Simply typed lambda calculus ($\vec{\lambda}$)

by hadeel AL-Daoud

The main thing that typed lambda calculus adds to the un-typed lambda calculus is a concept called **base types**.

For example:

type “N“: the set of natural numbers

type “B" which corresponds to Boolean true/false values; and a type “S" which corresponds to strings.

continue

- **Once we have basic types, then we can talk about the type of a function. A function maps from a value of one type (the type of parameter) to a value of a second type (the type of the return value). For example, for a function that takes a parameter of type “S", and returns a value of type "δ", we write its type as “S → δ". "→" is called the *function type constructor.***

- To apply types to the lambda calculus, we do a couple of things. First, we need a **syntax update** so that we can include type information in lambda terms. And second, we need to add a **set of rules** to show what it means for a typed program to be valid.

- **TYPED LANGUAGE CONCRETE SYNTAX:**

T: Type ($\lambda^{\rightarrow}$)

T := C | T1 -> T2 | (T)

TLCE:= expressions of simply typed lambda calculus

TLCE ::= c | x | λ(x : T) . M | M N | (M)

Example

- **λ (x : N) . x + 3.** This asserts that the parameter, x, has type "N", which is the natural numbers. There is no assertion of the type of the result of the function; but since we know that "+" is a function with type "N -> N", which can infer that the result type of this function will be "N".

- Definition: Set of type judgments E:

  E= {x: T1,x: T2,……..,x: Tn}

If a type context includes the judgment that "*x : T* ", We write that as "E I- x : T ".

E is called static type environment.

To talk about whether a program is valid with respect to types , we need to introduce a set of rules for type inference. When the type of an expression is inferred using one of these rules, we call that a type judgment.

Rule1: identifier:

E U {x: T} |- x: T

The simplest rule: if E indicates that identifier x has type T, Then x has that type.

Rule2: Constant:

E l- c: C

This rule states that a constant has whatever types associated with it in E.

Rule 3: Function

Given: E U {x : T1} |- M: T2

Infer: E |- (*λ* x : T1 . M) : T1 -> T2

This statement allows us to infer function types: if we know the type of the parameter to a function is “T1"; and we know that the type of the value returned by the function is “T2", then we know that the type of the function is “T1 -> T2“ . ***And finally,***

Rule 4: Application

Given: E |- M : T1-> T2, E |- N : T1

Infer: E |- M N : T2

If we know that a function has type “T1 -> T2", and we apply it to a value of type “T1", the result is an expression of type “T2".

Type checking rules example

E_o |- *λ (x: integer) . (plus x) x :* ***??***

B. substitution

1. Occurrences

Definition x occurs in:

(i) x,

(ii) (M N) if x occurs in M or N,

(iii) (λ y: T. M) if x occurs in M.

2. Free variables, Fv

Fv (x) = {x}

Fv(λ x: T. M) = Fv (M) - {x}

Fv (M N) = Fv (M) union Fv (N)

Syntactic substitution:

[N/x] M is the result of replacing all free occurrences of identifier x by N in expression M.

- [N/ x] x = N
- [N/ x] y = y if not (y = x)
- [N/ x] (L M) = ([N/ x] L) ([N/ x] M)
- [N/ x] (**λ** y: T. M) = **λ** (y: T) . ([N/ x] M) where not (y = x), not(y in Fv (N))

Reduction rules:

- $(\beta)(\lambda\,(x{:}\,T).\,M)N \xrightarrow{\beta} [N/\,x]\,M$
- $(\eta)\ \lambda\,(x{:}\,T).\,(M\,x) \xrightarrow{\eta} M$

EXAMPLE

(Λ(x: integer).(Plus x) x)17 **,reduces to:**

(Plus 17) 17 : Using *β- reduction , reduces to:*

34 : using δ- rule

- **So, now we have a simply typed lambda calculus. The reason that it's simply typed is because the type treatment here is minimal: the only way of building new types is through the unavoidable "->" constructor.**

THANKS FOR YOUR ATTENTION