Trabajo práctico 3 - λ–cálculo tipado

1. Introducción

El objetivo de este trabajo práctico es familiarizarse con un intérprete de λ–cálculo simple tipado. En las diferentes secciones se deberán implementar extensiones tales como naturales y listas de naturales, etc.
El trabajo se debe realizar en grupos de hasta dos personas y la fecha límite de entrega es el viernes 5 de Noviembre, en donde se debe entregar en forma electrónica (en un archivo comprimido) el contenido de los directorios `src` y `Ejemplos`, usando el sitio de la materia del campus virtual de la UNR (`http://comunidades.campusvirtualunr.edu.ar`).

2. Sobre el intérprete

En la carpeta `TP3` se encuentran los archivos correspondientes al intérprete. El documento `README` detalla la estructura de estos archivos y cómo ejecutarlos.

3. Generador de Parser

Happy es un generador de parsers para Haskell. Happy puede trabajar junto a un analizador lexicográfico (una función que divide la entrada en tokens, que son las unidades básicas de parseo) proporcionado por el usuario.
Un archivo de gramática para Happy contiene usualmente:

- Al comienzo del archivo, la definición de un módulo. Esto no es más que la definición en Haskell de un encabezado para un módulo, el cual escribiremos entre llaves.

  En general cualquier código escrito entre llaves será transcripto textualmente al archivo Haskell generado por Happy.

  ```
  {
  module Parse where
  }
  ```

- Algunas declaraciones, como ser:

  ```
  %monad {P} {thenP} {returnP}
  %name parseStmt Def
  %name parseStmts Defs
  %name term Exp
  %tokentype {Token}
  %lexer {lexer} {TEOF}
  ```

  con los nombres de las funciones de parseo que Happy generará. En este caso, los parsers generados serán 3: *parserStmt*, *parseStmts* y *term*. También se declara el tipo de tokens que el parser aceptará, entre otras cosas.

- A continuación se declaran los posibles tokens:

  ```
  %token
    '='    {TEquals}
    ':'    {TColon}
    '\\'   {TAbs}
    '.'    {TDot}
    '('    {TOpen}
    ')'    {TClose}
    '->'   {TArrow}
  ```

$$\begin{array}{ll} VAR & \{\, TVar\ \$\$ \} \\ TYPE & \{\, TType \} \\ DEF & \{\, TDef \} \end{array}$$

Los símbolos a la izquierda son los tokens y a la derecha tenemos los patrones de Haskell para cada token. La definición del tipo *Token* será dada más adelante.

Los simbolos $$ son marcadores de posición que representa el valor de este token. Normalmente el valor de un token es el token en sí mismo, pero usando $$ se puede especificar alguna componente del token para que sea el valor.

- Ahora escribiremos la gramática

$$\begin{array}{lll} Def & : Defexp & \{\$1\} \\ & \mid Exp & \{Eval\ \$1\} \\ Defexp & : DEF\ \ VAR\ \texttt{'='}\ Exp & \{Def\ \$2\ \$4\} \end{array}$$

  Cada producción consiste en un símbolo no terminal a la izquierda, seguido de dos puntos, seguido por una o más expansiones separadas por |. Cada expansión tiene asociada código Haskell entre llaves.

  En un parser cada símbolo tiene un valor. Definimos el valor de los tokens y ahora la gramática define el valor de los símbolos no terminales en terminos de secuencias de otros símbolos (tanto tokens como no terminales), en producciones como ésta:

$$n : t_1 \ldots t_n E$$

  cada vez que el analizador encuentra los símbolos $t_1 \ldots t_n$, construye el símbolo n y le da el valor E , donde puede referirse a los valores de $t_1 \ldots t_n$ usando los símbolos $\$1, \ldots, \$n$.

- Para resolver ambigüedades en la gramática Happy posee las directivas `%right`, `%left`, y `%nonassoc`. Estas directivas se aplican a una lista de tokens y declaran si un token es asociativo a derecha, a izquierda, o no asociativo, respectivamente. Además el orden de las declaraciones fija un orden de precedencia (de menor a mayor).

  Por ejemplo, si escribimos la siguiente gramática:

$$\begin{array}{ll} Exp : & Exp\ \texttt{'+'}\ Exp\ \{Plus\ \$\,1\ \$\,3\} \\ \mid & Exp\ \texttt{'-'}\ Exp\ \{Minus\ \$\,1\ \$\,3\} \\ \mid & Exp\ \texttt{'*'}\ Exp\ \{\,Times\ \$\,1\ \$\,3\} \\ \mid & Exp\ \texttt{'/'}\ Exp\ \{Div\ \$\,1\ \$\,3\} \end{array}$$

  Happy notificará que ocurren conflictos `shift/reduce`, dado que la gramática es ambigua (por ejemplo, $1 + 2 * 3$ puede parsearse como $1 + (2 * 3)$ o $(1 + 2) * 3$). Esta ambigüedad pueden resolverse especificando el orden de precedencia de los operadores de la siguiente manera:

$$\begin{array}{l} \%\ left\ \texttt{'+'}\ \texttt{'-'} \\ \%\ left\ \texttt{'*'}\ \texttt{'/'} \end{array}$$

- Finalmente, para completar el programa, se necesitan algunas definiciones que se escribirán entre llaves. Se incluirán en esta sección una función que sea invocada en caso que se alcance un error. También se declaran los tipos que representan: las expresiones parseadas y los tokens.

  Aquí es donde declaramos el **lexer** que realizará el analisis lexicográfico de la entrada. El lexer es imsplemente una función que toma la cadena de entrada y la transforma en una lista de tokens. Esta función también se encargará de contar las líneas leídas para que en caso de error se pueda retornar en que línea ha ocurrido.

Se puede encontrar la documentación completa sobre Happy en `http://www.haskell.org/happy/doc/html/`.

4. λ–cálculo simplemente tipado

Los tipos del cálculo implementado son dados por la siguiente gramática:

$$\mathsf{T} ::= \mathtt{E} \mid \mathsf{T} \to \mathsf{T}$$

donde `E` es un tipo básico. Una vez definidos estos, se pueden definir los términos:

$$\mathsf{t} ::= x \mid \lambda x : \mathsf{T}.\ \mathsf{t} \mid \mathsf{t}\ \mathsf{t}$$

Notar que no existe ninguna constante para introducir elementos del tipo `E`, por lo que a este se lo conoce como el tipo *vacío* o *empty*. La implementación de los mismos está en `src/Common.hs`:

data *Type* = *EmptyT* | *FunT Type Type*

data *LamTerm* = *LVar String* | *LAbs String Type LamTerm* | *LApp LamTerm LamTerm*

Los valores del cálculo serán las abstracciones:

$$\mathsf{v} ::= \lambda x : \mathsf{T}.\ \mathsf{t}$$

Notar que el cuerpo de la abstracción queda sin evaluar.
Al igual que en el trabajo práctico anterior, utilizaremos índices de De Bruijn para representar los términos. A diferencia del trabajo práctico anterior, también utilizaremos esta técnica para codificar los valores. Sus implementaciones están dadas por los tipos de datos *Term* y *Value* respectivamente.
Para este cálculo consideraremos una evaluación *call–by–value*, dada por las reglas:

$$\frac{\mathsf{t}_1 \to \mathsf{t}_1'}{\mathsf{t}_1\ \mathsf{t}_2 \to \mathsf{t}_1'\ \mathsf{t}_2} \tag{E-App1}$$

$$\frac{\mathsf{t}_2 \to \mathsf{t}_2'}{\mathsf{v}\ \mathsf{t}_2 \to \mathsf{v}\ \mathsf{t}_2'} \tag{E-App2}$$

$$(\lambda x : \mathsf{T}_1.\ \mathsf{t}_1)\ \ \mathsf{v} \to \mathsf{t}_1\,[x/\mathsf{v}] \tag{E-AppAbs}$$

La implementación esta dada por la función *eval*. Esta hace uso auxiliar de la función *sub*, que realiza la substitución de un término por una variable en otro término:

sub :: *Int* → *Term* → *Term* → *Term*

El primer argumento indica la cantidad de abstracciones bajo la cual se realizará la substitución, el segundo argumento es el término a substituir, y el tercero el término donde se efectuará la substitución.
Las reglas de tipado de nuestro cálculo serán las usuales:

$$\frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}} \tag{T-Var}$$

$$\frac{\Gamma, x : \mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2}{\Gamma \vdash \lambda x : \mathsf{T}_1.\ \mathsf{t}_2 : \mathsf{T}_1 \to \mathsf{T}_2} \tag{T-Abs}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1 \to \mathsf{T}_2 \quad \Gamma \vdash \mathsf{t}_2 : \mathsf{T}_1}{\Gamma \vdash \mathsf{t}_1\ \mathsf{t}_2 : \mathsf{T}_2} \tag{T-App}$$

La inferencia de tipos está dada sobre *Term*, por la función:

infer :: *NameEnv Value Type* → *Term* → *Either String Type*

El argumento de tipo *NameEnv Value Type* nos da el entorno con las definiciones efectuadas durante la interacción. Γ estará dado por este argumento y además por el entorno de las variables (y sus tipos) que se encuentren ligadas por una abstracción en la expresión a tipar (este entorno es el argumento extra en *infer′*).

Ejercicio 1. Dar una derivación de tipo para el término S definido en `Ejemplos/Prelude.lam`.

Ejercicio 2. Explique por qué la función *infer* retorna un valor de tipo *Either String Type* y no un valor de tipo *Type*. Explique el funcionamiento de ($\gg=$).

5. Mostrando Términos

Al mostrar términos es a menudo necesario indentar ciertos subtérminos para hacer más evidente su estructura. Para ello se puede utilizar una biblioteca de *pretty printing*. El GHC provee una biblioteca de combinadores de pretty-printing desarrollada inicialmente por John Hughes [**?**].
Los combinadores se centran alrededor del tipo *Doc*. Algunos de sus combinadores más usuales son:

- *empty* :: *Doc*, representa el documento vacío.
- *text* :: *String* → *Doc*, crea un documento de altura 1, con la cadena argumento.
- *parens* :: *Doc* → *Doc*, encierra el documento entre paréntesis.
- (<>) :: *Doc* → *Doc* → *Doc*, pone un documento al lado de otro.
- *sep* :: [*Doc*] → *Doc*, toma una lista de documentos y los combina horizontalmente separados por un espacio, o verticalmente si no entran horizontalmente.
- *nest* :: *Int* → *Doc* → *Doc*, indenta un documento un número n de posiciones.
- *render* :: *Doc* → *String*, convierte un documento a cadena de texto para poder mostrarlo en pantalla.

En el archivo `src/PrettyPrinter.hs` se encuentra implementado un pretty printer para los términos del lambda cálculo simplemente tipado. En varios de los ejercicios siguientes se les pide extenderlo.

6. λ–cálculo con `let` bindings

En la teoría se ha visto una sencilla extensión del cálculo: utilizar la construcción `let` para introducir definiciones y evitar repetir un subtérmino muchas veces en un término. Para ello modificamos los términos:

$$\mathtt{t} ::= \ldots \mid \mathtt{let}\ x = \mathtt{t}\ \mathtt{in}\ \mathtt{t}$$

Las reglas de evaluación son:

$$\mathtt{let}\ x = \mathsf{v}\ \mathtt{in}\ \mathtt{t} \to \mathtt{t}\,[x/\mathsf{v}] \qquad \text{(E-LetV)}$$

$$\frac{\mathtt{t}_1 \to \mathtt{t}_1'}{\mathtt{let}\ x = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 \to \mathtt{let}\ x = \mathtt{t}_1'\ \mathtt{in}\ \mathtt{t}_2} \qquad \text{(E-Let)}$$

Y su regla de tipado:

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_1 \quad \Gamma, x : \mathsf{T}_1 \vdash \mathtt{t}_2 : \mathsf{T}_2}{\Gamma \vdash \mathtt{let}\ x = \mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 : \mathsf{T}_2} \qquad \text{(T-Let)}$$

Ejercicio 3. Extender el intérprete con la construcción `let` (lexer, parser, pretty-printer, evaluador, etc.). La construcción `let` debe tener la misma precedencia que la abstracción. Extender el algoritmo de inferencia inspirándose en las reglas de tipado.

7. λ–cálculo con naturales

Hasta ahora no hemos introducido ningún tipo interesante en nuestro cálculo: únicamente podemos utilizar el tipo base `E` sin habitantes.
Introduciremos el tipo de datos `Nat`, con el cual representaremos los números naturales. Para ello agregamos la constante `0` y la función `suc` para representar los valores numéricos. Además agregamos la función `R` para consumirlos (en esencia, el operador R en la teoría de funciones recursivas). Los tipos y términos quedan:

$$\mathsf{T} ::= \ldots \mid \mathtt{Nat}$$

$$\mathsf{t} ::= \ldots \mid \mathsf{0} \mid \mathtt{suc}\ \mathsf{t} \mid \mathtt{R}\ \mathsf{t}\ \mathsf{t}\ \mathsf{t}$$

Tendremos nuevos valores, las constantes numéricas:

$$\mathsf{v} ::= \ldots \mid \mathsf{nv}$$

donde nv es:

$$\mathsf{nv} ::= \mathsf{0} \mid \mathtt{suc}\ \mathsf{nv}$$

Las reglas que extienden la evaluación son:

$$\mathtt{R}\ \mathsf{t}_1\ \mathsf{t}_2\ \mathsf{0} \to \mathsf{t}_1 \tag{E-RZero}$$

$$\mathtt{R}\ \mathsf{t}_1\ \mathsf{t}_2\ (\mathtt{succ}\ \mathsf{t}) \to \mathsf{t}_2\ (\mathtt{R}\ \mathsf{t}_1\ \mathsf{t}_2\ \mathsf{t})\ \mathsf{t} \tag{E-RSucc}$$

$$\frac{\mathsf{t}_3 \to \mathsf{t}_3'}{\mathtt{R}\ \mathsf{t}_1\ \mathsf{t}_2\ \mathsf{t}_3 \to \mathtt{R}\ \mathsf{t}_1\ \mathsf{t}_2\ \mathsf{t}_3'} \tag{E-R}$$

Las nuevas reglas de tipado son:

$$\Gamma \vdash \mathsf{0} : \mathtt{Nat} \tag{T-Zero}$$

$$\frac{\Gamma \vdash \mathsf{t} : \mathtt{Nat}}{\Gamma \vdash \mathtt{suc}\ \mathsf{t} : \mathtt{Nat}} \tag{T-Suc}$$

$$\frac{\Gamma \vdash \mathsf{t}_1 : \mathsf{T} \quad \Gamma \vdash \mathsf{t}_2 : \mathsf{T} \to \mathtt{Nat} \to \mathsf{T} \quad \Gamma \vdash \mathsf{t}_3 : \mathtt{Nat}}{\Gamma \vdash \mathtt{R}\ \mathsf{t}_1\ \mathsf{t}_2\ \mathsf{t}_3 : \mathsf{T}} \tag{T-Rec}$$

Ejercicio 4. Extender el intérprete con naturales (lexer, parser, pretty-printer, evaluador, etc.). La precedencia de `suc` debe ser mayor a la de `R`, además ambas precedencias deben ser mayores a la de la abstracción y menores a la de la aplicación. Extender el algoritmo de inferencia inspirándose en las reglas de tipado.

Ejercicio 5. Definir en un archivo `Ejemplos/Ack.lam` la función *Ack*, donde:

$$\begin{array}{ll} Ack & : \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ Ack\ 0\ \ n & = n + 1 \\ Ack\ m\ 0 & = Ack\ (m-1)\ 1 \\ Ack\ m\ n & = Ack\ (m-1)\ (Ack\ m\ (n-1)) \end{array}$$

Verificar que el intérprete la acepte y evalúe adecuadamente.

8. λ–cálculo con listas de naturales

Introduciremos el tipo de datos `List Nat`, con el cual representaremos las listas de naturales. Para ello agregamos los términos `nil` y `cons` para representar los constructores de listas. Además agregamos el término `RL` para consumirlas. Con `RL` modelaremos las funciones recursivas primitivas sobre listas. Los tipos y términos quedan:

$$\mathsf{T} ::= \ldots \mid \mathtt{List\ Nat}$$

$$\mathsf{t} ::= \ldots \mid \mathtt{nil} \mid \mathtt{cons}\ \mathsf{t}\ \mathsf{t}\ \mid \mathtt{RL}\ \mathsf{t}\ \mathsf{t}\ \mathsf{t}$$

Tendremos nuevos valores:

$$\mathsf{v} ::= \ldots \mid \mathsf{lv}$$

donde lv es:

$$\mathsf{lv} ::= \mathtt{nil} \mid \mathtt{cons}\ \mathsf{n}\ \mathsf{lv}$$

Las reglas que extienden la evaluación son:

$$\mathtt{RL}\ \mathsf{t}_1\ \mathsf{t}_2\ \mathtt{nil} \to \mathsf{t}_1 \tag{E-RNil}$$

$$\texttt{RL}\ \mathtt{t_1}\ \mathtt{t_2}\ (\texttt{cons n lv}) \to \mathtt{t_2}\ \texttt{n lv}\ (\texttt{RL}\ \mathtt{t_1}\ \mathtt{t_2}\ \texttt{lv}) \qquad \text{(E-RCons)}$$

$$\frac{\mathtt{t_3} \to \mathtt{t_3'}}{\texttt{RL}\ \mathtt{t_1}\ \mathtt{t_2}\ \mathtt{t_3} \to \texttt{RL}\ \mathtt{t_1}\ \mathtt{t_2}\ \mathtt{t_3'}} \qquad \text{(E-RL)}$$

$$\frac{\mathtt{t_1} \to \mathtt{t_1'}}{\texttt{cons}\ \mathtt{t_1}\ \mathtt{t_2} \to \texttt{cons}\ \mathtt{t_1'}\ \mathtt{t_2}} \qquad \text{(E-Cons1)}$$

$$\frac{\mathtt{t_2} \to \mathtt{t_2'}}{\texttt{cons}\ \mathtt{t_1}\ \mathtt{t_2} \to \texttt{cons}\ \mathtt{t_1}\ \mathtt{t_2'}} \qquad \text{(E-Cons2)}$$

Las nuevas reglas de tipado son:

$$\Gamma \vdash \texttt{nil} : \texttt{List Nat} \qquad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \texttt{Nat} \quad \Gamma \vdash \mathtt{t_2} : \texttt{List Nat}}{\Gamma \vdash \texttt{cons}\ \mathtt{t_1}\ \mathtt{t_2} : \texttt{List Nat}} \qquad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash \mathtt{t_1} : \mathsf{T} \quad \Gamma \vdash \mathtt{t_2} : \texttt{Nat} \to \texttt{List Nat} \to \mathsf{T} \to \mathsf{T} \quad \Gamma \vdash \mathtt{t_3} : \texttt{List Nat}}{\Gamma \vdash \texttt{RL}\ \mathtt{t_1}\ \mathtt{t_2}\ \mathtt{t_3} : \mathsf{T}} \qquad \text{(T-RL)}$$

Ejercicio 6. Extender el intérprete con listas de naturales (lexer, parser, pretty-printer, evaluador, etc.).
La precedencia de `cons` debe ser menor a la de `suc` y mayor a la de `RL`.
La precedencia de `RL` debe ser mayor a la de `R`. Extender el algoritmo de inferencia inspirándose en las reglas de tipado.

Ejercicio 7. Definir la función *sumPos* que dada una lista, le suma a cada elemento de la lista su posición en la lista más uno. Por ejemplo, *sumPos* $[3, 5, 7] = [3 + 1, 5 + 2, 7 + 3]$. Verificar que el intérpreta la acepte y evalúe correctamente.