Arithmetic Expressions

Values terms that have been reduced completely and correspond to mathematical objects

bv ::= true | false

Boolean Values

nv ::= zero | succ nv

Numeric Values (Numerals)

The goal or reduction is to compute the value of a term

Normal Forms

expressions that cannot be further reduced

Not all normal forms of the language Expr are values: they may be meaningless

Example: pred (iszero false)

Redex a part of a term that can be reduced (ex: pred (succ zero))

Reductum the result of reducing a redex (ex: zero)

Reduction Strategies

An expression can contain many redexes

if (iszero zero) then (pred (succ zero))
else (succ (pred zero))

This term contains 3 redexes

A reduction strategy tells us which redex to reduce first

- **Eager Evaluation:**
  Reduce all arguments to normal form before reducing the main expression

  ... ⇝ if true then zero else (succ zero)
  ⇝ zero

- **Lazy Evaluation:**
  Reduce only the arguments that are needed to compute the main expression

  ... ⇝ if true then (pred (succ zero))
  else (succ (pred zero))
  ⇝ pred (succ zero)
  ⇝ zero

We saved one reduction step: we didn't have to reduce the 'else' branch

Normal Form Theorem

Every term reduces (in many steps) to a normal form:

For each term e,
there exists a normal form e'
such that

$$e \rightsquigarrow^* e'$$

Proof:

Every reduction step makes the term smaller, so the reduction procers must terminate after a finite number of steps. ■

Functional Programming

In traditional imperative programming we have:

Program

A set of instruction to be executed sequentially

Data

Values given in input, stored in memory, manipulated during computation, returned as output

Programs and data are distinct

Programs are not modified during computation

In functional programming there is no distinction:

programs = data

Both programs and data are represented by terms/expressions

Computation consists in reduction of terms to normal form

The pure realization of this idea is the λ-calculus

only one kind of objects: λ-terms

they are both data structures and programs

Main idea:

Function definition by Abstraction

Example:

A function f on numbers:

$$f(x) = x^2 + 3$$

With the mapping notation:

$$x \xmapsto{f} x^2 + 3$$

We use the λ-notation:

$$f = \lambda x.\, x^2 + 3$$

the function that maps an argument (the variable x) to its square plus three

Application:

$(f\ 5) = (\lambda x.x^2+3)\ 5$

$\rightsquigarrow 5^2+3 \rightsquigarrow^* 28$

↑ replace the variable x with the argument 5

this is called **β-reduction**

In the λ-calculus we use only abstraction and application

We don't need numbers (3, 5) or operations ($-^2$, +)

They can be defined using abstraction and application!

Syntax of λ-calculus

NBF for λ-terms:

$t ::= x \mid y \mid z \mid \ldots$ variable names

$\mid \lambda x.t$ abstraction

$\mid t\ t$ application

β-reduction relation

$(\lambda x.t_1)\ t_2 \rightsquigarrow t_1[x := t_2]$

substitution

Substitute t_2 for x inside t_1

Substitution is tricky

Careful to do it correctly

(We'll be more precise later)

That's all!

The λ-calculus is a theory of pure functions

NO datatypes, NO operations

Data Structures are themselves realized as pure functions

Examples of simple λ-terms

$\lambda x.x$ identity function

it returns its argument unchanged

$(\lambda x.x)M \rightsquigarrow x[x:=M] = M$

(t_1 t_2)

$\lambda x.\lambda y.x$ first projection

it takes two arguments, returns the first

(Convention: λ associates to the right)

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

Application associates to the left

$t_1\, t_2\, t_3 = (t_1\, t_2)\, t_3$)

$(\lambda x.\lambda y.x)\, M_1\, M_2$

$\rightsquigarrow ((\lambda y.x)[x:=M_1])\, M_2$

$= (\lambda y.M_1)\, M_2$

$\rightsquigarrow M_1[y:=M_2] = M_1$

(Issues with variable capture, we'll talk about it)

$\lambda x.\lambda y.y$ second projection