Trabajo Practico Final
Análisis del Lenguaje de Programación
Sistema F

Ramiro Gatto

21/04/2025

1. Descripción del Proyecto

La idea principal del proyecto es la de implementar un EDSL sobre el Sistema F, el cual permita la evaluación de algunos términos del mismo. Para que el proyecto sea simple se eligieron un par de tipos bases para el evaluador, los cuales son:

- Empty
- Booleanos
- Naturales
- Funciones
- Listas (de cualquier tipo)

Además, para el evaluador también se definió lo siguiente:

- Chequeador de tipos.
- Pretty-printer.
- Parser.

Para poder realizar el mismo se tomo como inspiración el Trabajo Practico Nº2 [1], el cual se extendió/modifico para satisfacer con lo requerido.

2. Manual de uso e Instalación del software

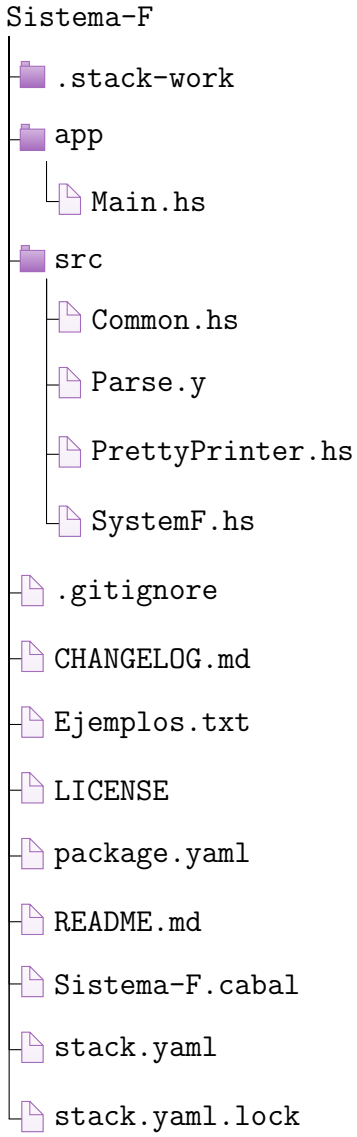
Para poder usar el evaluador se va a necesita de Stack [2], una vez instalado se tiene que abrir una consola en el directorio del proyecto (Sistema-F) y ejecutar:

1. stack setup (una única vez)
2. stack build
3. stack exec Sistema-F-exe

Con las dos primeras lineas compilamos el proyecto y con la tercera lo ejecutamos.

3. Organización de los archivos

La organización de los archivos del proyecto es la siguiente:



Para entender como funciona el proyecto veamos la función de los archivos en las carpetas app y src que son los principales para el funcionamiento del mismo, el resto de los archivos son de configuración.

3.1. app

3.1.1. Main.hs

Este archivo es donde comienza la ejecución del programa al compilarse y ejecutarse (implementa el ejecutable final). Hace uso de una serie de funciones para poder parser la entrada por teclado, determinar el comando ingresado, imprimir por pantalla, realizar la evaluación, entre muchas cosas más.

Gran parte del **Main.hs** es idéntico al del TP Nº2 [1], excepto por algunas modificaciones y simplificaciones para que se ajuste al comportamiento requerido.

3.2. src

3.2.1. Common.hs

En este archivo es donde se definen la forma que tendrán los valores, términos y tipos del Sistema F. Para estos nos basamos en la gramática del mismo.

3.2.2. Parse.y

Para generar el parser utilizaremos happy. En este archivo se generan los parsers a utilizar, se definen los tokens que acepta el parse, entre más funciones. También es donde se define el lexer que se utilizara como analizador lexicográfico de la entrada.

Se tiene 2 parses, uno es el parseStmt en donde el no terminal de nivel superior (top-level non-terminal) es **Def** y el segundo en donde el top-level non-terminal es **Exp**. Para crear el archivo se utilizo parte del Parser.y del TPº2 [1] y la documentación de Happy [3].

3.2.3. PrettyPrinter.hs

Para poder mostrar los términos se utilizo la biblioteca pretty printing, la cual consiste en una serie de combinados desarrollada por John Hughes. Aca es donde se encuentra implementado el pretty printer para el Sistema F.
En este podemos distinguir 2 función importantes:

- printTerm: la cual dado un Term lo imprime por pantalla.
- printType: la cual se encarga de imprimir por pantalla los Type.

3.2.4. SystemF.hs

En este archivo es donde se implementan las funciones de evaluación y el chequeador de tipo (además de unas cuantas funciones auxiliares para el correcto funcionamiento. Más adelante en el informe se explica una de estas funciones, llamada conversionType, la cual puede resultar complicada de entender).

4. Decisiones de diseño importantes

4.1. Representación del Sistema F

Los tipos, términos y valores en el Sistema F están dados por la siguientes gramática, respectivamente:

$$
\begin{aligned}
T &::= E \mid T \to T \mid X \mid \forall X\ .\ T \mid Bool \mid Nat \mid List\ T \\
t &::= x \mid \lambda x : T.\ t \mid t\ t \mid ifthenelse\ t\ t\ t \mid \Lambda X\ .\ t \mid t\ \langle X \rangle \\
v &::= True \mid False \mid nv \mid \lambda x : T.\ t \mid \Lambda X\ .\ t \\
nv &::= 0 \mid suc\ nv
\end{aligned}
$$

Como se menciono anteriormente, la implementación de estos se encuentra en el archivo **src/Common.hs** y es la siguiente:
Para los tipos es:

```
    data Type = EmptyT
              | FunT Type Type
              | BoundForAll Pos
              | VarT String
              | ForAllT Fat
              | BoolT
              | NatT
              | ListTEmpty
              | ListT Type
              deriving (Show, Eq)

data Pos = External Int | Inner Int deriving (Show)
instance Eq Pos where
  External t1 == Inner t2    = t1 == t2
  Inner t1    == External t2 = t1 == t2
  External t1 == External t2 = t1 == t2
```

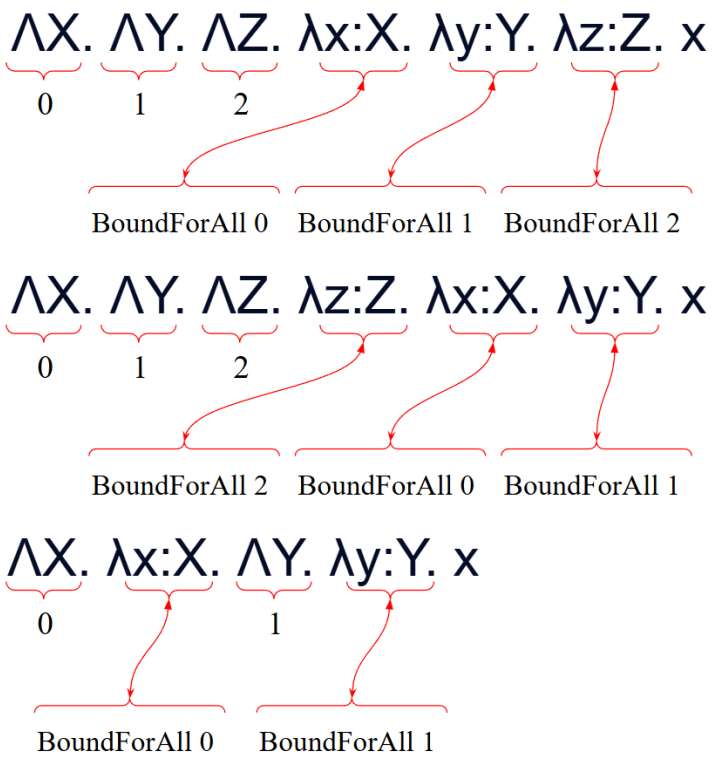
```
  Inner t1    == Inner t2    = t1 == t2

data Fat = Lambd Type | Ty Type deriving (Show)
instance Eq Fat where
    Ty t1    == Lambd t2 = t1 == t2
    Lambd t1 == Ty t2    = t1 == t2
    Ty t1    == Ty t2    = t1 == t2
    Lambd t1 == Lambd t2 = t1 == t2
```

De esta definición se pueden ver muchas cosas, asi que vayamos por parte. Empecemos con la declaración del Type, esta se basa en la gramática del Sistema F, pero posee '2' elementos de más, los cuales son:

- **ListTEmpty**
  Como se van a poder usar listas de todo tipo surge un problema a la hora de usar una lista vacía, ya que se debería especificar el tipo, a pesar de estar vacía. Para evitar y simplificar esto lo que decidió fue darle un tipo especial a la lista vacía, **ListTEmpty**, de esta forma se evita tener que darle un tipo especifico.

  Teniéndose que agregar un caso para lista vacía en las funciones que pueden recibir una como argumento (como infer' o match)

- **BoundForAll**
  En este caso, este no es un tipo per se, sino que su función es similar a la idea de los indices de De Brujin. Esto seria, coloquialmente hablando, indicar a que **para todo** esta ligada la variable cuantificada, en un inicio la idea fue la siguiente:

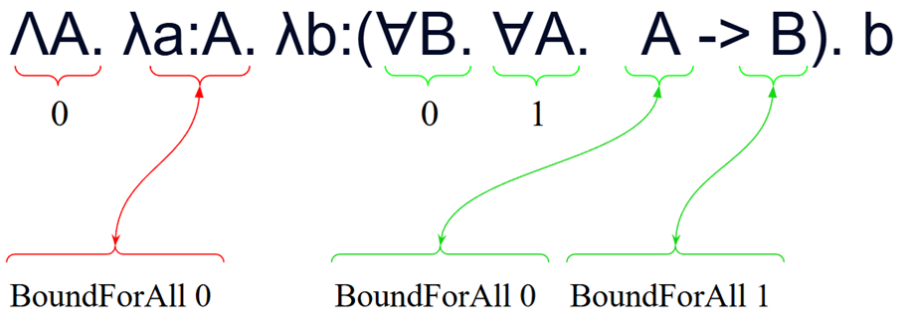


En un inicio la idea era poner (BoundForAll Int) con los Term (como el Bound que esta en Term), pero como esta idea esta relacionada con los tipos resulto más practico agregarlo aca.

Pero a medida que se fue avanzando con el Proyecto la idea de usar solo algo de la forma (BoundForAll Int) resulto insuficiente, ya que podría ocurrir algo como en el siguiente ejemplo:

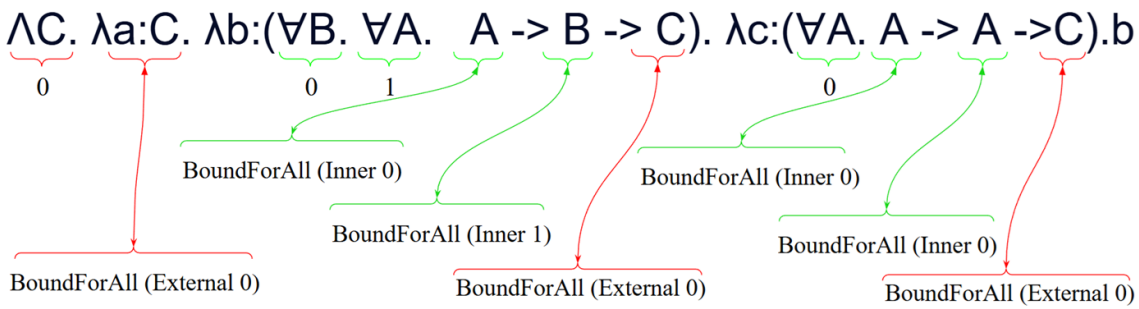
$$\Lambda A.\ \lambda a{:}A.\ \lambda b{:}\ (\forall B.\ \forall A.\ A \to B)$$

Si solo se usa la forma (BoundForAll Int) estos se pondrían en los siguientes lugares:



Donde se puede apreciar una cosa interesantes, se tiene un solo ForAll pero tenemos (BoundForAll 0) y (BoundForAll 1). Uno tendería a pensar que esto es un error, pero no lo es ya que estos BoundForAll no están 'ligados' al Λ sino al ∀.

Debido a esto surge la necesidad de poder distinguir cuando un BoundForAll esta 'dentro' o 'afuera'. Entonces para indicara a que están ligados se usa el tipo Pos, donde External y Inner indican a que Λ y a que ∀ esta ligado respectivamente. Una cuestión es que los External son 'globales' y los Inner 'locales', usemos este ejemplo para ilustrar esto:



En donde el Int que posee el External de ΛC. se respeta en todos lados, mientras el ∀A. se tiene en b con (Inner 1) y en c con (Inner 0). Es decir, el Inner depende del tipo actual, mientras que el External es el mismo sin importar donde este.

Además, hay que tener en cuenta una cosa más, los dos tipos de Pos deben ser iguales entre si, para que se pueda hacer este tipo de sustitución.

(\x:(\/Y. Y -> Y). x) (/\X. \x:X. x)

Ya que en la primera el tipo de x seria algo del estilo, (BoundForAll (Inner 0)) -> (BoundForAll (Inner 0)). Mientras que en la segunda expresión seria de tipo (BoundForAll (External 0)) -> (BoundForAll (External 0)). Por lo que la única forma de poder hacer la sustitución es que Externa e Inner sean iguales (Esto se logra definiendo 'manualmente' el Eq de Pos).

Otra razon para distinguir entre Inner y External es cuando se realize una aplicación de la forma,

/\X. t <TipoNuevo>

asi al reemplazar se reemplazaría los BoundForAll (External k) y no los Inner (lo cual estaría mal).

Una decision de diseño importante fue el **ForAllT Fat**, la idea del tipo **Fat** es la de distinguir el 'Λ' y el '∀' en las expresiones a la hora de imprimir por pantalla, pero que a la vez sean iguales para poder realizar la aplicación. Usemos un ejemplo para ilustrar esto, tenemos la siguiente expresión:

(/\X. \x:X. /\Y. \y:Y. y).

El printType no pude distinguir que variable cuantificada va en cada 'Λ', ya que no se guarda ningún nombre en esta. Para esto se lleva un contador, el cual controla la cantidad de '∀' que aparecen en la expresión. Ahora, si no se hiciese la distinción del **Fat**, en expresiones como la siguiente

(ΛA. λa:A. λb:(∀B. B -> B). b)

también se estaría teniendo en cuenta los '∀' al momento de contar, lo que estaría mal.

Para las expresiones del lambda calculo se tiene:

```
data LamTerm  =  LVar String
              |  LAbs String Type LamTerm
              |  LApp LamTerm LamTerm
              |  LTAbs String LamTerm
              |  LTApp LamTerm Type
              |  LTrue
              |  LFalse
              |  LIfThenElse LamTerm LamTerm LamTerm
```

```
  |  LZero
  |  LSuc LamTerm
  |  LRec LamTerm LamTerm LamTerm
  |  LNil
  |  LCons LamTerm LamTerm
  |  LRecL LamTerm LamTerm LamTerm
 deriving (Show, Eq)
```

Al igual que en el Trabajo Practico 2 [1] surge el problema del uso de nombre de variables, al momento de realizar operaciones como la sustitución. Para arreglar esto se mantiene la misma idea de usar la representación con **indices de De Brujin**.

Al usar una representación sin nombre surge el problema de no tener variables libres, para evitar este inconveniente se utiliza la representación localmente sin nombres (donde variables libres y ligadas están en diferentes categorías sintácticas).

Al utilizar esta representación los términos quedan asi:

```
data Term  = Bound Int
           | Free Name
           | Term :@: Term
           | Lam Type Term
           | ForAll Term
           | TApp Term Type
           | T
           | F
           | IfThenElse Term Term Term
           | Zero
           | Suc Term
           | Rec Term Term Term
           | Nil
           | Cons Term Term
           | RecL Term Term Term
          deriving (Show, Eq)
```

4.1.1. Evaluación

Para la evaluación el interprete sigue el orden de reducción **call-by-value** en donde tenemos las siguientes reglas, las cuales son las presentes en el TP N^{o}2 [1] y en el material de clase del Sistema F [4]:

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2}\ \text{(E-App1)} \qquad \frac{t_2 \to t_2'}{v\ t_2 \to v\ t_2'}\ \text{(E-App2)} \qquad \frac{}{(\lambda x : T_1\ .\ t_1)v \to t_1[x/v]}\ \text{(E-AppAbs)}$$

$$\frac{ifthenelse\ T\ t_2\ t_3}{t_2}\ \text{E-IFTrue} \qquad \frac{ifthenelse\ F\ t_2\ t_3}{t_3}\ \text{E-IFFalse}$$

$$\frac{t_1 \to t_1'}{ifthenelse\ t_1\ t_2\ t_3 \to ifthenelse\ t_1'\ t_2\ t_3}\ \text{E-IF}$$

$$\frac{}{R\ t_1\ t_2\ 0 \to t_1}\ \text{E-RZero} \qquad \frac{}{R\ t_1\ t_2(suc\ t) \to t_2(R\ t_1\ t_2\ t)t}\ \text{E-RSuc} \qquad \frac{t_3 \to t_3'}{R\ t_1\ t_2\ t_3 \to R\ t_1\ t_2\ t_3'}\ \text{E-R}$$

$$\frac{}{RL\ t_1\ t_2\ nil \to t_1}\ \text{E-RNil} \qquad \frac{}{RL\ t_1\ t_2(cons\ t\ l) \to t_2\ t\ l\ (RL\ t_1\ t_2\ l)}\ \text{E-RCons}$$

$$\frac{t_3 \to t_3'}{RL\ t_1\ t_2\ t_3 \to RL\ t_1\ t_2\ t_3'}\ \text{E-RL}$$

$$\frac{t_1 \to t_1'}{cons\ t_1\ t_2 \to cons\ t_1'\ t_2}\ \text{E-Cons1} \qquad \frac{t_2 \to t_2'}{cons\ t_1\ t_2 \to cons\ t_1\ t_2'}\ \text{E-Cons2}$$

$$\frac{t_1 \to t_1'}{t_1\ \langle T\rangle \to t_1'\ \langle T\rangle}\ \text{E-TApp} \qquad \frac{}{(\Lambda X\ .\ t)\ \langle T\rangle \to t[T/X]}\ \text{E-TAppAbs}$$

4.1.2. Tipos

Para realizar la inferencia de tipo usamos las siguientes reglas, las cuales al igual que en la sección anterior son las presentes en el TP Nº2 [1] y en el material de clase del Sistema F [4]:

$$\frac{}{\Gamma \vdash T : Bool}\ \text{T-True} \qquad \frac{}{\Gamma \vdash F : Bool}\ \text{T-False} \qquad \frac{\Gamma \vdash t_1 : Bool \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash ifthenelse\ t_1\ t_2\ t_3\ :\ T}\ \text{T-IF}$$

$$\frac{}{\Gamma \vdash 0 : Nat}\ \text{T-Zero} \qquad \frac{\Gamma \vdash t : Nat}{\Gamma \vdash suc\ t : Nat}\ \text{T-Suc}$$

$$\frac{\Gamma \vdash t_1 : Nat \quad \Gamma \vdash t_2 : T \ \rightarrow \ Nat \rightarrow \ T \quad \Gamma \vdash t_3 : Nat}{\Gamma \vdash R\ t_1\ t_2\ t_3 : T}\ \text{T-Rec}$$

$$\frac{}{\Gamma \vdash nil : ListEmpty}\ \text{T-Nil} \qquad \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : List\ T}{\Gamma \vdash cons\ t_1\ t_2 : List\ T}\ \text{T-Cons}$$

$$\frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T_1 \ \rightarrow \ List\ T_1 \rightarrow \ T \rightarrow \ T \quad \Gamma \vdash t_3 : List\ T_1}{\Gamma \vdash \ RL\ t_1\ t_2\ t_3 \ : \ T}\ \text{T-RL}$$

$$\frac{\Gamma, X \vdash t : T}{\Gamma \vdash \Lambda X\ .\ t : \forall X\ .\ T}\ \text{T-TAbs} \qquad \frac{\Gamma \vdash t_1 : \forall X\ .\ T}{\Gamma \vdash t_1\langle T_2 \rangle : T[T_2/X]}\ \text{T-TApp}$$

4.2. Mostrar términos

Al igual que en el TP Nº2 [1] se va a utilizar la biblioteca **pretty printing** para mostrar por pantalla. En el archivo **src/PrettyPrinter.hs** es donde se implementa el **pretty printing** para el sistema F.

4.3. Explicación de la función conversionType

Decidí que era importante explicar como se comporta esta función ya que, fue la que más me costo de pensar y es una de las mas complejas del código. La idea de esta función es transformar los (VarT X) que aparecen en los tipos por su respectiva forma (BoundForAll Pos), el resto de los tipos se quedan igual. Para esto hay que tener en cuenta unos cuantos factores, como las abstracciones afuera del tipo y las de adentro.

Para las abstracciones de afuera se pasa una lista con todas ellas, creada en el momento de pasar de LamTerm a Term. El problema aparece al buscar las de adentro, ya que al formar un la lista de (VarT) estos pueden hacer referencia, tanto a las abstracciones de afuera como a las de adentro lo que dificulta la tarea. Además, no se puede hacer como las abstracciones de afuera, que al momento de tener algo de la forma (LTAbs name LamTerm) se arma una lista con los name's, ya que en los tipo no hay nombre para los ∀, son de la forma (ForAll (Ty)), lo cual complica la búsqueda.

Entonces lo que se hace es, si el tipo tiene la forma (ForAllT (Ty)) se almacenan las "N"de los (VarT "N"), sin que se repitan y luego se compara la cantidad de (ForAllT (Ty)) y la de (VarT "N") distintos encontradas. De esta toda esta parte se

hace cargo la función conversionForAll.

Aca es donde se complica un poco la cosa, una vez realiza la función conversionForAll hay que fijarse si dentro de la nueva lista que junta los "VarT.[en]contrados hay alguno de afuera, y sacarlo de esta para que no genere problemas (tanto los de afuera como los de adentro son de tipo VarT). Podemos distinguir 2 casos:

- Si la cantidad de (ForAllT (Ty)) dentro coincide con la cantidad de (VarT) encontrados no hay problema.
  Aca tenemos una especie de asunción importante, puede pasar que en la expresión pasada los nombres de los ∀X. no coincidan con los nombres de las variables dentro (Ej: (\/X. \/Y. A -> B)) o que se llamen igual que las que se están usando a fuera (Ej: /\X. \x:(\/X. X -> X). x). En estos casos se interpreta que los nombre en los ∀X. están mal e interpreta que deben ir los de adentro.

- Si se tiene que no coincide es debido a 2 posibles casos,

  - Se usen cuantificadores de afuera
  - Esta mal el tipo y hay variables cuantificadas de más

  Para verificar esto tenemos que ver si la cantidad de variables encontradas es menor o igual que la suma de los cuantificadores de afuera (encontrados hasta ese momento) y los de adentro. Si esto no se cumple se genera un error (hay variables cuantificadas de más), si se cumple hay que sacar de la lista la de los cuantificadores de afuera.

Una vez hecho esto se empieza con el reemplazo de los tipos en la función conversionType'. Dentro de la misma es donde se diferencia el caso del External e Inner. Primero se verifica si es Inner y si no lo es, se verifica si es External, si no es ninguna se genera un error.

4.4. Ejemplos con resultados

Una vez que se haya compilado y ejecutado el programa nos aparecerá en la consola esto:

```
Intérprete de Sistema F
Escriba :help para recibir ayuda.
SF>
```

Luego si se quiere evaluar una expresión del Sistema F, se la ingresa por teclado, se presiona el enter y listo (también hay más opciones como el :print para mostrar los ASTs y el :type para ver el tipo de la expresión, entre muchas otras).

Una cosa a mencionar en el tema de los Bool es que si bien en la gramática el if aparece como **ifthenelse** t_1 t_2 t_3 y tanto en los LamTerm como en los Term también aparece con esta forma, a la hora de escribirlo y mostrarlo por consola es **if** t_1 **then** t_2 **else** t_3, este cambio se hizo por comodidad y para que sea mas legible, es más común escribirlo como la segunda forma que como la primera.

Veamos ejemplos (estos ejemplos están en el archivo Ejemplos.txt para que puedan ser testeados sin problemas por el lector):

4.4.1. Funcion identidad polimorfica

En el Sistema F se escribiría: $\Lambda X.\ \lambda$x:X. x
En la consola escribimos: /\X. \x:X . x
Si quisiéramos evaluarla a un natural escribimos: (/\X. \x:X . x) <Nat> (suc 0)
El cual se reduce a: suc 0

4.4.2. Funcion length para listas polimorfica

En el Sistema F se escribiría: $\Lambda X.\ \lambda xs : List\ X.$ RL 0 (λx:X .ys:List X .r:Nat .$suc\ r$) xs
En la consola escribimos: (/\X. (\xs:List X. RL 0 (\x:X .\ys:List X. \r:Nat. suc r) xs))
Si quisiéramos evaluarla a una lista de funciones polimorficas escribimos: ((/\X. (\xs:List X. RL 0 (\x:X .\ys:List X. \r:Nat. suc r) xs)) <\/X. X -> X>) (cons (/\X. \x:X. x) cons (/\X. \x:X. x) nil)
El cual se reduce a: suc suc 0

(Si se prueba con nil da como resultado 0)

4.4.3. Funcion que toma como argumento una funcion polimorfica

En el Sistema F se escribiría: $\Lambda A.\ \lambda a : A.\ \lambda b : (\forall B.B \to B).\ b$
En la consola escribimos: (/\A. \a:A. \b:(\/B. B -> B) . b)
Si quisiéramos evaluarla podría ser algo asi: (((((/\A. \a:A. \b:(\/B. B -> B) . b) <Nat>) 0) (/ \X. \x:X. x)) <Bool>) T
El cual se reduce a: T

Referencias

[1] Cátedra de Análisis del Lenguaje de Programación. Trabajo practico n^{o}2. *Departamento de Ciencias de la Computación*, 2024.

[2] Mike Pilgrem. Stack documentation, haskell. `https://docs.haskellstack.org/en/stable/`.

[3] Andy Gill and Simon Marlow. Happy documentation, haskell. `https://haskell-happy.readthedocs.io/en/latest/`.

[4] Cátedra de Análisis del Lenguaje de Programación. Polimorfismo. *Departamento de Ciencias de la Computación*, 2024.