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

1. Introducción

El objetivo del siguiente trabajo es implementar en Haskell un evaluador de términos de lambda cálculo y probarlo implementando un algoritmo. El trabajo se debe realizar individualmente y se debe entregar un informe en papel con los ejercicios resueltos y el código fuente del programa, y enviar el código fuente por correo electrónico a `entregas.alp@gmail.com`, antes del fin del 2 de Octubre.

2. Representación de Lambda Términos

El conjunto Λ de términos del λ-cálculo se define inductivamente con las siguientes reglas:

$$\frac{x \in X}{x \in \Lambda} \qquad \frac{t \in \Lambda \quad u \in \Lambda}{(t\ u) \in \Lambda} \qquad \frac{x \in X \quad t \in \Lambda}{(\lambda x.t) \in \Lambda}$$

donde X es un conjunto infinito numerable de identificadores.
La representación mas obvia de los términos lambda en Haskell consiste en tomar como conjunto de variables las cadenas de texto, de la siguiente manera:

```
data LamTerm = LVar String
             | App LamTerm LamTerm
             | Abs String LamTerm
```

Ejercicio 1. Definir una función $num :: Integer \to LamTerm$ de Haskell que dado un entero devuelve la expresión en λ-cálculo de su numeral de Church.

$$\underline{0} = \lambda s\ z.\ z \qquad \underline{1} = \lambda s\ z.\ s\ z \qquad \underline{2} = \lambda s\ z.\ s\ (s\ z) \qquad \underline{3} = \lambda s\ z.\ s\ (s\ (s\ z)) \qquad \underline{4} = \lambda s\ z.\ s\ (s\ (s\ (s\ z))) \qquad \dots$$

Normalmente se usan ciertas convenciones para escribir los λ-términos. Por ejemplo se supone que la aplicación asocia a la izquierda (podemos escribir $M\ N\ P$ en lugar de $((M\ N)\ P)$), las abstracciones tienen el alcance más grande posible (podemos escribir $\lambda x.P\ Q$ en lugar de $(\lambda x.P\ Q)$, y podemos juntar varias abstracciones consecutivas bajo un mismo λ (podemos escribir $\lambda x_1\, x_2 \dots x_n.M$ en lugar de $(\lambda x_1.(\lambda x_2.(\dots(\lambda x_n.M)\dots))$. Estas convenciones definen una *gramática extendida* del λ-cálculo.

Ejercicio 2. Se desea escribir un parser para lambda expresiones que acepte términos de la gramática extendida del λ-cálculo. Para ello:

1. Definir la gramática extendida del λ-cálculo.
2. Dar una gramática equivalente a la del ítem anterior que evite la recursión a izquierda.
3. Implementar el parser $parseLamTerm :: Parser\ LamTerm$ usando la biblioteca `Parsec`. Utilizar la función del ejercicio 1 para interpretar números como numerales de Church.

`Parsec` es una biblioteca para construir parsers con combinadores similares a los vistos en clase, pero mucho más potente (`http://www.haskell.org/haskellwiki/Parsec`). La biblioteca está incluida en la *Haskell Platform*.
Además de permitir trabajar con combinadores a nivel de caracter, `Parsec` permite trabajar con *tokens*. Es decir que el parseo se hace en dos pasos:

1. Se transforma la cadena de entrada en una lista de tokens. Cada token indica si se tiene un identificador, una palabra clave, un operador, etc. Durante esta transformación se eliminan espacios y comentarios.

   Se utiliza la función *makeTokenParser* para generar parsers que funcionen sobre tokens. Para ello, se especifica la forma de los comentarios, identificadores, etc. La biblioteca provee algúnos definiciones de lenguaje que se pueden modificar. En particular, en `Untyped.hs`, hemos tomado como base la definición para Haskell provista por la biblioteca (*haskellStyle*) y hemos modificado el formato de los identificadores y las palabras clave.

```
untyped :: TokenParser u
untyped = makeTokenParser (haskellStyle { identStart = letter <|> P.char '_',
                                          reservedNames = ["def"] })
```

El uso de un parser de tokens hace que no sea necesario lidiar con espacios en blanco o comentarios (usando el parser de tokens *untyped* el código fuente puede usar comentarios como en Haskell sin esfuerzo adicional.)

2. Se utilizan combinadores que trabajan sobre tokens. Por ejemplo, el parser *reservedOp untyped* `"."`, parsea el operador ".", el parser *identifier untyped*, parsea un identificador (según las reglas de la definición de *untyped*), y el parser *parens untyped p* parsea lo mismo que p, pero entre paréntesis. Se recomienda no mezclar los operadores que trabajan a bajo nivel con los operadores que trabajan sobre tokens ya que pueden surgir problemas, por ejemplo, con el manejo de los espacios en blanco.

Muchos combinadores son similares a los de la biblioteca simple vista en clase, por ejemplo *many*, *many1*, y <|>.

2.1. 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 escritos con esta notación:

$$\lambda x.\, x \;\mapsto\; \lambda 0 \qquad\qquad \lambda x \lambda y \lambda z.\, x \;\mapsto\; \lambda\lambda\lambda 2$$
$$\lambda y.\, (\lambda x\ y.x)\ y \;\mapsto\; \lambda(\lambda\lambda 1)\ 0 \qquad\qquad \lambda x.\, \lambda y.\, x\ (\lambda y.\, y\ x) \;\mapsto\; \lambda\lambda 1\ (\lambda 0\ 2)$$

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.

2.2. 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].
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.

```
data Term = Bound Int
          | Free Name
```

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

$$\frac{na_1 \to t_1'}{na_1\ t_2 \to t_1'\ t_2} \quad \text{(E-App1)}$$

$$\frac{t_2 \to t_2'}{neu_1\ t_2 \to neu_1\ t_2'} \quad \text{(E-App2)}$$

$$\frac{t_1 \to t_1'}{\lambda x.\, t_1 \to \lambda x.\, t_1'} \quad \text{(E-Abs)}$$

$$(\lambda x.\, t_1)\ t_2 \to t_1[t_2/x] \quad \text{(E-AppAbs)}$$

$$nf ::= \lambda x.\, nf \mid neu$$
$$neu ::= x \mid neu\ nf$$
$$na ::= x \mid t_1\ t_2$$

Figura 1: Semántica operacional del orden de reducción normal

| *Term* :@: *Term*
| *Lam Term*

Una variable libre es un nombre global:

data *Name* = *Global String*

Por ejemplo el término $\lambda x\ y.\, z\ x$, está dado por *Lam* (*Lam* (*Free* (*Global* `"z"`) :@: *Bound* 1)).
En esta representación, generalmente se agrega otro constructor (digamos *Local Int*) al tipo *Name*, para poder ver una variable ligada como libre cuando se analiza localmente un subtérmino. Por ejemplo, considere el término $M = \lambda y.\, x\ z$. La variable x está ligada en $\lambda x.\, M$, pero localmente libre en M. Sin embargo, nosotros no lo necesitaremos en este TP, ya que no necesitaremos "meternos" adentro de una abstracción.

Ejercicio 3. Definir en Haskell la función *conversion* :: *LamTerm* → *Term* que convierte términos de λ-cálculo a términos equivalentes en la representación localmente sin nombres.

2.3. Testeando el parser y 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 el último muestra el término en forma legible. Por ejemplo, un uso del comando `:print` sería:

```
UT> :p \x y. (z x) y
LamTerm AST:
Abs "x" (Abs "y" (App (App (LVar "z") (LVar "x")) (LVar "y")))

Term AST:
Lam (Lam ((Free (Global "z") :@: Bound 1) :@: Bound 0))

Se muestra como:
\x y. z x y
```

3. Evaluación

Nos interesa implementar un intérprete de lambda-cálculo que siga el orden de reducción normal (Fig. 1).
Al implementar un evaluador de lambda-cálculo parecería que uno debe necesariamente meterse en el pantanoso terreno de la substitución. Una de las medidas que tomamos para simplificar la implementación de la substitución es usar índices de De Bruijn para variables ligadas. La otra medida que tomaremos es usar el espacio de funciones de Haskell para representar los valores que sean abstracciones.

$$\begin{aligned}\textbf{data}\ \mathit{Value} &= \mathit{VLam} \quad (\mathit{Value} \to \mathit{Value}) \\ &\mid \mathit{VNeutral}\ \mathit{Neutral} \\ \textbf{data}\ \mathit{Neutral} &= \mathit{NFree}\ \mathit{Name} \\ &\mid \mathit{NApp}\ \mathit{Neutral}\ \mathit{Value}\end{aligned}$$

Ejercicio 4. Implementar la aplicación de valores $vapp :: Value \to Value \to Value$.

Ejercicio 5. Implementar un evaluador $eval :: [(Name, Value)] \to Term \to Value$, donde $eval\ t\ nvs$ devuelve el valor de evaluar el término t en el entorno nvs utilizando la estrategia de reducción normal. El entorno nvs asocia cada variable global a su valor. Por ejemplo, si ya definimos la función identidad con nombre `"id"`, entonces un entorno va a contener el par ($Global$ `"id"`, $VLam\ id$).

4. Mostrando Valores

Un problema de la representación de valores como funciones de Haskell es que a las funciones de Haskell no podemos examinarlas (por ejemplo para imprimirlas), sólo podemos aplicarlas.
Una manera de lograr mostrar los valores es convertirlos nuevamente en términos. Para esto, en el caso de una función, la podemos aplicar a una variable fresca, y examinar el resultado. Notar que el resultado de aplicar una función $f :: Value \to Value$ puede ser nuevamente una función que debe ser a su vez convertida, por lo que necesitamos una nueva variable fresca. Para poder obtener variables frescas fácilmente extendemos el tipo de datos de los nombres con variables de tipo Int:

$$\begin{aligned}\textbf{data}\ \mathit{Name} &= \mathit{Global}\ \mathit{String} \\ &\mid \mathit{Quote}\ \mathit{Int}\end{aligned}$$

De esta manera se puede llevar un contador i con las variables usadas hasta el momento, y crear una variable fresca simplemente con $Quote\ i$. Por supuesto luego habrá que incrementar el contador.
Si el resultado de la aplicación de la función a la variable fresca es una variable $Quote\ k$ entonces se deberá reemplazar por una variable ligada $Bound\ (i - k - 1)$, donde i es la cantidad de variables que se crearon.

Ejercicio 6. Escribir la función $quote :: Value \to Term$.

5. Programando en λ-cálculo

En los archivos del trabajo práctico se incluye un archivo `Prelude.lam`, que contiene algunas definiciones básicas y que es cargado automáticamente por el intérprete. Otros archivos con más definiciones pueden ser cargados pasando el nombre de archivo en la línea de comandos, o simplemente usando el comando `:load` del intérprete.

Ejercicio 7. Escribir en un archivo `Sqrt.lam` una implementación en lambda-cálculo de un algoritmo que implemente la raíz cuadrada entera para numerales de Church.

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.