Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencia de la Computación
Análisis de Lenguajes de Programación

Trabajo práctico 2 - λ–cálculo tipado

1. Introducción

El objetivo de este trabajo práctico es implementar en Haskell un intérprete de λ–cálculo de simple tipado con algunas extensiones tales como naturales, listas de naturales, etc.
El trabajo se debe realizar en grupos de hasta tres personas y la fecha límite de entrega es el 30 de octubrels, en donde se debe entregar en forma electrónica (en un archivo comprimido) el contenido de los directorios `src` y `Ejemplos`, usando el sitio Comunidades de la materia.

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 }
  ```

VAR { *TVar* $$ }
TYPE { *TType* }
DEF { *TDef* }

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

Def : *Defexp* { $1 }
| *Exp* { *Eval* $1 }
Defexp : *DEF VAR* '=' *Exp* { *Def* $2 $4 }

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:

$$n : t_1 \dots t_n E$$

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

- 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:

Exp : *Exp* '+' *Exp* { *Plus* $ 1 $ 3 }
| *Exp* '-' *Exp* { *Minus* $ 1 $ 3 }
| *Exp* '*' *Exp* { *Times* $ 1 $ 3 }
| *Exp* '/' *Exp* { *Div* $ 1 $ 3 }

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 puede resolverse especificando el orden de precedencia de los operadores de la siguiente manera:

% *left* '+' '-'
% *left* '*' '/'

- 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 análisis lexicográfico de la entrada. El lexer es simplemente 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/`.
En este trabajo se provee un parser implementado usando Happy en el archivo Parser.y para un lambda cálculo simplemente tipado y sólo será necesario modificar este archivo para implementar las extensiones al cálculo.

4. Representación de Lambda Términos

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.

4.1. Representación sin nombres

Las variables ligadas en los términos de lambda cálculo escritos usando la representación estándar se reconocen por el uso de nombres. Es decir que si una variable x está al alcance de una abstracción de la forma λx, la ocurrencia de esta variable es ligada. Esta convención trae dificultades cuando se definen operaciones sobre los términos, ya que es necesario aplicar α-conversiones para evitar captura de variables. Por ejemplo, si una variable libre en una expresión tiene que ser reemplazada por una segunda expresión, cualquier variable libre de la segunda expresión puede quedar ligada si su nombre es el mismo que el de alguna de las variables ligadas de la primer expresión, ocasionando un efecto no deseado. Otro caso en el cual es necesario el renombramiento de variables es cuando se comparan dos expresiones para ver si son equivalentes, ya que dos expresiones lambdas que difieren sólo en los nombres de las variables ligadas se consideran equivalentes. En definitiva, implementar los términos lambda usando nombres para las variables ligadas dificulta la implementación de operaciones como la substitución.
Una forma de representar términos lambda cálculo sin utilizar nombres de variables es mediante la representación con *índices de De Bruijn*[1], también llamada *representación sin nombres* [DB72]. En esta notación los nombres de las variables son eliminados al reemplazar cada ocurrencia de variable por enteros positivos, llamados índices de De Bruijn. Cada índice representa la ocurrencia de una variable en un término y denota la cantidad de variables ligadas que están al alcance de ésta y están entre la ocurrencia de la variable y su correspondiente "binder". De esta manera la ocurrencia de una variable indica la distancia al λ que la liga.

Los siguientes son algunos ejemplos de lambda términos (sin tipos) escritos con esta notación:

$$\begin{array}{lcl@{\qquad}lcl} \lambda x.\, x & \mapsto & \lambda 0 & \lambda x.\, \lambda y.\, \lambda z.\, x & \mapsto & \lambda\lambda\lambda 2 \\ \lambda y.\, (\lambda x\ y.x)\ y & \mapsto & \lambda(\lambda\lambda 1)\ 0 & \lambda x.\, \lambda y.\, x\ (\lambda y.\, y\ x) & \mapsto & \lambda\lambda 1\ (\lambda 0\ 2) \end{array}$$

Notar que una variable puede tener asignados diferentes índices de De Bruijn, dependiendo su posición en el término. Por ejemplo, en el último término la primer ocurrencia ligada de la variable x se representa con el número 1, mientras que la segunda ocurrencia se representa con el 2.

4.2. Representación localmente sin nombres

Un problema de la representación sin nombres es que no deja lugar para variables libres.
Una forma de manejar las variables libres es mediante un desplazamiento de índices una distancia dada por la cantidad de variables libres. De esta manera, se representan las variables libres por los índices más bajos, como si existieran lambdas invisibles alrededor del término, ligando todas las variables. Adicionalmente, se utiliza un *contexto de nombres* en el que se relacionan índices con su nombre textual [Pie02, Cap. 6].

[1] Pronunciar "De Bron" como una aproximación modesta a la pronunciación correcta.

Otra forma, que es lo que usaremos, es la representación localmente sin nombres [MM04]. En esta representación las variables libres y ligadas están en diferentes categorías sintácticas.
Utilizando esta representación, los términos quedan definidos de la siguiente manera:

```
data Term = Bound Int
          | Free Name
          | Term :@: Term
          | Lam Type Term
```

donde el tipo Name es un renombramiento de String.

Ejercicio 1. Definir en Haskell la función conversion::LamTerm → Term en el archivo Simplytyped.hs que convierte términos de λ-cálculo tipado a términos equivalentes en la representación localmente sin nombres.

4.3. Testeando la conversión.

Para facilitar el testeo de las implementaciones, ya se encuentra implementada en el intérprete la operación `:print`, que dado un término muestra el LamTerm obtenido luego del parseo, el Term obtenido luego de la conversion, y por último muestra el término en forma legible.
Por ejemplo, un uso del comando `:print` sería:

```
ST> :print \x:E .x
LamTerm AST:
LAbs "x" EmptyT (LVar "x")

Term AST:
Lam EmptyT (Bound 0)

Se muestra como:
\x:E. x
```

5. Evaluación

Nos interesa implementar un intérprete de lambda-cálculo que siga el orden de reducción *call–by–value*, dado por las reglas:

$$\frac{t_1 \to t_1'}{t_1\ t_2 \to t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \to t_2'}{v\ t_2 \to v\ t_2'} \qquad \text{(E-App2)}$$

$$(\lambda x : T_1.\ t_1)\ v \to t_1\,[x/v] \qquad \text{(E-AppAbs)}$$

La función sub (definida en Simplytyped.hs) 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. Utilizar esta función para definir el evaluador de términos en el siguiente ejercicio:

Ejercicio 2. Implementar un evaluador eval :: NameEnv Value → Term → Value, donde eval *nvs t* devuelve el valor de evaluar el término *t* en el entorno *nvs* utilizando la estrategia de reducción *call–by–value*.
El entorno *nvs* asocia a cada variable una tupla con su valor y tipo.
Una vez definida la función eval podrá testear la misma en el intérprete interactivo escribiendo el término a evaluar.

6. Inferidor de tipos

Las reglas de tipado de nuestro cálculo serán las usuales:

$$\frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}} \quad \text{(T-Var)}$$

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

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

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

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

Esta función se define en términos de la función auxiliar infer′ :: Context → NameEnv Value Type → Term → Either String Type que, además de un entorno de variables globales, toma un entorno de variables locales. Donde la inferencia de tipos de una variable localmente ligada se reduce a una indexación del entorno local: infer′ $c\ e$ (Bound i) = Just $(c!!i)$. De esta manera el entorno Γ de las reglas de tipado estaría representado por los dos primeros argumentos de infer′.

7. 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 [Hug95].
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.

8. Extensiones

En los ejercicios de esta sección se extenderá la implementación del λ-calculo simplemente tipado, con naturales, listas de naturales y la construcción `let`.

8.1. 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:

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

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

Y su regla de tipado:

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

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.

8.2. 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:

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

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

Tendremos nuevos valores, las constantes numéricas:

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

donde `nv` es:

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

Las reglas que extienden la evaluación son:

$$\mathtt{R}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{0} \to \mathtt{t}_1 \qquad \text{(E-RZero)}$$

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

$$\frac{\mathtt{t}_3 \to \mathtt{t}_3'}{\mathtt{R}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3 \to \mathtt{R}\ \mathtt{t}_1\ \mathtt{t}_2\ \mathtt{t}_3'} \qquad \text{(E-R)}$$

Las nuevas reglas de tipado son:

$$\Gamma \vdash \mathtt{0} : \mathtt{Nat} \qquad \text{(T-Zero)}$$

$$\frac{\Gamma \vdash \mathtt{t} : \mathtt{Nat}}{\Gamma \vdash \mathtt{suc}\ \mathtt{t} : \mathtt{Nat}} \qquad \text{(T-Suc)}$$

$$\frac{\Gamma \vdash t_1 : \mathsf{T} \quad \Gamma \vdash t_2 : \mathsf{T} \to \mathtt{Nat} \to \mathsf{T} \quad \Gamma \vdash t_3 : \mathtt{Nat}}{\Gamma \vdash \mathtt{R}\ t_1\ t_2\ t_3 : \mathsf{T}} \quad \text{(T-Rec)}$$

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:

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

Verificar que el intérprete la acepte y evalúe adecuadamente.

8.3. 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:

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

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

Tendremos nuevos valores:

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

donde `lv` es:

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

Las reglas que extienden la evaluación son:

$$\mathtt{RL}\ t_1\ t_2\ \mathtt{nil} \to t_1 \quad \text{(E-RNil)}$$

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

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

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

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

Las nuevas reglas de tipado son:

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

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

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

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 en un archivo `Ejemplos/Ej7.lam` la función sumPos, que dada una lista, sume a cada elemento de la misma su posición más 1. Por ejemplo, sumPos $[3, 5, 7] = [3 + 1, 5 + 2, 7 + 3]$. Verificar que el intérprete la acepte y evalúe correctamente.

Referencias

[DB72] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

[Hug95] John Hughes. The Design of a Pretty-printing Library. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, pages 53–96. Springer Verlag, LNCS 925, 1995.

[MM04] Conor McBride and James McKinna. Functional pearl: I am not a number–I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 1–9. ACM, 2004.

[Pie02] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.