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 Miércoles 28 de octubre, en donde se debe entregar (en un archivo comprimido) los contenidos de los directorios `Ejemplos` y `src`, 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 *índices de De Bruijn*[1], también llamada *representación sin nombres* [**?**]. En esta notación los nombres de las

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

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:

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

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 [**?**, Cap. 6].
Otra forma, que es lo que usaremos, es la representación localmente sin nombres [**?**]. 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, por lo tanto podríamos definir el tipo *Name* como un renombramiento de *String*, sin embargo más adelante vamos a necesitar definir variables como números, por lo tanto usaremos el siguiente tipo de datos para *Name*:

data *Name* = *Global String*
| *Quote Int*

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 para este TP, ya que no será necesario meterse dentro de una abstracción.

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. Para esto, se imprime el término usando colores para asociar cada variable con la ligadura correspondiente. Adicionalmente, también se muestra el término asignandole a cada variable un nombre arbitrario.
Por ejemplo, un uso del comando `:print` sería:

```
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))
```

$$\frac{app_1 \to t_1'}{app_1\ t_2 \to t_1'\ t_2} \quad \text{(E-App1)}$$

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

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

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

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

Figura 1: Semántica operacional del orden de reducción normal

```
Se muestra como:
λ λ z #1 #0

Se muestra con nombres arbitrarios como:
λ 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.

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

Ejercicio 3. Implementar la aplicación de valores $vapp :: Value \to Value \to Value$.

Ejercicio 4. Implementar un evaluador $eval :: NameEnv\ Value \to Term \to Value$, 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 $(Global\ \texttt{"id"}, VLam\ id)$.
Para ello se puede hacer uso de la función auxiliar $eval' :: Term \to (NameEnv\ Value, [Value]) \to Value$ que, además de un entorno de variables globales, toma un entorno de variables locales. De esta manera, la evaluación de una variable localmente ligada se reduce a una indexación del entorno local: $eval'\ (Bound\ ii)\ (\_, lEnv) = lEnv\ !!\ ii$.

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 usamos como nombre de una variable un entero.

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.
En el resultado de la aplicación de la función se deberá reemplazar cada variable fresca *Quote* k por una variable ligada *Bound* $(i - k - 1)$, donde i es la cantidad de variables que se crearon.

Ejercicio 5. Escribir la función $quote :: Value \to Term$.

5. Programando en λ-cálculo

En los archivos del trabajo práctico se incluye un archivo `Ejemplos/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 `Ejemplos/Map.lam` una implementación en lambda-cálculo de un algoritmo que implemente la función mapN que dada una lista y una función f, aplique f a cada elemento de la lista tantas veces como el índice de su posición en la lista, es decir:

$$\mathsf{mapN}\ f\ [x_1, .., x_n] = [x_1, f\ x_2, .., f^{n-1}\ x_n]$$

Un algoritmo para definir mapN consiste en aplicar una función sobre tuplas n veces, generando en cada aplicación las siguientes tuplas:

$([\,], [x_1, .., x_n])$
$([x_1], [x_2, .., x_n])$
$([x_1, f\ x_2], [x_3, .., x_n])$
...
$([x_1, f\ x_2, .., f^{n-1}\ x_n], [\,])$

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.