Simply Typed λ-calculus: λ→

Types: $T ::= o \mid T \to T$ ←

o is the "base type"
it is unspecified
it contains no values

T→T type of functions

Typing Rules:

Variable $$\frac{x:T \in \Gamma}{\Gamma \vdash x:T}$$

Abstraction $$\frac{\Gamma, x:A \vdash t:B}{\Gamma \vdash \lambda x:A.t:A \to B}$$

Application $$\frac{\Gamma \vdash f:A \to B \quad \Gamma \vdash a:A}{\Gamma \vdash fa : B}$$

Reduction Rule: $(\lambda x:A.t)a \rightsquigarrow t[x:A]$

Examples of types:

o, $o \to o$, $(o \to o) \to o$,

$o \to (o \to o)$, $(o \to o) \to (o \to$

↑ also written $o \to o \to o$

↑ also written $(o \to o) \to o \to$

arrows associate to the right

$((o \to o) \to o) \to (o \to o) \to o$

Explanation of the rules

Variable Rule

A variable has the type that is assigned to it by the context

ex:

$$x:0, y:(0\to0)\to0, z:0\to0 \vdash y:(0\to0)\to0$$

Abstraction Rule

$$\frac{\Gamma, x:A \vdash t:B}{\Gamma \vdash \lambda x:A.t : B}$$

The variable x is in the context of the assumption not in context of conclusion

abstracted variables are explicitly typed

Application Rule

$$\frac{\Gamma \vdash f:A\to B \quad \Gamma \vdash a:A}{\Gamma \vdash f\,a : B}$$

Same context for all three judgments

A term can be applied only if it has a function type

It can only be applied to arguments that have its domain type

Example typing derivation

Variable rule with no assumptions

$x:T \in \Gamma$ is just a verification

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f:0\to0,\ x:0 \vdash f:0\to0} \quad \dfrac{}{f:0\to0,\ x:0 \vdash x:0}}{f:0\to0,\ x:0 \vdash f\,x:0}}{f:0\to0,\ x:0 \vdash f\,(f\,x):0}}{f:0\to0 \vdash \lambda x{:}0.\ f\,(f\,x) : 0\to0}$$

$$\vdash \lambda f{:}0\to0.\ \lambda x{:}0.\ f\,(f\,x) : (0\to0)\to0\to0$$

This is a typed version of the Church numeral $\overline{2}$

Every Church numeral can be given the type $(0\to0)\to0\to0$

Church numerals at different types

$\vdash \lambda f{:}T\to T.\,\lambda x{:}T.\,f\,(f\,x) : (T\to T)\to T\to T$

We can call this

The type of naturals (with base T)

$$Nat_T := (T\to T)\to T$$

Exercise:

Derive types for plus, mult

What about exp?

It is impossible to give types to diverging terms:

$(\lambda x{:}?.\,\underline{x\,x})\,(\lambda x{:}?.\,x\,x)$

No type permits self-application

But we also lose the fixed point combinator Y

Other typeable terms:

- identity

  $id_T := \lambda x{:}T.x : T\to T$

- projections

  $\lambda x{:}A.\,\lambda y{:}B.\,x : A\to B\to A$

  $\lambda x{:}A.\,\lambda y{:}B.\,y : A\to B\to B$

- Booleans

  $true := \lambda x{:}T.\,\lambda y{:}T.\,x : T\to T\to T$

  $false := \lambda x{:}T.\,\lambda y{:}T.\,y : T\to T\to T$

  $Bool_T := T\to T\to T$

Properties of $\lambda\!\rightarrow$:

Progress

A typed closed term is either a value or can be reduced

closed term: no free variables

$\vdash t : T$

↑ empty context

Preservation (Subject Reduction)

Reduction preserves types

If $\Gamma \vdash t : T$ and $t \rightsquigarrow^* t'$

then $\Gamma \vdash t' : T$

Normalization

Every term reduces to a (unique) normal form

Strong Normalization

Every evaluation strategy produces a normal form

Confluence

still holds

Proof of progress and preservation: induction on structure of terms

Proof of normalization: more difficult

done by Turing using double induction on types and number of redexes