Trabajo Practico 3

Facundo Emmanuel Messulam
Franco Ignacio Vallejos Vigier

4 de noviembre de 2021

1. Ejercicio 1

$\Gamma = \{x : E \to E \to E, y : E \to E, z : E\}$

A) x: E $\to E \to E \in \Gamma$

———————T-VAR

$\Gamma \vdash x : E \to E \to E$

B) y: E $\to E \in \Gamma$

————————-T-VAR

$\Gamma \vdash y : E \to E$

C) z: E $\in \Gamma$

—————-T-VAR

$\Gamma \vdash z : E$

D) (de A) $\Gamma \vdash x : E \to (E \to E)$ (de C) $\Gamma \vdash z : E$

———————————————————————T-APP

$\Gamma \vdash (x\ z) : E \to E$

E) (de B) $\Gamma \vdash y : E \to (E)$ (de C) $\Gamma \vdash z : E$

———————————————————————T-APP

$\Gamma \vdash (y\ z) : E$

F) (de D) $\Gamma \vdash (x\ z) : E \to (E \to E)$ (de E) $\Gamma \vdash (y\ z) : E$

———————————————————————————————T-APP

$\Gamma \vdash (x\ z)\ (y\ z) : E$

G) (de F) $\Gamma, z : E \vdash (x\ z)\ (y\ z) : E$

———————————————T-ABS

$\Gamma \vdash (\lambda z : E.(x\ z)\ (y\ z)) : E \to E$

H) (de G) $\Gamma, y : E \to E \vdash (\lambda z : E.(x\ z)\ (y\ z)) : E \to E$

———————————————————————————————— T-ABS

$\Gamma \vdash (\lambda y.E \to E.(\lambda z : E.(x\ z)\ (y\ z))) : (E \to E) \to E \to E$

I) (de H) $\Gamma, x : E \to E \to E \vdash (\lambda y.E \to E.(\lambda z : E.(x\ z)\ (y\ z))) : (E \to E) \to E \to E$

—— T-ABS

$\Gamma \vdash (\lambda x : E \to E \to E.(\lambda y : E \to E.(\lambda z : E.(x\ z)\ (y\ z)))) : (E \to E \to E) \to (E \to E) \to E \to E$

2. Ejercicio 2

Usamos en la función `infer` un valor de retorno `Either String Type` y no directamente un valor `Type`, por el control de errores que se pueda dar en el proceso de la inferencia de tipos. Por ejemplo cuando se quiere inferir el tipo de variables no definidas o en el caso de las funciones si el tipo esperado no coincide, o si se quiere aplicar algo que no es una función definida aun, ya que al no serlo no vamos a poder inferir el tipo.

(`>>=`) El operador realiza un análisis por casos del argumento v que es de tipo `Either`. Si v es de la forma `Left String`, estamos en presencia de un error y devolvemos directamente `Left String`. (Ya que `Left` es la primera función pasada como parámetro a `either`,la cual indica la función a ser usada en caso de que v sea `Left String`). Si v es de la forma `Right Type`, entonces aplicamos la función a la derecha del operador (en este caso `f`), ya que no encontramos ningún error de tipo (al menos hasta el momento).

3. Ejercicio 3, 4, 6

Hecho en la carpeta `src`.

4. Ejercicio 5

Esto es una implementacion de una version del algoritmo proveida por J.R.B. Cockett:

$$
\begin{aligned}
\mathsf{A\ Zero} &= \mathsf{Succ} \\
\mathsf{A\ (Succ}\ m) &= \mathsf{IT(A}\ m) \\
\mathsf{IT}\ f\ \mathsf{Zero} &= f\ \mathsf{(Succ\ Zero)} \\
\mathsf{IT}\ f\ \mathsf{(Succ}\ n) &= f\ (\mathsf{IT}\ f\ n)
\end{aligned}
$$

```
{- De "Notes on the -calculus" escrito por J.R.B. Cockett -}
def IT = \f: Nat -> Nat. \n: Nat. R (f (suc 0)) (\r: Nat. \i: Nat. f r) n

def s = \x: Nat. suc x
def A = \m: Nat. R s (\r: Nat -> Nat. \i: Nat. IT r) m
```

5. Ejercicio 7

Esta implementación simplemente resta el largo de la lista inicial al largo de la lista restante en cada elemento de la lista.

```
def sum = \x: Nat. \y: Nat. R (x) (\r: Nat. \i: Nat. suc r) (y)
def ant = \x: Nat. R 0 (\v: Nat. \i: Nat. i) (x)
def resta = \x: Nat. \y: Nat. R (x) (\r: Nat. \i: Nat. ant r) (y)
def length = \l: List Nat. RL 0 (\head: Nat. \tail: List Nat. \r: Nat. suc r) (l)
def sumPos = \n: List Nat. RL
                                nil
                                (\head: Nat. \tail: List Nat. \r: List Nat.
                                    (cons
                                       (sum head (resta (length n) (length tail)))
                                       r
                                    )
                                )
                                (n)
```