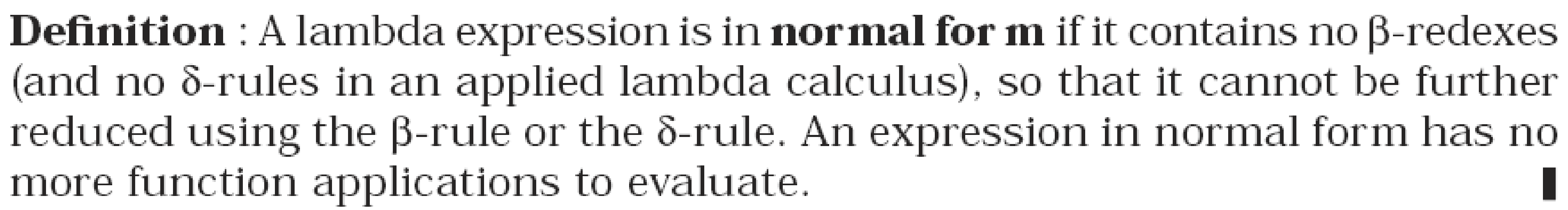
LAMBDA CALCULUS

Part II

SOLVING A LAMBDA EXPRESSION

The main goal of manipulating a lambda expression is to reduce it to a “simplest form” and consider that as the value of the lambda expression.



Definition : A lambda expression is in **normal form** if it contains no β-redexes (and no δ-rules in an applied lambda calculus), so that it cannot be further reduced using the β-rule or the δ-rule. An expression in normal form has no more function applications to evaluate. ∎

(λx. λz.z) ((λy. yy) (λu. uu))

The only reduction possible for an expression in normal form is an @-reduction.

REDUCTION STRATEGIES

Can every lambda expression be reduced to a normal form?

Is there more than one way to reduce a particular lambda expression?

If there is more than one reduction strategy, does each one lead to the same normal form expression?

Is there a reduction strategy that will guarantee that a normal form expression will be produced?

CAN EVERY LAMBDA EXPRESSION BE REDUCED TO A NORMAL FORM?

The identity function:

$(\lambda x.\ x)\ E \rightarrow [E\ /\ x]\ x = E$

Another example with the identity:

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

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

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

$(\lambda x.\ x)\ (\lambda y.\ y) \rightarrow$

$[\lambda y.\ y\ /x]\ x = \lambda y.\ y$

REDUCTION STRATEGIES

Can every lambda expression be reduced to a normal form?

Is there more than one way to reduce a particular lambda expression?

If there is more than one reduction strategy, does each one lead to the same normal form expression?

Is there a reduction strategy that will guarantee that a normal form expression will be produced?

IF THERE IS MORE THAN ONE REDUCTION STRATEGY, DOES EACH ONE LEAD TO THE SAME NORMAL FORM EXPRESSION?

$(\lambda\ y.\ (\lambda\ x.\ x)\ y)\ E$

- could reduce the inner or the outer λ
- which one should we pick?

$$\begin{tikzcd}
& (\lambda\ y.\ (\lambda\ x.\ x)\ y)\ E \arrow[dl, "\text{inner}"'] \arrow[dr, "\text{outer}"] & \\
(\lambda y.\ [y/x]\ x)\ E = (\lambda y.\ y)\ E \arrow[dr] & & [E/y]\ (\lambda x.\ x)\ y = (\lambda x.\ x)\ E \arrow[dl] \\
& E &
\end{tikzcd}$$

IS THERE MORE THAN ONE WAY TO REDUCE A PARTICULAR LAMBDA EXPRESSION?

Example : (λy . 5) ((λx . x x) (λx . x x))

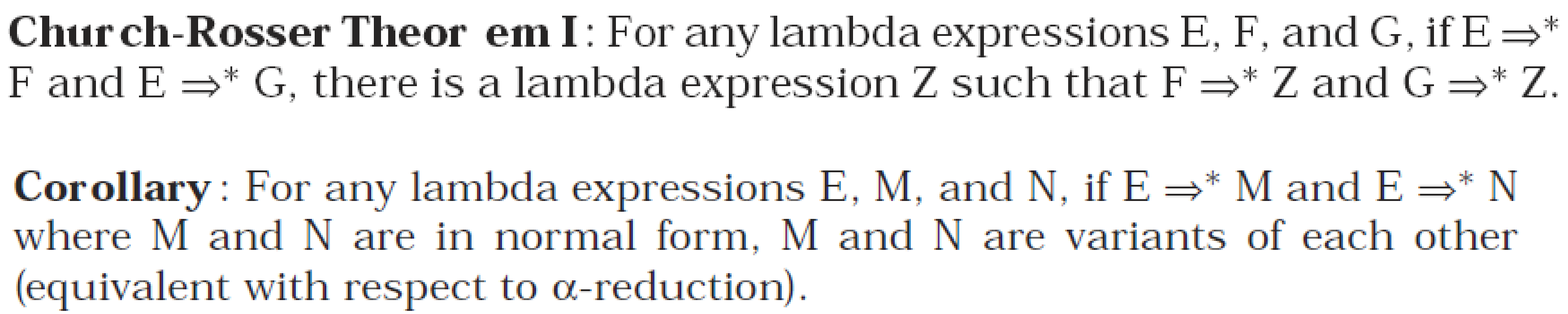
Definition : A **normal order** reduction always reduces the leftmost outermost β-redex (or δ-redex) first. An **applicative order** reduction always reduces the leftmost innermost β-redex (or δ-redex) first. ∎

Definition : For any lambda expression of the form E = ((λx . B) A), we say that β-redex E is **outside** any β-redex that occurs in B or A and that these are **inside** E.
A β-redex in a lambda expression is **outer most** if there is no β-redex outside of it, and it is **inner most** if there is no β-redex inside of it.

ORDER OF EVALUATION (CONT.)

The Church-Rosser theorem says that any order will compute the same result

- A result is a λ-term that cannot be reduced further



Church-Rosser Theorem I: For any lambda expressions E, F, and G, if E $\Rightarrow^*$ F and E $\Rightarrow^*$ G, there is a lambda expression Z such that F $\Rightarrow^*$ Z and G $\Rightarrow^*$ Z.

Corollary: For any lambda expressions E, M, and N, if E $\Rightarrow^*$ M and E $\Rightarrow^*$ N where M and N are in normal form, M and N are variants of each other (equivalent with respect to α-reduction).

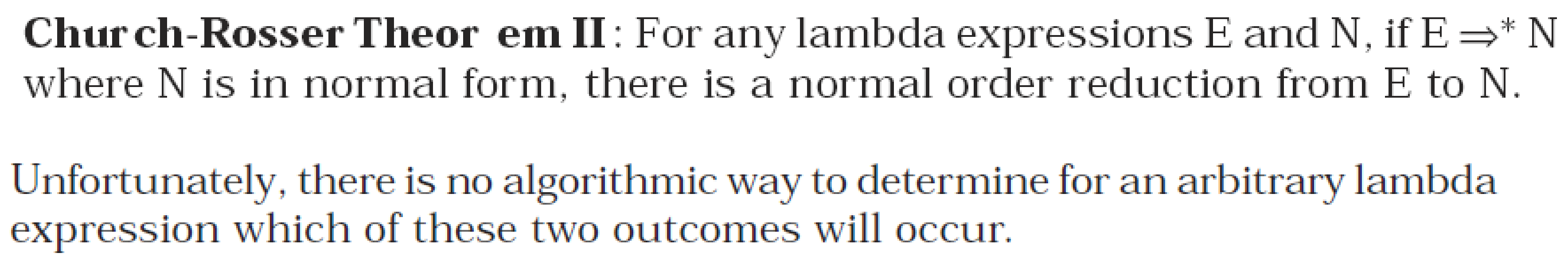
Some evaluations may terminate while others may diverge. If two evaluations terminate, it will be to the same normal form.

DIAMOND PROPERTY (confluence)

NORMAL ORDER REDUCTION

A normal order reduction can have either of the following outcomes

- 1. It reaches a unique (up to ɑ-conversion) normal form lambda expression
- 2. It never terminates



Church-Rosser Theorem II: For any lambda expressions E and N, if $E \Rightarrow^* N$ where N is in normal form, there is a normal order reduction from E to N.

Unfortunately, there is no algorithmic way to determine for an arbitrary lambda expression which of these two outcomes will occur.

THE CONJECTURE

- **Church's Thesis :** The effectively computable functions on the positive integers are precisely those functions definable in the pure lambda calculus (and computable by Turing machines).

- The conjecture cannot be proved since the informal notion of "effectively computable function" is not defined precisely.

- But since all methods developed for computing functions have been proved to be no more powerful than the lambda calculus, it captures the idea of computable functions

REDUCTION STRATEGIES

- Can every lambda expression be reduced to a normal form?
- Is there more than one way to reduce a particular lambda expression?
- If there is more than one reduction strategy, does each one lead to the same normal form expression?
- Is there a reduction strategy that will guarantee that a normal form expression will be produced?

TURING MACHINE

•Turing machines are abstract machines designed in the 1930s by Alan Turing to model computable functions. It has been shown that the lambda calculus is equivalent to Turing machines in the sense that every lambda expression has an equivalent function defined by some Turing machine and vice versa.

Is there a reduction strategy that will guarantee that a normal form expression will be produced?

- Alan Turing proved a fundamental result, called the undecidability of the **halting problem,** which states that there is no algorithmic way to determine whether or not an arbitrary Turing machine will ever stop running. Therefore, there are lambda expressions for which it cannot be determined whether a normal order reduction will ever terminate.
- But we might want to fix the order of evaluation when we model a certain language

CALL BY NAME

- Similar to Normal Order reduction
- Do not evaluate the argument prior to call
- Example:
  - $(\lambda y.\ (\lambda x.\ x)\ y)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta n}$
  - $(\lambda x.\ x)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta n}$
  - $(\lambda u.\ u)\ (\lambda v.\ v) \rightarrow_{\beta n}$
  - $\lambda v.\ v$

- There is no evaluation of the actual parameter at the moment of the call
- The actual parameter will be evaluated only during the execution of the body, if needed

```
if (x==0 || 1/x > 0.3) ... // C++
```

- Does not give any exception for divide by zero

CALL BY VALUE

- Same as Applicative order reduction
- Evaluate an argument prior to call

Example:

$(\lambda y.\ (\lambda x.\ x)\ y)\ ((\lambda u.\ u)\ (\lambda v.\ v)) \rightarrow_{\beta v}$

$(\lambda y.\ (\lambda x.\ x)\ y)\ (\lambda v.\ v) \rightarrow_{\beta v}$

$(\lambda x.\ x)\ (\lambda v.\ v) \rightarrow_{\beta v}$

$\lambda v.\ v$

SHORT-CIRCUITING

- Call by value cannot be used to implement non-strict operators, because the parameters are evaluated at the moment of the call (even if not needed)
- Call by reference or call by value-result (supported by Ada) cannot be used either, because the actual parameters (in general) must be variables
- Even if the language allows to use generic expressions in call by reference (like C++), they are evaluated first and stored in temporary variables
- Hence the non-strictness effect cannot be achieved

```
void p(int value-result x, int value-result y){
   x = x + 1;
   y = y + 1;
}
```

The following is an example of a call to p:

```
int z = 0;
p(z,z);
cout << z; // prints 1
```

CALL BY NAME AND CALL BY VALUE

CBN

- difficult to implement
- order of side effects not predictable

CBV:

- easy to implement efficiently
- might not terminate even if CBN might terminate
- Example: $(\lambda x.\ \lambda z.z)\ ((\lambda y.\ yy)\ (\lambda u.\ uu))$

Outside the functional programming language community, only CBV is used

http://www.lix.polytechnique.fr/~catuscia/teaching/cg428/02Spring/lecture_notes/L07.1.html#:~:text=Call%20by%20nee
d%20is%20similar,evaluated%20each%20time%20is%20needed.

LAZY EVALUATION-CALL BY NEED

It is an evaluation strategy that combines on-demand evaluation with memoisation

- When a variable is bound, it is stored in the environment in an unevaluated form called a thunk, with the evaluation being delayed until the result is required to proceed
- When the thunk is evaluated, it is replaced by the result of its evaluation, avoiding any repetition of work if the value of the variable were to be needed again
- Benefits
  - A more compositional programming style
    - No need to ever construct the entire intermediate results
  - Infinite data and circular definitions
    - This would lead to non-termination in a strict language

Jennifer Hackett and Graham Hutton. 2019. Call-by-Need Is Clairvoyant Call-by-Value. Proc. ACM Program. Lang. 3, ICFP, Article 114 (August 2019), 23 pages. https://doi.org/10.1145/3341718

CALL-BY-NEED

- Problems
- The key issue with understanding lazy evaluation is that while the external interface provided to the programmer is declarative and pure, the internal implementation is stateful.
- Evaluating a term can result in updates to pre-existing environment variables
- The three benefits of better termination, faster evaluation and infinite data structures together represent the **GOOD** of laziness
- The difficulty of reasoning about efficiency represents the **BAD** of laziness.
- The need to give a stateful interpretation to an apparently pure language is downright **UGLY**!

COMPARISON

- In call by need the actual parameter is evaluated only the first time it is needed
- Then the value is stored and used whenever it is needed again
- In call by name, on the contrary, the parameter is re-evaluated each time it is needed
- call-by-value is less terminating than call-by-need, as there could be some diverging subterm that a call-by-need strategy would avoid evaluating
- call-by-value is typically favoured by language designers because it is much easier to understand and reason about

LAZY VS EAGER | | FUNCTIONAL VS IMPERATIVE

- ❑C# and Java have adopted lambda expressions and added streams
- ❑The marriage of paradigms is not always a happy one
- ❑ laziness and parallelism expose a deep rift between functional and imperative programming
- ❑ The delayed or parallel evaluation of an impure function may cause its side effects to be lost, to occur out-of-order, or to interfere with each other, leading to potentially disastrous consequences.

Magnus Madsen, Jaco van de Pol , Programming with Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism 37th European Conference on Object-Oriented Programming (ECOOP 2023). Editors: Karim Ali and Guido Salvaneschi; Article No. 18; pp. 18:1–18:27

LAZY VS EAGER || FUNCTIONAL VS IMPERATIVE

- For these reasons, imperative programming languages tend to use eager and sequential semantics everywhere, thus foregoing the potential benefits of lazy and/or parallel evaluation.
- Most mainstream languages, such as Java, Kotlin, and Scala, offer access to a limited form of laziness and parallelism with streams.
- Yet anarchy reigns: the use of side effects in streams can have unpredictable consequences and nothing prohibits stream operations from having side effects, except for stern warnings in the documentation.

LAZY VS EAGER || FUNCTIONAL VS IMPERATIVE

- List.map when given a *pure* function it lazily maps the function over the list, whereas when given an *impure* function it eagerly maps the function over the list.
- List.map ensures that side effects are never lost or re-ordered while simultaneously allowing lazy evaluation for pure functions.
- We say that the List.map function is *purity reflective*.
- Similarly, Set.count can vary its behavior as follows:
- When given a *pure* function it performs the counting in parallel over the set, whereas when given an *impure* function it performs the counting sequentially following the order of the elements in the set.

LAZY VS EAGER || FUNCTIONAL VS IMPERATIVE

- Thus, Set.count ensures that side effects do not lead to thread-safety hazards (like deadlocks, race conditions), while still admitting parallel evaluation when given a pure function.
- Purity reflection empowers programmers, and in particular library authors, to write new data structures that selectively use lazy and/or parallel evaluation under the hood, while semantically appearing to their clients *as-if* always under eager, sequential evaluation.

Definition: η-reduction

If v is a variable, E is a lambda expression (denoting a function), and v has no free occurrence in E,

$$\lambda v\ .\ (E\ v)\ \Rightarrow_\eta\ E.$$ ▮

$$\lambda x\ .\ (\mathrm{sqr}\ x)\ \Rightarrow_\eta\ \mathrm{sqr}$$

$$\lambda x\ .\ (\mathrm{add}\ 5\ x)\ \Rightarrow_\eta\ (\mathrm{add}\ 5).$$

The η-reduction rule can be used to justify the extensionality of functions; namely, if f(x) = g(x) for all x, then f = g

The rule fails when E represents some constants

λx.(5 x) is not 5

Extensionality Theor em: If $F_1\ x \Rightarrow^* E$ and $F_2\ x \Rightarrow^* E$ where $x \notin FV(F_1\ F_2)$, then $F_1 \Leftrightarrow^* F_2$ where $\Leftrightarrow^*$ includes η-reductions.

Definition: δ-reduction

If the lambda calculus has predefined constants (that is, if it is not pure), rules associated with those predefined values and functions are called δ rules; for example, (add 3 5) $\Rightarrow_\delta$ 8 and (not true) $\Rightarrow_\delta$ false. ▮

LAMBDA CALCULUS TO PROGRAMMING

Data Types

- Booleans, numbers
- Collections

Conditional expressions

Arithmetic expressions

Recursions

a ***combinator*** is a λ-term with no free variables

PAIR

λleft. λright. (left)(right)

get_left = λleft. λright. (true)(left)(right)

get_right = λleft. λright. (false)(left)(right)

make_pair = λleft. λright. λf. f(left)(right)

make_pair(sock1)(sock2)

get_right(make_pair(sock1)(sock2))

get_right = λpair. pair(false)

GET_LEFT = ΛPAIR. PAIR(TRUE)
GET_RIGHT = ΛPAIR. PAIR(FALSE)
MAKE_PAIR = ΛLEFT. ΛRIGHT. ΛF. F(LEFT)(RIGHT)

get_right(make_pair(three)(two))

= get_right(λf. f(three)(two))

=(λpair. pair(false))(λf. f(three)(two))

=(λpair. pair(false))(λf. f(three)(two))

=(λf. f(three)(two)) (false)

=(false) (three)(two)

=two

LISTS

A list may contain

- Nothing (empty)
- One thing
- Multiple things

List contains 2 values

- make_pair = λleft. λright. λf. f(left)(right)
- get_left = λpair. pair(true)
- get_right = λpair. pair(false)

A list will have the form # (empty, (head, tail))

null=make_pair(true)(true)

is_empty = get_left

is_empty(null) would return true

null=make_pair(true)(true)

(prepend(two)(single_item_list))

LISTS

prepend = λitem. λl. make_pair(false)(make_pair(item)(l))

non-empty lists

- *[2, 1] ==> (empty=false, (2,(1,null)))*
- *# (false, (1,null))*
  - single_item_list = prepend(one)(null)
- *# (false, (3,(2,(1,null))))*
  - multi_item_list = prepend(three)(prepend(two)(single_item_list))

head = λl. get_left(get_right(l))

tail = λl.get_right(get_right(l))

LISTS

```
hours_day_1=prepend(2)(null)
hours_per_day=prepend(three)(prepend(two)(hours_day_1))
head(tail(hours_per_day)
```

PREDECESSOR

- The general strategy will be to create a pair (n,n-1) and then pick the second element of the pair as the result
- make_pair = λleft. λright. λf. f(left)(right)

Left=λpair. pair(true)

Right= λpair. pair(false)

Φ combinator generates from the pair (n, n-1) (which is the argument p in the function) the pair (n + 1, n)

Φ =λp. λf. f (succ(p true))(p true)

(succ(p true))(p true)

(zero,zero)→(one,zero)→(two,one)→(three,two)→…

The predecessor of a number n is obtained by applying n times the function to the pair (f zero zero) and then selecting the second member of the new pair

Pred=λn. n Φ (λg. g zero zero)(false)

Φ =λp. Pair (snd p)(succ(snd p))

Pred= λn.fst(n Φ (pair(zero zero))

ENCODING NATURAL NUMBERS IN LAMBDA CALCULUS

What can we do with a natural number?

- we can iterate a number of times

A natural number is a function that given an operation f and a starting value s, applies f a number of times to s:

$1 =_{def} \lambda f.\ \lambda s.\ f\ s$

$2 =_{def} \lambda f.\ \lambda s.\ f\ (f\ s)$

and so on

$0 =_{def} \lambda f.\ \lambda s.\ s$

pred=λn. n ϕ (λf. f zero zero)(false)
ϕ =λp. λf. f (succ(p true))(p true)

Pred one=(λn. n Φ (λf. f zero zero)(false))(one)

=one Φ (λf. f zero zero)(false)

= (λuv.uv) Φ (λf. f zero zero)(false)

= Φ (λf. f zero zero)(false)

= (λp. λf. f (succ(p true))(p true))(λf. f zero zero)(false)

= (λp. λf. f (succ(p true))(p true))(λg. g zero zero)(false)
= (λf. f (succ((λg. g zero zero) true)((λg. g zero zero) true))(false)

= (λf. f (succ (zero))(zero))(false)

=(false) (succ (zero))(zero)

=zero

COMPUTING WITH NATURAL NUMBERS

$$1 \equiv \lambda sz.s(z)$$

$$2 \equiv \lambda sz.s(s(z))$$

```
var successor = n => f => x => f(n(f)(x));
```

The successor function

successor n $=_{def}$ $\lambda n.\lambda f.\lambda x.f(nfx)$

Successor of 0 (S0) is $_{def}$ $(\lambda nfx.f(nfx))\ (\lambda sz.z)$

$$\lambda yx.y((\lambda sz.z)yx) = \lambda yx.y((\lambda z.z)x) = \lambda yx.y(x) \equiv 1$$

Successor of 1 (S1) is $_{def}$ $(\lambda wyx.y(wyx))\ (\lambda sz.s(z))$

Addition

$_{def}$ $\lambda m.\lambda n.\lambda f.\lambda x.m(f)(n(f)(x))$

2S3 is $_{def}$ $(\lambda sz.s(s(z)))(\lambda wyx.y(wyx))(\lambda uv.u(u(u(v))))$

λn.n(λx. false)(true)

SUBTRACTION AND COMPARISON

Subtraction: n-m

- λm. λn. mPRED n

Comparison

- λn. λm. isZero(subtract n m)
  - If the predecessor function applied n times to m yields zero, then it is true that??
    - *greaterOrEqual*
- *lessOrEqual*= λn. λm. isZero(subtract m n)
- *areEqual*= λn. λm. AND (*greaterOrEqual n m*) (*lessOrEqual n m*)

DIVISION

```
if(a>=b) then
      return 1+ (a-b)/b;
else
   return 0
```

if_then_else=$_{\text{def}}$ λ*cond*.λ*then_do*. λ*else_do*. Cond (*then_do*) (*else_do*)

The problem here is that we need to express recursion without explicitly calling itself

- *a/b*
  - *if a>=b then 1+ (a-b)/b else 0*

divide=λa. λb. if_then_else(*greaterOrEqual a b*) (succ ((zero)

divide= λa. λb. if_then_else(*greater b a*) (zero) (succ (*self (subtract a b) b*)

- divide seven three
- if_then_else(*greaterOrEqual seven three*) (succ(*self (subtract seven three) three*) (zero)
- (succ(*self (subtract seven three) three*)
- *(succ* (if_then_else(*greaterOrEqual four three*) (*succ*(*self (subtract four three) three*) (zero)))
- *(succ((succ(self (subtract four three) three)))*
- *(succ((succ(*if_then_else(*greaterOrEqual one three*) (*succ*(*self (subtract one three) three*) (zero))))
- *(succ(succ(*zero)))

LITTLE BIT OF CREATIVITY + LITTLE BIT OF ELEGANCE

Self application

- sa = λx. x x

This function takes an argument x, which is apparently a function

- Loop : (λx. x x) (λx. x x)

- Ω=(λx. x x) (λx. x x)
- The Omega Combinator is just the simplest function which infinitely recurs without calling itself.

- Y = λf. (λx. f (x x)) (λx. f (x x))

https://youtu.be/BC8ZAMwfwi4

Y COMBINATOR

Yt=t(Yt)=t(t(Yt))=…

Y combinator can be defined as

- `Y = λt. (λx. t (x x)) (λx. t (x x))`

```
Yz=(λt. (λx. t (x x)) (λx. t (x x)))z
```

```
     =(λx. z (x x)) (λx. z (x x))
```

```
Yz =(λx. z (x x)) (λg. z (g g))
      = z (λg. z (g g)) (λg. z (g g))
       =z(Yz)
       =z((λg. z (g g)) (λh. z (h h)))
       =z(z((λh. z (h h)) (λh. z (h h))))
       =z(z(Yz)…
```

Y COMBINATOR

Y combinator can be defined as

- Y = λf. (λ x. f(x x)) (λx. f (x x))

Yf=f(Yf)=f(f(Yf))=…

$$\begin{aligned}\textbf{define}\quad \text{factorial} \quad &= \quad \lambda n.\,\text{if}\ (=n\ 1)\ 1\\ &\qquad\qquad (*\ n\ (\text{factorial}\ (-\ n\ 1)))\end{aligned}$$

```
T= λn. If_then_else(isZero n)one (mult n(Fact(pred n))
```

Fact:=Y (λ f. λ n. `If_then_else` (isZero n) one (mult n(f(pred n))

CALCULATING FACTORIAL

```
T= (λ f. λ n. If_then_else (isZero n) one (mult n(f(pred n))
Fact=YT  Fact 2= (YT) two
=T(YT)two
= (λ f. λ n. If_then_else (isZero n) one (mult n(f(pred n)) (YT) two
= mult two (YT(one))
=mult two (T(YT)one)
=mult two ((λ f. λ n. If_then_else (isZero n) one (mult n(f(pred n)YT one)
=mult two (mult one(YT zero))
=mult two (mult one one)
```

SUMMATION

To compute sum of natural numbers from 0 to n

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i.$$

$$\mathsf{R} \equiv (\lambda rn.\mathsf{Z}n0(n\mathsf{S}(r(\mathsf{P}n))))$$

- Zn0 : if n==0 then the result of the sum is 0
- else the successor function is applied n times through the recursive call (r)

FIBONACCI SERIES

$$\boxed{\begin{array}{ll} F(n)=f(n-1)+f(n-2) & \text{if } n>2 \\ \quad =1 & \text{else} \end{array}}$$

Y := λf.(λx.f (x x)) (λx.f (x x))

F := Y (λf. λn.if_then_else (LessThanOrEqual two) one))

YF three

F(YF) three

TAIL RECURSION

Tail recursion is a situation where a recursive call is the last thing a function does before returning, and the function either returns the result of the recursive call or (for a procedure) returns no result. A compiler can recognize tail recursion and replace it by a more efficient implementation.

RECURSIVE SUM

```
function recsum(x) {
    if (x === 0) {
        return 0;
    } else {
        return x + recsum(x - 1);
    }
}
```

```
function tailrecsum(x, running_total = 0) {
    if (x === 0) {
        return running_total;
    } else {
        return tailrecsum(x - 1, running_total + x);
    }
}
```

- If the continuation is empty and there are no backtrack points, nothing need be placed on the stack; execution can simply jump to the called procedure, without storing any record of how to come back. This is called LAST–CALL OPTIMIZATION

- A procedure that calls itself with an empty continuation and no backtrack points is described as TAIL RECURSIVE, and last–call optimization is sometimes called TAIL–RECURSION OPTIMIZATION

Is_zero one=?

Is_zero true=?

TYPED LAMBDAS

- Let us exclude single names as expressions, the only objects are functions which take function arguments and return function results
- For the moment, we won’t consider nonterminating applications
- We have constructed functions which we can interpret as boolean values, Boolean operations, numbers, arithmetic operations and so on but particular functions have no intrinsic interpretations other than in terms of their effects on other functions
- Because functions are so general, there is no way to restrict the application of functions to specific other functions, for example we cannot restrict ‘arithmetic’ functions to ‘numeric’ operands
- We can carry out function applications which are perfectly valid but have results with no relevant meaning

Is_Zero= (λn.n (λx. false)true)

NEED FOR TYPES

- In most CPUs, the sole objects are undifferentiated bit patterns which have no intrinsic meanings but are interpreted in different ways by the machine code operations applied to them.
- For example, different machine code instructions may treat a bit pattern as a signed or unsigned integer, decimal or floating point number or a data or instruction address.
- Thus, a machine code program may twos-complement an address or jump to a floating point number.

TYPED VS UNTYPED

- It is claimed that this ‘freedom’ from types makes languages more flexible.
- It does ease implementation dependent programming where advantage is taken of particular architectural features on particular CPUs through bit or word level manipulation
- This in turn leads to a loss of portability because of gross incompatibilities between architectures.
- For example, many computer memories are based on 8 bit bytes so 16 bit words require 2 bytes.
- However, computers differ in the order in which these bytes are used: some put the top 8 bits in the first byte but others put them in the second byte.
- Thus, programs using ‘clever’ address arithmetic which involves knowing the byte order won’t work on some computers.
- ‘Type free’ programming also increases incomprehensible errors

INTRODUCTION TO TYPED LAMBDAS

- ❑ Types are introduced into languages to control the use of operations on objects so as to ensure that only meaningful combinations are used
- ❑ In weakly typed languages, like LISP and Prolog, objects are typed but variables are not.
- ❑ There are restrictions on object/operation combinations but not on variable/object associations.

- ❑A type is a collection of computational entities that share some common property.
- ❑ A type specifies a class of objects and associated operations
- ❑For example, the type int represents all expressions that evaluate to an integer
- ❑ The type int $\longrightarrow$ int represents all functions from integers to integers.
- ❑The difference between types and sets is that types are *syntactic* objects, i.e., we can speak of types without having to speak of their elements. We can think of types as *names* for sets

CONSTRUCTING TYPES

- A type specifies a class of objects and associated operations
- Object classes may be defined by listing their values, for example for booleans:

```
TRUE is a boolean

FALSE is a Boolean
```

- Or by specifying a base object and a means of constructing new objects from the base,
- For example for natural numbers:

```
0 is a number

SUCC N is a number

if N is a number
```

Operations may be specified exhaustively with a case for each base value, for example for booleans, negation:

```
NOT TRUE = FALSE

NOT FALSE = TRUE
```

Operations may also be specified constructively in terms of base cases for the base objects and general cases for the constructive objects.

For example for natural numbers, the predecessor function:

```
PRED 0 = 0

PRED (SUCC X) = X
```

TYPES

- ❑ Even though the lambda calculus is untyped, a large majority of the lambda terms that we look at can be given types
- ❑ In fact, looking at the types of the terms provides insight into the kind of functions these terms represent
- ❑ So, wherever possible, we mention the types of the functions. We use capital letters A, B, . . . to represent arbitrary types and the → symbol to represent function types.
- ❑ A → B represents the type of functions from A to B, i.e., functions that given A-typed arguments, return B-typed results. (predicates:object→Boolean)
- ❑ We use a bracketing convention to parse type expressions with multiple → symbols

TYPED OBJECTS

We are going to construct functions to represent typed objects

In general, an object will have a type and a value

We need to be able to:

- i) construct an object from a value and a type
- ii) select the value and type from an object
- iii) test the type of an object

We will represent an object as a type/value pair

```
def make_obj type value = λs.(s type value)
```

https://sookocheff.com/post/fp/typed-lambda-calculus/
Chapter 5, AN INTRODUCTION TO FUNCTIONAL ROGRAMMING THROUGH LAMBDA CALCULUS
Greg Michaelson, Heriot-Watt University, Edinburgh, UK

```
def selectSecond=λfirst.λsecond.second
def value obj =obj selectSecond
```

EXTRACTING TYPE AND/OR VALUE

```
def selectFirst=λfirst. λsecond. first
def type obj=obj selectFirst
```

we can use these functions to define a (type, value) pair and then access the type

```
def myObj ⟨type⟩ ⟨value⟩=λs.(s ⟨type⟩⟨value⟩)

type myObj=myObj selectFirst
          =λs.(s ⟨type⟩⟨value⟩) selectFirst
          =(selectFirst ⟨type⟩⟨value⟩)
          =(λfirst.λsecond.first ⟨type⟩ ⟨value⟩)
          =(λsecond.⟨type⟩) ⟨value⟩
          =⟨type⟩
```

Once types are defined, however, we should only manipulate typed objects with typed operations to ensure that the type checks aren't overridden.

A type is generally denoted by a number and type checking as a predicate returning a boolean

TYPE BOOLEAN

- We will represent the boolean type as one:

```
def  bool_type = one
```

- Constructing a boolean type involves preceding a boolean value with bool_type:

```
def MAKE_BOOLEAN = make_obj bool_type
```

- which expands as:

```
λvalue.λs.(s bool_type value)
```

- We can now construct the typed booleans TRUE and FALSE from the untyped versions by:

```
def TRUE = MAKE_BOOLEAN true
```

- which expands as:

```
λs.(s bool_type true)
```

TYPE BOOLEAN

We will use numbers to represent types and numeric comparison to test the type

```
def istype t obj = equal (type obj) t
```

```
def FALSE = MAKE_BOOLEAN false
```

- which expands as:

```
λs.(s bool_type false)
```

- The test for a boolean type involves checking for bool_type:

```
def isbool = istype bool_type
```

- This definition expands as:

```
λobj.(equal (type obj) bool_type)
```

SELF APPLICATION IN TYPED LAMBDA CALCULUS

- Even though self-application allows calculations using the laws of the lambda calculus, what it means conceptually is not at all clear
- We can see some of the problems by just trying to give a type to sa = λx. x x.
- Suppose the argument x is of type A.
- But, since x is being applied as a function to x, the type of x should be of the form A → . . ..
- How can x be of type A as well as A →B . . .?
- Is there a type A such that A = (A → B)?
- In traditional mathematics (set theory), there is no such type.
- The concept of “domains” which can be used to represent types (instead of traditional sets)
- This led to the development of an elegant theory of domains, which serves as the foundation for the mathematical meaning of programming languages.

https://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/lambda-calculus-handout.pdf

OBJECTS IN LAMBDA CALCULUS

- ❑Self application is used very fundamentally in implementing object-oriented programming languages. Suppose we have an object x with a method m.
- ❑We might invoke this method by writing something like x.m(y).
- ❑Inside the method m, there would be references to keywords like "self" or "this" which are supposed to represent the object x itself.
- ❑One way of solving the problem is to translate the method m into a function m' that takes two arguments: in addition to the proper argument y, the object on which the method is being invoked. So, the definition of m' looks like:
- ❑m' = λ self. λ y. . . . the body of m . . .
- ❑ The method call x.m(y) is then translated as x.m' (x)(y).

OBJECTS IN LAMBDA CALCULUS

- The object x has a collection of such functions encoding the methods.
- The method call x.m(y) is then translated as x.m'(x)(y).
- This is a form of self application.
- The function m', which is a part of the structure x, is applied to the structure x itself.

EXPRESSIVENESS OF LAMBDA CALCULUS

The λ-calculus can express

- data types (integers, booleans, lists, trees, etc.)
- branching (using booleans)
- recursion

This is enough to encode Turing machines

Encodings can be done

But programming in pure λ-calculus is painful

- add constants (0, 1, 2, …, true, false, if-then-else, etc.)
- add types