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 en grupos de dos personas y la fecha límite de entrega es el Jueves 26 de septiembre, en donde se debe entregar:

- en papel, un informe con los ejercicios resueltos incluyendo todo el código que haya escrito;
- en forma electrónica (en un archivo comprimido) el código fuente de los programas, usando el sitio de la materia del campus virtual de la UNR (`http://comunidades.campusvirtualunr.edu.ar`).

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 más 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* → *LamTerm* de Haskell en el archivo `Parser.hs` que dado un entero devuelve la expresión en λ-cálculo de su numeral de Church.

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

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 $\lambda x_1\, x_2 \ldots x_n.M$ en lugar de $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.M)\ldots))$. Estas convenciones definen una *gramática extendida* del λ-cálculo. Para simplificar el trabajo se brinda un analizador sintáctico que acepta términos de la gramática extendida del λ-cálculo y utiliza la función del ejercicio 1 para interpretar números como numerales de Church.

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 *niveles 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 niveles de De Bruijn. Cada número representa la ocurrencia de una variable en un término. y denota la posición ocupa su "binder"desde afuera hacia adentro de la abstracción.

Los siguientes son algunos ejemplos de lambda términos escritos con esta notación:

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

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.
Utilizando esta representación, los términos quedan definidos de la siguiente manera:

data *Term* = *Bound Int*
| *Free Name*
| *Term* :@: *Term*
| *Lam Term*

Una variable libre es un nombre global:

type *Name* = *String*

Por ejemplo el término $\lambda x\ y.\, z\ x$, está dado por *Lam* (*Lam* (*Free* `"z"` :@: *Bound* 0)).

Ejercicio 2. 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 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 "z" :@: Bound 0) :@: Bound 1))

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).

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

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

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

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

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

$$\begin{array}{lll} nf & ::= \lambda x.\, nf \mid neu \\ neu & ::= x \mid neu\ nf \\ na & ::= x \mid t_1\ t_2 \end{array}$$

Figura 1: Semántica operacional del orden de reducción normal

Para implementarlo se necesita una operación que represente una β-reducción sobre términos en la representación sin nombres. Dado que una β-reducción disminuye en uno la cantidad de "binders" y puede requerir sustituir variables dentro de abstracciones, los números que representan variables ligadas en un término pueden cambiar luego de la aplicación de esta operación. Por ejemplo, la expresión $\lambda x\, y.\, (\lambda z.\, y\, (\lambda k.\, z)\, x)\, (\lambda w.\, w)$ queda representada en niveles de De Bruijn como:

$$\lambda\lambda(\lambda\, 1\, (\lambda 2)\, 0)\, (\lambda\, 2)$$

La expresión contiene el β-redex $(\lambda\, 1\, (\lambda 2)\, 0)\, (\lambda\, 2)$ de nivel 2 (o sea, bajo dos λ) que para β-contraerse debe sustituir la variable 2 por $(\lambda\, 2)$. Sin embargo, la solución no es

$$\lambda\lambda\, 1\, (\lambda\lambda 2)\, 0$$

que corresponde a $\lambda x\, y.\, y\, (\lambda k\, w.\, k)\, x)$, sino que es

$$\lambda\lambda\, 1\, (\lambda\lambda 3)\, 0$$

que corresponde a $\lambda x\, y.\, y\, (\lambda k\, w.\, w)\, x)$. Por lo tanto, vemos que al hacer una substitución puede ser necesario adaptar algunos índices.

Se verán primero dos operaciones necesarias para definir un intérprete sobre términos sin nombres. La primera de ellas, llamada *shifting* es usada para modificar los valores correspondientes a variables que aparecen ligadas en un término (en el ejemplo anterior, el número 2 del término debe aumentar en 1 luego de la aplicación de la sustitución) cuando se sustituyen dentro de un termino más profundo del que estaba. La operación de *shifting*, la cual escribiremos como $\uparrow_c^d$, es la encargada de tomar un termino y sumarle d unidades, a cada variable ligada con un índice mayor o igual a c. A continuación se muestra una definición para esta operación:

$$\begin{array}{rcl} \uparrow_c^d (k) & = & \begin{cases} k & \text{si } k < c \\ k + d & \text{si } k \geqslant c \end{cases} \\ \uparrow_c^d (\lambda\, t) & = & \lambda\ \uparrow_c^d (t) \\ \uparrow_c^d (t\ t') & = & \uparrow_c^d (t)\ \uparrow_c^d (t') \end{array}$$

Ejercicio 3. Definir una función *shift* :: *Term* → *Int* → *Int* → *Term* que implemente el operador $\uparrow_c^d$.

Ahora, podemos definir la sustitución de variables sobre términos sin nombres. Se denotará con $t[t'/i]_j$ a la sustitución de la variable representada con el número i por t' en t en un nivel j-veces más alto. Sea $t = \lambda\, t_1$, al reemplazar el término t' por i en t_1 debemos asegurarnos de que las variables en t' mantengan su referencia a los mismos nombres al que hacían referencia antes de la sustitución. La definición de esta sustitución es la siguiente:

$$\begin{array}{rcl} k[t'/i]_j & = & \begin{cases} \uparrow_i^j (t') & \text{si } k = i \\ k - 1 & \text{si } k > i \\ k & \text{si } k < i \end{cases} \\ (\lambda\, t_1)[t'/i]_j & = & \lambda\, (t_1[t'/i]_{j+1}) \\ (t_1\, t_2)[t'/i]_j & = & (t_1[t'/i]_j)\, (t_2[t'/i]_j) \end{array}$$

Ejercicio 4. Definir una función $subst :: Term \to Term \to Int \to Term$, tal que $subst\ t\ t'\ i$ sea el resultado de $t[t'/i]_0$.
Para implementar el intérprete de términos se definirá la β-reducción sobre términos sin nombres usando la sustitución definida anteriormente *subst* y la operación *shifting*. Si se tiene un β-redex de la forma $(\lambda\, t)\, v$, luego el resultado de la reducción sería el siguiente término:

$$(t[v/j]_0)$$

Ejercicio 5. Implementar un evaluador $eval :: [(Name, Value)] \to Term \to Term$, 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 (*Free* `"id"`, *Lam* (*Bound* 0)). Tener en cuenta que todas la variables globales se refieren a términos de nivel 0.

4. 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 6. 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.

[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.