCaml use `lazy` and `Lazy.force` to change the eager evaluation to the lazy. e.g.

```
# let x = lazy (print_string "Hello"; 3*4);;
val x :  int lazy_t = <lazy>
# Lazy.force x;;
Hello- :  int = 12
# Lazy.force x;;
- :  int = 12
```

- .NET can simulate lazy evaluation using the type `Lazy<T>`.
- C's boolean expression is compiled to lazy by using short circuit technics.
- because the order of operations becomes indeterminate, it is difficult to combine with imperative features such as exception handling and input/output in lazy languages.

Encoding data in the λ-calculus

- In the imperative programming languages, data and controls are different objects. e.g. "Algorithms + Data Structures = Program" by N. Wirth.
- In the λ-calculus, data and controls are unified to the same objects — terms.
- The λ-calculus is expressive enough to encode boolean values, ordered pairs, natural numbers and lists as terms
- So the encoded data can carry their control with them. this mecanism let us realize high level abstract function. e.g. fold_left in OCaml.

Booleans

- **if** can be seen as 3 argument function. **if true** $M\ N$ will return M and **if false** $M\ N$ return N. so **true** and **false** will be 2 argument functions.
- encoding **if**, **true** and **false** as:
  $$\begin{aligned}\textbf{true} &\equiv \lambda xy.x\\ \textbf{false} &\equiv \lambda xy.y\\ \textbf{if} &\equiv \lambda pxy.pxy\end{aligned}$$
  so **if true** $M\ N =_\beta M$ and **if false** $M\ N =_\beta N$
- conjunction, disjunction and negation can be expressed as:
  $$\begin{aligned}\textbf{and} &\equiv \lambda pq.\textbf{if}\ p\ q\ \textbf{false}\\ \textbf{or} &\equiv \lambda pq.\textbf{if}\ p\ \textbf{true}\ q\\ \textbf{not} &\equiv \lambda p.\textbf{if}\ p\ \textbf{false}\ \textbf{true}\end{aligned}$$

Ordered Pairs

- the pair is the control which contains two element in order. **fst** and **snd** will return the first and second element
- encoding **pair**, **fst** and **snd** as:

$$\begin{aligned}\textbf{pair} &\equiv \lambda xyf.fxy\\ \textbf{fst} &\equiv \lambda p.p\,\textbf{true}\\ \textbf{snd} &\equiv \lambda p.p\,\textbf{false}\end{aligned}$$

  so for any terms M, N, $\textbf{pair}\,M\,N =_\beta \lambda f.f\,M\,N$, packaging M and N consecutively. f will be the place of control for output the first and second element.
- if a pair apply **fst**, it will binding f to **true** and out the first element:

$$\begin{aligned}&\textbf{fst}\,(\textbf{pair}\,M\,N)\\ \rhd_\beta\,&\textbf{fst}\,(\lambda f.f\,M\,N)\\ \rhd_\beta\,&(\lambda f.f\,M\,N)\,\textbf{true}\\ \rhd_\beta\,&\textbf{true}\,M\,N \rhd_\beta M\end{aligned}$$

  and $\textbf{snd}\,(\textbf{pair}\,M\,N) =_\beta N$.

Natural numbers

- Church numerals are the representations of natural numbers under Church encoding. the "value" $\underline{n}$ is equivalent to the number of times the function encapsulates its argument:

$$f^n = f \circ f \circ \cdots \circ f$$

- so the Church numerals are defines as

$$\begin{aligned}\underline{0} &\equiv \lambda fx.x\\ \underline{1} &\equiv \lambda fx.fx\\ \underline{2} &\equiv \lambda fx.f(fx)\\ &\vdots\\ \underline{n} &\equiv \lambda fx.\underbrace{f(\cdots(f}_{n\text{ times}}\,x)\cdots)\end{aligned}$$

- so for any term F and X, we have:

$$\underline{n}F\,X =_\beta F^nX$$

  where $F^nX \equiv F(F(\cdots(F\,X)\cdots))$.

Arithmetic on Church numberals

- for function compositions, we have

$$\begin{aligned}f^m \circ f^n &= f^{m+n}\\ (f^m)^n &= f^{mn}\end{aligned}$$

  and the monoid $\langle f\rangle$ is isomorphic to $\mathbb{N}$.
- so the addition, multiplication and expoentiation are defines as

$$\begin{aligned}\textbf{add} &\equiv \lambda mnfx.mf(nfx)\\ \textbf{mult} &\equiv \lambda mnfx.m(nf)x\\ \textbf{expt} &\equiv \lambda mnfx.nmfx\end{aligned}$$

- so for any $\underline{m}$ and $\underline{n}$, we have:

$$\begin{aligned}\textbf{add}\,\underline{m}\,\underline{n} &\rhd_\beta (\lambda mnfx.mf(nfx))\underline{m}\,\underline{n}\\ &\rhd_\beta \lambda fx.\underline{m}f(\underline{n}fx)\\ &\rhd_\beta \lambda fx.\underline{m}f(f^nx)\\ &\rhd_\beta \lambda fx.f^m(f^nx)\\ &\rhd_\beta \lambda fx.f^{m+n}x\end{aligned}$$

Arithmetic on Church numberals (cont'd)

- and

$$\begin{aligned}\textbf{mult}\,\underline{m}\,\underline{n} &\rhd_\beta (\lambda mnfx.m(nf)x)\underline{m}\,\underline{n}\\ &\rhd_\beta \lambda fx.\underline{m}(\underline{n}f)x)\\ &\rhd_\beta \lambda fx.(\underline{n}f)^mx\\ &\rhd_\beta \lambda fx.(f^n)^mx)\\ &\rhd_\beta \lambda fx.f^{m\times n}x\end{aligned}$$

- and

$$\begin{aligned}\textbf{expt}\,\underline{m}\,\underline{n} &\rhd_\beta (\lambda mnfx.nmfx)\underline{m}\,\underline{n}\\ &\rhd_\beta \lambda fx.\underline{n}\,\underline{m}fx\\ &\rhd_\beta \lambda fx.\underline{m}^nfx\\ &\rhd_\beta \lambda fx.f^{m^n}x\end{aligned}$$

- because the Church numerals have an inbuilt source of repetition, we can encode arithmetic operation without the recursion.

Basic operations on Church numberals

- the successor and zero test can be encoded as

$$\begin{aligned}\textbf{succ} &\equiv \lambda nfx.f(nfx)\\ \textbf{iszero} &\equiv \lambda n.n(\lambda x.\textbf{false})\textbf{true}\end{aligned}$$

- so for any $\underline{n}$, we have:

$$\begin{aligned}\textbf{succ}\,\underline{n} &\rhd_\beta \underline{n+1}\\ \textbf{iszero}\,\underline{0} &\rhd_\beta (\lambda n.n(\lambda x.\textbf{false})\textbf{true})\underline{0}\\ &\rhd_\beta \underline{0}(\lambda x.\textbf{false})\textbf{true}\\ &\rhd_\beta (\lambda fx.x)(\lambda x.\textbf{false})\textbf{true}\\ &\rhd_\beta \textbf{true}\\ \textbf{iszero}\,\underline{n+1} &\rhd_\beta (\lambda n.n(\lambda x.\textbf{false})\textbf{true})\underline{n+1}\\ &\rhd_\beta \underline{n+1}(\lambda x.\textbf{false})\textbf{true}\\ &\rhd_\beta (\lambda x.\textbf{false})^{n+1}\,\textbf{true}\\ &\equiv\ (\lambda x.\textbf{false})^{n}((\lambda x.\textbf{false})\,\textbf{true})\\ &\rhd_\beta \textbf{false}\end{aligned}$$

Basic operations on Church numberals (cont'd)

- because the Church numeral is an iterator, we must use the $n+1$ iterator to generate the one of n. if choosing $\textbf{predfn}(f)\langle x,x\rangle = \langle f(x),x\rangle$ as first argument and $\langle x,x\rangle$ as second argument of $\underline{n+1}$. then

$$\begin{aligned}&\underline{n+1}(\textbf{predfn}\,f)\langle x,x\rangle\\ =&(\textbf{predfn}\,f)^n((\textbf{predfn}\,f)\langle x,x\rangle)\\ =&(\textbf{predfn}\,f)^n\langle f(x),x\rangle\\ =&(\textbf{predfn}\,f)^{n-1}((\textbf{predfn}\,f)\langle f(x),x\rangle)\\ =&(\textbf{predfn}\,f)^{n-1}\langle f^2(x),f(x)\rangle\\ &\ \cdots\\ =&\langle f^{n+1}(x),f^n(x)\rangle\end{aligned}$$

- so

$$\begin{aligned}\textbf{predfn} &\equiv \lambda fp.\textbf{pair}(f(\textbf{fst}\,p))(\textbf{fst}\,p)\\ \textbf{pred} &\equiv \lambda nfx.\textbf{snd}(n(\textbf{predfn}\,f)(\textbf{pair}\,x\,x))\\ \textbf{sub} &\equiv \lambda mn.n\,\textbf{pred}\,m\end{aligned}$$

List

- in maths, a list $[x_1,x_2,\cdots,x_n]$ can be expressed as an n tuple $\langle x_1,x_2,\cdots,x_n\rangle \triangleq \langle x_1,\langle x_2,\langle\cdots,\langle x_n,[]\rangle\cdots\rangle\rangle\rangle$.
- so the list can be encoded as nested pairs :

$$\begin{aligned}\textbf{cons} &\equiv \textbf{pair} \equiv \lambda xyf.fxy\\ \textbf{hd} &\equiv \textbf{fst} \equiv \lambda p.p\,\textbf{true}\\ \textbf{tl} &\equiv \textbf{snd} \equiv \lambda p.p\,\textbf{false}\\ \textbf{nil} &\equiv \lambda x.\textbf{true}\\ \textbf{null} &\equiv \lambda l.l\lambda xy.\textbf{false}\end{aligned}$$

- then for any term M and N, we have

$$\begin{aligned}\textbf{null}(\textbf{cons}\,M\,N)\ &\rhd_\beta (\textbf{cons}\,M\,N)\,\lambda xy.\textbf{false}\\ &\rhd_\beta (\lambda f.f\,M\,N)\,\lambda xy.\textbf{false}\\ &\rhd_\beta (\lambda xy.\textbf{false})\,M\,N \rhd_\beta \textbf{false}\end{aligned}$$

  the reduction does not use any list element the testing if the list is empty. so the **cons** and **pair** are lazy constructors. with this, we can infinite lists.

Ackermann's function

- Most computable can be encoded by Church numerals with the power of inbuit repetition. e.g. Ackermann's function is not primitive recursive, but it can be encoded by Church numerals as:

$$\textbf{ack} \equiv \lambda m.m(\lambda fn.nf(f\underline{1}))\textbf{succ}$$

- we can see:

$$\begin{aligned}\textbf{ack}\,\underline{0}\,\underline{n} &\rhd_\beta \underline{0}(\lambda fn.nf(f\underline{1}))\textbf{succ}\,\underline{n}\\ &\rhd_\beta \textbf{succ}\,\underline{n} \rhd_\beta \underline{n+1}\end{aligned}$$

- and

$$\begin{aligned}\textbf{ack}\,\underline{m+1}\,\underline{n} &\rhd_\beta \underline{m+1}(\lambda fn.nf(f\underline{1}))\textbf{succ}\,\underline{n}\\ &\rhd_\beta (\lambda fn.nf(f\underline{1}))(m(\lambda fn.nf(f\underline{1}))\,\textbf{succ})\underline{n}\\ &=_\beta (\lambda fn.nf(f\underline{1}))(\textbf{ack}\,\underline{m})\underline{n}\\ &\rhd_\beta \underline{n}(\textbf{ack}\,\underline{m})(\textbf{ack}\,\underline{m}\,\underline{1})\end{aligned}$$

- so

$$\begin{aligned}\textbf{ack}\,\underline{m+1}\,\underline{0}\ &\rhd_\beta \underline{0}(\textbf{ack}\,\underline{m})(\textbf{ack}\,\underline{m}\,\underline{1})\\ &\rhd_\beta \textbf{ack}\,\underline{m}\,\underline{1}\end{aligned}$$

Ackermann's function (cont'd)

- $\textbf{ack}\,\underline{m+1}\,\underline{n} \rhd_\beta \underline{n}(\textbf{ack}\,\underline{m})(\textbf{ack}\,\underline{m}\,\underline{1})$ as the lemma.
- and:
$$\begin{aligned}\textbf{ack}\,\underline{m+1}\,\underline{n+1} &\rhd_\beta \underline{n+1}(\textbf{ack}\,\underline{m})(\textbf{ack}\,\underline{m}\,\underline{1})\\ &\rhd_\beta (\textbf{ack}\,\underline{m})(\underline{n}(\textbf{ack}\,\underline{m})(\textbf{ack}\,\underline{m}\,\underline{1}))\\ &=_\beta (\textbf{ack}\,\underline{m})(\textbf{ack}\,\underline{m+1}\,\underline{n})\end{aligned}$$
- then we have:
$$\begin{aligned}\textbf{ack}\,\underline{0}\,\underline{n} &=_\beta \underline{n+1}\\ \textbf{ack}\,\underline{m+1}\,\underline{0} &=_\beta \textbf{ack}\,\underline{m}\,\underline{1}\\ \textbf{ack}\,\underline{m+1}\,\underline{n+1} &=_\beta (\textbf{ack}\,\underline{m})(\textbf{ack}\,\underline{m+1}\,\underline{n})\end{aligned}$$
which perfectly match the recursive definition of Ackermann's function.

Ackermann's function

- Most computable can be encoded by Church numerals with the power of inbuit repetition. e.g. Ackermann's function is not primitive recursive, but it can be encoded by Church numerals as:
$$\textbf{ack} \equiv \lambda m.m(\lambda fn.nf(f\underline{1}))\textbf{succ}$$
- we can see:
$$\begin{aligned}\textbf{ack}\,\underline{0}\,\underline{n} &\rhd_\beta \underline{0}(\lambda fn.nf(f\underline{1}))\textbf{succ}\,\underline{n}\\ &\rhd_\beta \textbf{succ}\,\underline{n} \rhd_\beta \underline{n+1}\end{aligned}$$
- and
$$\begin{aligned}\textbf{ack}\,\underline{m+1}\,\underline{n} &\rhd_\beta \underline{m+1}(\lambda fn.nf(f\underline{1}))\textbf{succ}\,\underline{n}\\ &\rhd_\beta (\lambda fn.nf(f\underline{1}))(\underline{m}(\lambda fn.nf(f\underline{1}))\,\textbf{succ})\underline{n}\\ &=_\beta (\lambda fn.nf(f\underline{1}))(\textbf{ack}\,\underline{m})\underline{n}\\ &\rhd_\beta \underline{n}(\textbf{ack}\,\underline{m})(\textbf{ack}\,\underline{m}\,\underline{1})\end{aligned}$$
- so
$$\begin{aligned}\textbf{ack}\,\underline{m+1}\,\underline{0} &\rhd_\beta \underline{0}(\textbf{ack}\,\underline{m})(\textbf{ack}\,\underline{m}\,\underline{1})\\ &\rhd_\beta \textbf{ack}\,\underline{m}\,\underline{1}\end{aligned}$$

Recursion and fixed-points

- although it's possible encoding nearly all computable functions directly using Church numerals, but it's barely feasable with the complexity of recursions under composition. we must find the general method to express the recursions.
- recursion is the definition of a function using the function itself. e.g. the mathematical definition of factorial is
$$F\,N = \textbf{if}\,(\textbf{iszero}\,N)\,\underline{1}\,(\textbf{mult}\,N\,(F(\textbf{pred}\,N)))$$
the right hand side can be seen as a functional $(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$:
$$G \equiv \lambda gn.\textbf{if}\,(\textbf{iszero}\,n)\,\underline{1}\,(\textbf{mult}\,n\,(g(\textbf{pred}\,n)))$$
so the factorial F is a fixed-point of the functional G: $G(F) = F$. In fact, all recursive definition can be seen as the fixed-point of a functional.

Recursion and fixed-points (cont'd)

- if g is the solution of the above term, then $g\,n = F(n) = G\,g\,n$. so $G\,g = g$. and g is fixed-point if G.
- so we must have $G(g) =_\beta g$ to solve the recursion.
- in fact, there is magic term called fixed-point combinator $\mathbf{Y}$ such that $\mathbf{Y}\,F =_\beta F(\mathbf{Y}\,F)$ for all terms F.
- so $\mathbf{Y}\,G =_\beta G(\mathbf{Y}\,G)$ is the fixed-point we expect.
- $\mathbf{Y}$ was discovered by Haskell B. Curry. it is defined as:
$$\mathbf{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$
- then
$$\begin{aligned}\mathbf{Y}\,F &\rhd_\beta (\lambda x.F(xx))(\lambda x.F(xx))\\ &\rhd_\beta F((\lambda x.F(xx))(\lambda x.F(xx)))\\ &\lhd_\beta F(\mathbf{Y}\,F)\end{aligned}$$
- thus $\mathbf{Y}\,F =_\beta F(\mathbf{Y}\,F)$. from the reduction above, we can see that $\mathbf{Y}$ has self-replicating engine which is just the essence of recursion.

Example of reductions with **Y** (normal order)

$$
\begin{aligned}
& \mathbf{Y}\, G\, \underline{2} \\
\rhd_\beta\ & (\lambda x.G(xx))(\lambda x.G(xx))\, \underline{2} \\
\rhd_\beta\ & G((\lambda x.G(xx))\lambda x.G(xx))\, \underline{2} \quad (F \equiv (\lambda x.G(xx))\lambda x.G(xx)) \\
\rhd_\beta\ & (\lambda gn.\mathbf{if}\ (\mathbf{iszero}\ n)\ \underline{1}\ (\mathbf{mult}\ n\ (g(\mathbf{pred}\ n))))F\ \underline{2} \\
\rhd_\beta\ & \mathbf{if}\ (\mathbf{iszero}\ \underline{2})\ \underline{1}\ (\mathbf{mult}\ \underline{2}\ (F(\mathbf{pred}\ \underline{2}))) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ (F\underline{1}) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ (((\lambda x.G(xx))\lambda x.G(xx))\underline{1}) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ G((\lambda x.G(xx))(\lambda x.G(xx))\ \underline{1}) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ (\mathbf{if}\ (\mathbf{iszero}\ \underline{1})\ \underline{1}\ (\mathbf{mult}\ \underline{1}\ (F(\mathbf{pred}\ \underline{1})))) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ (\mathbf{mult}\ \underline{1}\ (F\ \underline{0})) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ (\mathbf{mult}\ \underline{1}\ ((\lambda x.G(xx))\lambda x.G(xx))\ \underline{0})) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ (\mathbf{mult}\ \underline{1}\ (G((\lambda x.G(xx))\lambda x.G(xx))\ \underline{0})) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ (\mathbf{mult}\ \underline{1}\ (\mathbf{if}\ (\mathbf{iszero}\ \underline{0})\ \underline{1}\ (\mathbf{mult}\ \underline{0}\ (F(\mathbf{pred}\ \underline{0}))))) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ (\mathbf{mult}\ \underline{1}\ \underline{1}) \\
\rhd_\beta\ & \mathbf{mult}\ \underline{2}\ \underline{1} \\
\rhd_\beta\ & \underline{2}
\end{aligned}
$$

Remarks

- **Y** will not work in the applicative order:

$$
\begin{aligned}
& \mathbf{Y}\, G\, \underline{0} \\
\rhd_\beta\ & \mathbf{Y}\ (\lambda gn.\mathbf{if}\ (\mathbf{iszero}\ n)\ \underline{1}\ (\mathbf{mult}\ n\ (g(\mathbf{pred}\ n))))\ \underline{0} \\
\rhd_\beta\ & (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\ F\ \underline{0} \\
\rhd_\beta\ & (\lambda f.f((\lambda x.f(xx))(\lambda x.f(xx))))\ F\ \underline{0} \\
\rhd_\beta\ & (\lambda f.f(f((\lambda x.f(xx))(\lambda x.f(xx)))))\ \underline{0} \\
\rhd_\beta\ & \cdots
\end{aligned}
$$

- for applicative order evaluation, we can use the fixed-point combinator **Z** defined by:
$$\mathbf{Z} = \lambda f.(\lambda x.f(\lambda y.(xx)y))(\lambda x.f(\lambda y.(xx)y)))$$
but it works only if the **if then else** must be evaluated in lazy.
- in fact, the set of fixed-point combinators is recursively enumerable
- **Y** is discovered by the encoded Russell's paradox: if let $R \equiv \lambda x.\mathbf{not}(xx)$, then $R\,R =_\beta \mathbf{not}(R\,R)$. which is a contradiction in logic. if replacing **not** by an arbitrary term F, we got **Y**. the typed λ-calculus does not admit this unptyped term.

Examples of encoding recursions

- just place **Y** before the recursive definition to obtain the fixed-point:

$$
\begin{aligned}
\mathbf{fact} &\equiv \mathbf{Y}\ (\lambda gn.\mathbf{if}\ (\mathbf{iszero}\ n)\ \underline{1}\ (\mathbf{mult}\ n\ (g(\mathbf{pred}\ n)))) \\
\mathbf{sum} &\equiv \mathbf{Y}\ (\lambda fn.\mathbf{if}\ (\mathbf{iszero}\ n)\ \underline{0}\ (\mathbf{add}\ n\ (f(\mathbf{pred}\ n)))) \\
\mathbf{append} &\equiv \mathbf{Y}\ (\lambda gzw.\mathbf{if}\ (\mathbf{null}\ z)\ w\ (\mathbf{cons}\ (\mathbf{hd}\ z)\ (g(\mathbf{tl}\ z)\ w))) \\
\mathbf{getn} &\equiv \mathbf{Y}\ (\lambda fnl.\mathbf{if}(\mathbf{null}\ l)\ \mathbf{false}\ (\mathbf{if}(\mathbf{iszero}\ n)(\mathbf{hd}\ l)(f\ (\mathbf{pred}\ n)(\mathbf{tl}\ l)))) \\
\mathbf{fibogen} &\equiv \mathbf{Y}\ (\lambda lab.\mathbf{cons}\ a\ (l\ b\ (\mathbf{add}\ a\ b))) \\
\mathbf{fibo} &\equiv \mathbf{fibogen}\ \underline{0}\ \underline{1}
\end{aligned}
$$

- **fibo** will recursively defined the infinite Fibonacci sequence $[0, 1, 1, 2, 3, 5, 8, \ldots]$, if using the normal order (or lazy), we will get the expected result without any risk to trap in the infinite loops. e.g.
$$\mathbf{getn}\ \underline{5}\ \mathbf{fibo} \rhd_\beta \underline{5}$$

Exercises 6

- reimplement the function `lambdaToString` which return the most simplest of term replace the one with redundant parentheses. (e.g. `(@x.(@y.(xy)))` will simply output `@xy.xy`.
- show that the $\mathbf{sub}\underline{m}\,\underline{n}$ will perform $m - n$.
- show for all terms F, $\mathbf{Z}F =_\beta F(\mathbf{Z}F)$.
- give recursive definitions in term of exercises 5(1). (you can use relation operations: **gt**, **ge**, **lt**, **le** and **eq**)
- give recursive definition of the infinite list $[0, 2, 4, \ldots]$.

Contents