XIV Conferencia Latinoamericana de Informática
17avas. Jornadas Argentinas de Informática
e Investigación Operativa
Buenos Aires, Setiembre 1988.

Implementación de una Máquina de Multi-Combinadores Categóricos

Martin A.Musicante* & Rafael D.Lins†
Dept. Informática - Univ. Federal de Pernambuco - Recife - Brasil
ESLAI - Parque Pereyra Iraola - Berasategui - Bs.As. - Argentina.

Resumen

Los Multi-Combinadores Categóricos forman un sistema de reescritura desarrollado para proveer implementaciones eficientes de lenguajes funcionales con evaluación lazy. El núcleo del sistema contiene sólo dos reglas de reescritura, con muy baja complejidad de reconocimiento. El sistema realiza el equivalente a varias β-reducciones en un solo paso, evitando generar expresiones trivialmente reducibles.

Se presenta aqui la implementación de una máquina de reducción de grafos basada en este formalismo. Se dan también medidas de tiempo y espacio consumidos por la máquina para algunos casos de prueba, y se realizan comparaciones con otras implementaciones existentes.

El presente trabajo forma parte de la implementación de un intérprete de SASL usando Multi-Combinadores Categóricos.

Palabras clave: Multi-Combinadores Categóricos, lambda cálculo, programación funcional, evaluación lazy, reducción de grafos.

Abstract

Categorical Multi-Combinators form a rewriting system developed with the aim of providing efficient implementation of lazy functional languages. The core of the system consists of only two rewriting laws with a very low pattern-matching complexity. This system allows the equivalent of several β-reductions to be performed at once, and avoids the generation of trivially reducible sub-expressions.

Herein we present implementation details of a Categorical Multi-Combinator graph reduction machine. Performance figures of time complexity are also presented. These figures are compared with implementations using a different formalism.

This work is part of an implementation for SASL using Categorical Multi-Combinators.

Keywords: Categorical Multi-Combinators, lambda calculus, functional programming, lazy evaluation, graph reduction.

*ESLAI - CC 3193 - cp 1000 - Bs.As. - Argentina
†Dept.de Informática - U.F.PE. - 50.739 - Recife - PE - Brasil

Introducción

Los lenguajes de la programación funcional proveen una forma clara y fácil de especificación de soluciones a problemas computacionales. Ellos están siendo foco de intensa atención [15], y su aceptación, en general era postergada a causa de la ineficiencia de las implementaciones disponibles.

Esta situación esta siendo revertida. Existen hoy implementaciones de este tipo de lenguajes que compiten con los compiladores de los lenguajes tradicionales [6].

Podemos ver a un programa funcional como un conjunto de definiciones de funciones y otros objetos. La ejecución de un programa funcional consiste en la evaluación de una expresión; esto puede hacerse reescribiendo sucesivamente la expresión hasta que alcance una forma "imprimible" (o forma normal). Diremos que una expresión está en forma normal cuando ella no puede reescribirse con las reglas del sistema. Por ejemplo, si tenemos

```
fac n = n * fac(n-1)   , n>0          (*)
      = 1              , otherwise
```

entonces, la expresión

```
fac 2
```

será evaluada de la siguiente forma:

$$
\begin{aligned}
\texttt{fac2} &\rightarrow 2 * \mathit{fac}\ (2-1) \qquad \dagger \\
&\rightarrow 2 * \mathit{fac}\ 1 \\
&\rightarrow 2 * (1 * \mathit{fac}\ (1-1)) \\
&\rightarrow 2 * (1 * \mathit{fac}\ 0) \\
&\rightarrow 2 * (1 * 1) \\
&\rightarrow 2 * 1 \\
&\rightarrow 2
\end{aligned}
$$

La definición (*) se usó en la linea (†), substituyendo el valor 2 en la variable *n*. Este proceso de pasaje de parámetros es la mayor fuente de "overhead" en las implementaciones de lenguajes funcionales como sistemas de reescritura. En el método de compilación elaborado primeramente por Turner en [18] se evita este problema al substituir las variables del programa por una combinación aplicativa de funciones constantes llamadas

combinadores (En general, un combinador será una expresión funcional sin variables libres).

Turner usó un sistema de combinadores basados en la Lógica Combinatoria de Curry [1]. Puede obtenerse otra teoría formal de funciones basandose en la Teoría de Categorías. Este es el caso de los **Combinadores Categóricos**, que forman un sistema similar al de la lógica combinatoria. El sistema original es debido a Curien [4] y está inspirado en la equivalencia entre el λ-cálculo con tipos y las Categorías Cartesianas Cerradas (ésto último puede verse en Lambek [7] y Scott [16]).

En [3] se da una máquina de pila para la ejecución de Combinadores Categóricos; pero, como lo muestra Turner en [18], este tipo de máquinas resulta sumamente ineficiente para la implementación de lenguajes funcionales lazy.

El sistema llamado **Combinadores Categóricos Simplificados** fue desarrollado por Lins [8] para proveer implementaciones eficientes de lenguajes funcionales lazy. Es un sistema basado en el original de Curien, pero con las ventajas de obtener una relación lineal entre el tamaño de una λ-expresión y su equivalente en el sistema (en el original, esta relación es cuadrática), y de poseer pocas y muy simples reglas de reescritura de términos. Existen simulaciones [9] que muestran que una implementación basada en este sistema puede ser al menos, un orden de magnitud mas rápida que una basada en el original, y presenta complejidad en espacio y tiempo del mismo orden que los combinadores de Turner en [9].

En [9 y 10] son introducidas dos nuevas modificaciones al sistema, formando los llamados **Combinadores Categóricos Lineales**. Estas modificaciones reducen el conjunto de reglas de reescritura, incrementando la eficiencia en la evaluación al disminuir también el número de reescrituras necesarias para llevar un término a su forma normal. Estas reglas no alteran la complejidad del algoritmo de "pattern matching".

Multi-Combinadores Categóricos son una generalización de los Combinadores Categóricos Lineales. El núcleo del sistema consiste sólo de dos reglas de reescritura, con una muy baja complejidad de reconocimiento, y evitando la generación de sub-expresiones trivialmente reducibles.

Existen otros trabajos en el mismo sentido. Un ejemplo de ello son los **Supercombinadores** de Hughes [5]. En [11] se muestran las similitudes y diferencias entre estas máquinas.

Se presenta aquí la especificación e implementación de una máquina de Multi-Combinadores Categóricos, haciendo un muy sencillo estudio comparativo con otras implementaciones.

1 Multi-Combinadores Categóricos

Multi-Combinadores Categóricos forman un sistema de combinadores (formas funcionales sin variables libres) basados en la teoría de categorías.

Multi-Combinadores Categóricos utilizan la notación del λ-cálculo de DeBruijn para variables. En dicho λ-cálculo, cada variable está representada

por un número natural, que es la "cantidad de lambdas" entre dica variable y el lambda que la liga. Así, por ejemplo, el λ-término

$$\lambda x \lambda y \lambda z.xz$$

se traduce en el λ-cálculo de DeBruijn como

$$\lambda\lambda\lambda.20$$

Una forma de presentar el sistema, es dando su traducción desde el λ-cálculo. En Multi-Combinadores Categóricos, la aplicación de funciones se denota por yuxtaposición - tomándose esta operación como asociativa a izquierda.

El algoritmo de compilación [10] para traducir λ-expresiones hacia Multi-Combinadores Categóricos es el siguiente:

(T .1) $[\underbrace{\lambda x_i \dots \lambda x_j}_{n}.a] = L^{n-1}(R^{x_i,\dots,x_j}_{n-1,\dots,0}a)$

(T .2) $[a \dots b] = [a] \dots [b]$

(T .3) $[c] = c$, donde c es una constante

Donde $R^{x_i \dots x_j}_{n_i \dots n_j}$ es la función de *substitución* que reemplaza recursivamente las ocurrencias de cada variable x_i por el entero que la representará en el sistema. Esta función se define como sigue:

(T .4) $R^{x_i \dots x_j}_{n_i \dots n_j} \underbrace{\lambda x_k \dots \lambda x_l}_{m}.a = L^{m-1}(R^{x_i,\dots,x_j,x_k,\dots,x_l}_{n_i+m,\dots,n_j+m,m-1,\dots,0}a)$

(T .5) $R^{x_i \dots x_j}_{n_i \dots n_j}(a \dots b) = (R^{x_i \dots x_j}_{n_i \dots n_j}a) \dots (R^{x_i \dots x_j}_{n_i \dots n_j}b)$

(T .6) $R^{x_i \dots x_j}_{n_i \dots n_j}b = \begin{cases} b \text{ , si } b \text{ es una constante} \\ n_k \text{ , si } b = x_k \end{cases}$

Si al aplicar la regla T.6 a una variable b, ésta puede asociarse a más de un x_k, entonces se elije el mínimo n_k correspondiente para hacer la substitución. Esto tiene por objeto preservar la localidad en el "binding" de las variables (notemos que, en realidad, la función mantiene una pila, y a cada variable it b se le asocia la mínima distancia al tope).

Damos a continuación un ejemplo de traducción de una λ-expresión usando el algoritmo anterior:

$$\begin{aligned}
[\lambda x \lambda y (y \lambda z (zz))] &\overset{T.1}{=} (L^1(R^{x,y}_{1,0}y\ \lambda z(zz))) \\
&\overset{T.5}{=} (L^1((R^{x,y}_{1,0}y)\ (R^{x,y}_{1,0}\lambda z(zz)))) \\
&\overset{T.6}{=} (L^1(0\ (R^{x,y}_{1,0}\lambda z(zz))) \\
&\overset{T.4}{=} (L^1(0\ L^0(R^{x,y,z}_{2,1,0}(zz)))) \\
&\overset{T.5}{=} (L^1(0\ L^0((R^{x,y,z}_{2,1,0}z)\ (R^{x,y,z}_{2,1,0}z))) \\
&\overset{T.6}{=} (L^1(0\ L^0(0\ (R^{x,y,z}_{2,1,0}z))) \\
&\overset{T.6}{=} (L^1(0\ L^0(0\ 0)))
\end{aligned}$$

Con esto queda definida la sintaxis del núcleo del sistema. Ahora bien, es necesario proveerlo (con fines practicos) de estructuras de datos. Para ello, en [11] se propone la adición de dos nuevos combinadores: el combinador de Bloques de Datos (**D**) y los selectores (S^i); lo que implica la adición de una nueva regla de reescritura al núcleo del sistema, como se verá mas adelante.

Luego, deben agregarse las siguientes reglas para compilar listas y operadores sobre ellas

(1.1) $[\![[\,]]\!] \Rightarrow [\,]$

(1.2) $[\![Hd]\!] \Rightarrow S^0$

(1.3) $[\![Tl]\!] \Rightarrow S^1$

(1.4) $[\![a : b]\!] \Rightarrow D\ [\![a]\!]\ [\![b]\!]$

(1.5) $R^{x_i \ldots x_j}_{n_i \ldots n_j}[\,] \Rightarrow [\,]$

(1.6) $R^{x_i \ldots x_j}_{n_i \ldots n_j} Hd \Rightarrow S^0$

(1.7) $R^{x_i \ldots x_j}_{n_i \ldots n_j} Tl \Rightarrow S^1$

(1.8) $R^{x_i \ldots x_j}_{n_i \ldots n_j}(a : b) \Rightarrow D\ (R^{x_i \ldots x_j}_{n_i \ldots n_j} a)(R^{x_i \ldots x_j}_{n_i \ldots n_j} b)$

donde ":" denota el operador de construcción de listas y $[\,]$ es la lista vacía. La compilación de operadores sobre listas se realiza reemplazando estos operadores por el codigo compilado de sus definiciones. Por ejemplo, para la función ++ de SASL, que concatena dos listas se tiene:

(1.9) $[\![a \ ++ \ b]\!] \Rightarrow concat\ a\ b$, *donde*

$$concat \equiv L^1((Cond\ ([\,] = 1)\ 0)\ (D\ (S^0 1)\ (concat\ (S^1 1)\ 0)))$$

Se da ahora un ejemplo de compilación de una función usando listas. Definimos una función que cambia el orden de los elementos de una lista.

```
ref n = if n=[ ] then [ ]
                 else (ref (Tl n)) : (Hd n);
```

O con la sintaxis de SASL

```
ref n = (n=[ ]) -> [ ]; (ref (Tl n)) : (Hd n)
```

Esta función se traduce a Multi-Combinadores Categoricos como sigue:

$$\begin{aligned}
& [\![ref\ n = (n = [\,]) \rightarrow [\,];\ (ref\ (Tl\ n)) : (Hd\ n)]\!] \\
\Rightarrow\ & ref \equiv L^0(R^n_0((n = [\,]) \rightarrow [\,];\ (ref\ (Tl\ n)) : (Hd\ n))) \\
\Rightarrow\ & ref \equiv L^0(Cond\ (R^n_0(n = [\,]))\ (R^n_0[\,])\ (R^n_0(ref\ (Tl\ n)) : (Hd\ n))) \\
\Rightarrow\ & ref \equiv L^0(Cond\ (= (R^n_0 n)\ (R^n_0[\,]))\ (R^n_0[\,])\ (R^n_0(ref\ (Tl\ n)) : (Hd\ n))) \\
\Rightarrow\ & ref \equiv L^0(Cond\ (= 0\ (R^n_0[\,]))\ (R^n_0[\,])\ (R^n_0(ref\ (Tl\ n)) : (Hd\ n)))
\end{aligned}$$

$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ (R_0^n[\,])\ (R_0^n(ref\ (Tl\ n)) : (Hd\ n)))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (R_0^n(ref\ (Tl\ n)) : (Hd\ n)))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (R_0^n(ref(Tl\ n)))(Hd\ n))))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ ((R_0^n ref)\ (R_0^n((Tl\ n)))(R_0^n(Hd\ n))))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (R_0^n(Tl\ n)))\ (R_0^n(Hd\ n))))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ ((R_0^n Tl)\ (R_0^n n)))\ (R_0^n(Hd\ n))))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ (R_0^n n)))\ (R_0^n(Hd\ n))))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ 0))\ (R_0^n(Hd\ n))))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ 0))\ ((R_0^n Hd)\ (R_0^n n))))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ 0))\ (S^0\ (R_0^n n))))$$
$$\Rightarrow \quad ref \equiv L^0(Cond\ (=0\ [\,])\ [\,]\ (D\ (ref\ (S^1\ 0))\ (S^0\ 0))))$$

El conjunto de reglas de reescritura del sistema (enriquecido con bloques de datos y operaciones aritméticas es el siguiente (para una explicación más detallada, el lector puede recurrir a [11]).

(M.1) $\circ(x_c x_0 x_1 x_2 \ldots x_n)\ (P\ y_m \ldots y_1 y_0) \Rightarrow (x_c x'_0 x'_1 \ldots x'_n)$,

$$donde \begin{cases} x'_i &= x_i\ if\ x_i\ es\ una\ constante\ o\ de\ tipo\ L^a(b) \\ &= y_k\ if\ x_i\ es\ la\ variable\ k \\ &= \circ x_i\ (P\ y_m \ldots y_1 y_0),\ en\ otro\ caso \end{cases}$$

(M.2) $L^n(y)\ x_0 x_1 \cdots x_n x_{n+1} \cdots x_z \begin{cases} \Rightarrow y\ x_{n+1} \ldots x_z\ si\ y\ es\ una\ constante \\ \Rightarrow x_{n-y} x_{n+1} \ldots x_z\ si\ y\ is\ una\ variable \\ \Rightarrow (\circ y\ (P\ x_0 \cdots x_n)) x_{n+1} \cdots x_z,\ en\ otro\ caso \end{cases}$

(S) $S^n(D\ x_i \ldots x_1 x_0) \Rightarrow x_n$

(+) $+xy \Rightarrow x + y$

(=) $Cond\ x\ m\ n \begin{cases} \Rightarrow m,\ si\ x = True \\ \Rightarrow n,\ en\ otro\ caso \end{cases}$

Puede verse ahora un ejemplo de ejecución de un "programa" usando Multi-Combinadores Categóricos. La λ-expresión

$$(\lambda x \lambda y.y(\lambda z.z\ x)\ x)\ \mathbf{2}\ +$$

se traduce para Categorical Multi-Combinators usando el algoritmo de compilación como

$$L^1(0(L^0(0)1)1)\ \mathbf{2}\ +$$

y usando las leyes dadas más arriba, puede reescribirse sucesivamente

$$\overset{M.2}{\Rightarrow} \quad \circ(0(L^0(0)1)1)\ (P\ \mathbf{2}\ +)$$

$$
\begin{aligned}
&\overset{M.1}{\Rightarrow} \; + \; (\circ\,(L^0(0)1)\;(P\;2\;+))\;2 \\
&\overset{M.1}{\Rightarrow} \; + \; (L^0(0)\;2)\;2 \\
&\overset{M.2}{\Rightarrow} \; + \; 2\;2 \\
&\overset{+}{\Rightarrow} \; 4
\end{aligned}
$$

Como se observa, en la secuencia de reducciones, el código de Multi-Combinadores sufre una "metamorfosis". La aplicación de la regla (M.2) cambia la estructura del código y genera un "ambiente", en el cual, a cada variable se asocia un valor. Este "ambiente de evaluación" se distribuye en el cuerpo de la Multi- abstracción (L^n) por medio de sucesivas aplicaciones de la regla (M.1).

2 La Máquina de Reducción de Grafos

La idea de este tipo de máquinas es presentada por Turner en [18]. Se basa en la construcción (y transformación) de un grafo, cuyos nodos son celdas donde se almacena información, y los arcos están representados por punteros a dichas celdas.

Por razones de tiempo y objetivos del trabajo, no utilizamos un método formal de especificación de la máquina de Multi-Combinadores Categóricos. La especificación en el método VDM puede verse en [14]

Antes de considerar detalles de implementación damos una explicación, con una estructura funcional, de la máquina. La misma tiene por objeto fijar el comportamiento y mostrar la "filosofía" de la solución adoptada.

Esta "especificación funcional" se usa como paso intermedio en la construcción de la implementación final.

Debe tenerse en cuenta que serán necesarios ajustes de implementación.

2.1 Nuevos Combinadores

Observando la regla (M.2) abajo,

$$
\textbf{(M.2)}\;\; L^n(y)\;x_0x_1\cdots x_nx_{n+1}\cdots x_z \begin{cases} \Rightarrow y\;x_{n+1}\ldots x_z \;\; si\;\; y\;\; es\;\; una\;\; constante \\ \Rightarrow x_{n-y}x_{n+1}\ldots x_z \;\; si\;\; y\;\; es\;\; una\;\; variable \\ \Rightarrow (\circ y\;(P\;x_0\cdots x_n))x_{n+1}\cdots x_z, en\;\; otro\;\; caso \end{cases}
$$

Las funciones constantes y variables pueden ser identificadas en tiempo de compilación; posibilitando darles un tratamiento más sencillo que el de las funciones más complejas.

En el caso de las funciones constantes, tenemos que la regla anterior expresa:

$$
L^n(c)\;x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow c\;x_{n+1}\ldots x_z \;\; si\;\; c\;\; es\;\; una\;\; constante
$$

De esta forma, puede agregarse al sistema un nuevo combinador (K_c^n), para funciones constantes; y con regla de reescritura

(K) $K^n(c)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow c\ x_{n+1}\ldots x_z$

Análogamente, para las funciones Variable, en el sistema original la regla (M.2) se comporta

$L^n(i)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow x_{n-i}x_{n+1}\ldots x_z$ *si* i *es una variable*

Pudoendo entonces dar una nueva regla

(V) $V^n(i)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow x_{n-i}x_{n+1}\ldots x_z$

Y para expresiones más complejas podemos dar la regla (L) siguiente:

(L) $L^n(y)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow (\circ y\ (P\ x_0\cdots x_n))x_{n+1}\cdots x_z$

Con lo cual, eliminando la regla (M.2), el nuevo sistema queda:

(M.1) $\circ(x_c x_0x_1x_2\ldots x_n)\ (P\ y_m\ldots y_1y_0) \Rightarrow (x_c x_0'x_1'\ldots x_n')$,

$$\text{donde} \begin{cases} x_i' & = x_i \text{ if } x_i \text{ es una constante o de tipo } L^a(b) \\ & = y_k \text{ if } x_i \text{ es la variable } k \\ & = \circ x_i\ (P\ y_m\ldots y_1y_0), \text{en otro caso} \end{cases}$$

(K) $K^n(c)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow c\ x_{n+1}\ldots x_z$

(V) $V^n(i)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow x_{n-i}x_{n+1}\ldots x_z$

(L) $L^n(y)\ x_0x_1\cdots x_nx_{n+1}\cdots x_z \Rightarrow (\circ y\ (P\ x_0\cdots x_n))x_{n+1}\cdots x_z$

(S) $S^n(D\ x_i\ldots x_1x_0) \Rightarrow x_n$

(+) $+xy \Rightarrow x+y$

(=) $Cond\ x\ m\ n \begin{cases} \Rightarrow m, \text{ si } x = True \\ \Rightarrow n, \text{en otro caso} \end{cases}$

2.2 Estructuras de Datos

El primer paso en la "especificación" de la máquina es la identificación de las estructuras de datos necesarias.

Tanto en [11] como en [15] se concluye que una representación con grafos "de gran profundidad" lleva a implementaciones ineficientes. Por tanto, en esta implementación se utilizan celdas de tamaño variable, lo que permite una representación más compacta de los términos.

Las celdas son representadas por secuencias acotadas, lo que da la ventaja de poder acceder en tiempo constante a cada componente de las mismas.

Cada término de Multi-Combinadores Categóricos se denota por una lista

$$[x_0, x_1, \ldots x_n]$$

En la definición de los elementos de estas listas debe tenerse en cuenta la representación de

- Constantes.
- Variables.
- Operadores del lenguaje (cond, mas,...).
- Multi-abstracción.
- Constructor de Bloque de Datos.
- Selectores de Datos.

Además, cualquier término puede contener sub-términos complejos.

Son **términos** de la Máquina de Reducción sólo aquellas listas que respondan a alguno de los siguientes esquemas:

- $[cte(c), x_0, \ldots, x_z]$ con c una constante
- $[var(i), x_0, \ldots, x_z]$ con $i \in N$
- $[D(n), a_0, \ldots, a_n]$ con $n \in N$
- $[mas, a_0, a_1]$
- $[cond, a_0, a_1, a_2]$
- $[L(n), a_0, x_1, \ldots, x_z]$ con $n \in N$
- $[V(n,k), x_0, \ldots, x_z]$ con $0 <= k <= n$
- $[K(n,m), x_0, \ldots, x_z]$ con $n \in N$ y m constante
- $[alfa, x_0, \ldots, x_z]$
- $[P(n), b_0, \ldots, b_n]$

donde α es un término,
los a_i son constantes, variables o términos, y deben figurar en el término
los b_i son constantes o términos, y deben figurar en el término
los x_i son constantes, variables o términos y pueden omitirse.

Esta forma de presentar las celdas tiene el inconveniente de no expresar explicitamente la posibilidad de compartir términos. Esta posibilidad es tenida en cuenta en la implementación.

2.3 Distribución del ambiente de evaluación

Observando la regla (M.1), puede verse que el sistema de Multi-Combinadores categóricos permite dos formas de substituir valores en variables:

- Retardando la substitución hasta que sea necesaria para la evaluación.

- Substituyendo todas las variables del cuerpo de la función aplicada, en un solo paso.

Esta segunda forma (llamada "eager environment distribution") es considerada la más conveniente, pues no interfiere en el proceso de evaluación propiamente dicho, - dado básicamente por las reglas (K), (V) y (L), responsables por las Multi-β-reducciones.

El proceso de "eager environment distribution" presenta la ventaja de no necesitar de la genereción dinámica de celdas para el ambiente; pero puede caerse en el caso de substituir términos que durante la ejecución serían descartados.

La "función" *EnvDi* dada en la próxima sección realiza esta tarea.

2.4 Dinámica de la reducción

Con esto, es posible dar la máquina de reducción de grafos, que toma un término y devuelve su forma normal equivalente (si es que existe).

La idea básica de la función de evaluación es tomar un nodo del grafo y retornarlo si ya está en forma normal; si el nodo no está evaluado (es un nodo de Multi-aplicación), se realiza la operación indicada en él, componiendo un nuevo término, que será evaluado recursivamente.

Así, la función de evaluación queda

$$eval[cte(c), x_0, \ldots, x_n] = [cte(c), x_0, \ldots, x_n]$$

$$eval[undefined, x_0, \ldots, x_n] = [undefined]$$

$$eval[var(i), x_0, \ldots, x_n] = [var(i), x_0, \ldots, x_n]$$

$$eval[L(m), x_0, \ldots, x_n] = [L(m), x_0, \ldots, x_n]$$

$$eval[D(m), x_0, \ldots, x_n] = [D(m), eval\ x_0, \ldots, eval\ x_n]$$

$$eval[S^i, [D, x_0, \ldots, x_n]] = [x_i]$$

$$\begin{array}{l} eval[mas, a_0, a_1] = \\ \quad lett_1 = eval[a_0] \\ \quad in \\ \quad if t_1 = [cte(i)] \\ \quad then\ lett_2 = eval[a_2] \\ \quad in \\ \quad if t_2 = [cte(j)] \\ \quad then[cte(i+j)] \\ \quad else[undefined] \\ \quad else[undefined] \end{array}$$

Respectivamente para los otros operadores aritméticos y lógicos

$$eval[cond, a_0, a_1, a_2] =$$
$$\quad lett = eval[a_0]$$
$$\quad in$$
$$\quad ift = [cte(truc)]$$
$$\quad theneval[a_1]$$
$$\quad elseeval[a_2]$$

$$eval[V(n,k), x_0, \ldots, x_n, \ldots, x_z] = eval[x_{n-k}, x_{n+1}, \ldots, x_z]$$

$$CMC - eval[K(n,i), x_0, \ldots, x_n, \ldots, x_z] = [cte(i), x_{n+1}, \ldots, x_z]$$

$$eval[[L(n), y_0, \ldots, y_m], x_0, \ldots, x_n, \ldots, x_z] =$$
$$\quad eval(appendEnvDi([y_0, \ldots, y_m], [P(n), x_0, \ldots, x_n])[x_{n+1} \ldots x_z])$$

$$eval[alfa, x_0, \ldots, x_n] =$$
$$\quad eval(append(evalalfa), [x_0, \ldots, x_n])$$

$$EnvDi([x_0 \ldots x_n], [y_m \ldots y_0]) = [x'_0 \ldots x'_n]$$

$$donde \begin{cases} x'_i & = x_i \text{ si } x_i \text{ es una constante o de tipo } [L^a(b)\ldots] \\ & = y_k \text{ si } x_i \text{ es la variable } k \\ & = EnvDi([x_i], [y_m \ldots y_0]), \text{ sino} \end{cases}$$

Puede verse que esta máquina realiza las reglas de reescritura de Multi-Combinadores Categóricos, ya que ella es, básicamente, otra forma sintáctica para expresar dichas reglas.

3 Implementación de la Máquina de Reducción de Grafos

La implementación final de la máquina de reducción de Multi-Combinadores Categóricos está realizada en el lenguaje C, bajo el Sistema Operativo VMS, y sobre un equipo VAX/750.

La máquina consta de cuatro partes principales: lector, impresor, gestor de la memoria y evaluador.

El lector y el impresor son rutinas muy simples de estrada-salida. El módulo lector es el menos importante de esta parte del proyecto. Realiza la lectura de un "programa" en una codificación numérica del lenguaje de la máquina. No se puso mucho énfasis en permitir una gran facilidad de uso del lector, ya que en el futuro será substituido por un analizador sintáctico para la gramática de SASL que efectúe la carga de la memoria.

La rutina de impresión recorre recursivamente la lista resultado, imprimiendo de forma comprensible su contenido.

3.1 El gestor de la Memoria

En la especificación dada no se tienen en cuenta las limitaciones e inconvenientes provocados por la necesidad de trabajar con máquinas de memo-

ria finita. Por esto fue necesario usar un administrador de la memoria disponible. El mismo consta de dos partes:

- Un administrador de la memoria en uso, que gestiona por demanda, devolviendo la dirección de la primera de las células pedidas (que deben ser contiguas).
- Un *Garbage Collector*, que es invocado por el anterior.

Se trata de un "*Copying Garbage Collector*". Usa una técnica extremadamente simple, que consiste en tener dos heaps gemelos que se alternan en el uso. Cuando el gestor no posee en la memoria en uso la cantidad de celdas requeridas, se invoca al Garbage Collector, que copia (compactando) las celdas conocidas por el evaluador hacia la otra memoria. Luego de ésta operación, queda en uso la memoria recién asignada.

En [12] puede verse un algoritmo de Garbage Collection capaz de tratar el tipo de celdas usadas aquí (de longitud variable y con estructura cíclica). Este algoritmo no se implementó por razones de tiempo y prioridad en los objetivos del proyecto.

3.2 El Intérprete de Multi-Combinadores Categóricos

Es básicamente una traducción de las funciones dadas en la especificación; con la diferencia básica de tener en cuenta em manejo de la memoria.

Esto se hace mediante el mantenimiento de una estructura de datos que aquí es vista como una pila de referencias a direcciones de la memoria. En esta pila se almacenan las direcciones de los sub-términos alcanzables desde el intérprete. El Garbage Collector ve a ésta estructura de datos como un arreglo, y obtiene de allí las direcciones de comienzo de las celdas a copiar.

En realidad, se extiende el de esta "pila", utilizándola también para el pasaje de parámetros en las llamadas del evaluador (recordemos que se trata de un programa recursivo).

4 Consideraciones sobre la performance del sistema

En esta sección se hace un análisis comparativo entre el desempeño de la presente implementación de la máquina de Multi-Combinadores Categóricos y el de una implementación de K.R.C [19], basada en la máquina de combinadores de Turner[18].

4.1 Programas de Prueba

Se eligieron como programas de prueba la función de Fibonacci, dada por el algoritmo

```
fib n = (n < 2) -> 1; fib (n - 1) + fib (n - 2)
```

Esta función (evaluada en 20) es un "brenchmark" standard para lenguajes funcionales, haciendo uso extensivo de la recursión, y siendo un caso clásico de función estricta.

Otro programa de test es

```
twice twice twice succ 1
```

donde

```
twice f x = f (f x)
succ n    = n + 1
```

que explota el uso de funciones de orden superior[1].

Con estos tests se obtuvieron los siguientes resultados:

	M-C.C	K.R.C
fib 20	81 s	71 s
twice	0.10 s	0.10 s

Estos números son tiempos de CPU de VAX/750.

4.2 Análisis de los resultados

Aunque la diferencia de tiempo entre los medidos de *fib* 20 son del orden del 15 % a favor de K.R.C, puede decirse que los resultados obtenidos son favorables a la máquina de Multi-Combinadores Categóricos, si se tiene en cuenta que:

- K.R.C usa aritmética entera, mientras que M-C.C está implementado con aritmética real.
- K.R.C posee muchas rutinas implementadas en VAX-Assembler, cosa que podría hacerse también en M-C.C.
- M-C.C no está concluida. Al momento de redactar este artículo están comenzando a efectuarse optimizaciones a la implementación, removiendo ineficiencias, al adoptar una representación más compacta para las celdas, y al evitar la generacion de grafos no necesarios.

Conclusiones

Multi-Combinadores Categoricos demuestran ser un formalismo sencillo y que permite implementaciones eficientes para lenguajes funcionales con lazy evaluation.

Dado el poco tiempo transcurrido desde su desarrollo, creemos que todavía puede ser refinado, posibilitando implementaciones mas rápidas que ésta, y que, como la presente traten de obtener eficiencia, sin dificultar la claridad de los programas.

[1] Funciones que toman y/o devuelven funciones.

Agradecimientos

Agradecemos el apoyo de Claudio Hermida, Alberto Pardo y Regina Motz.

Este trabajo es parte de un proyecto de Pasantía de la ESLAI, y está siendo financiado por el Programa Argentino-Brasileño de Investigación y Estudios Avanzados en Informática - Proyecto ETHOS.

Referencias

1. H.P.Barendregt, The Lambda Calculus Its Syntax and Semantics, North Holland (1984).

2. N.G.DeBruijn, Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, Indag.Math. 34, 381-392 (1972).

3. G.Cousineau, P-L.Curien, & M.Mauny, The Categorical Abstract Machine, Functional Programming Languages and Computer Architecture, ed. J-P.Jouannaud, SLNCS 201.

4. P-L.Curien, Categorical Combinators, Sequential Algorithms and Functional Programming, Research Notes in Theoretical Computer Science, Pitman Publishing Ltd., (1986).

5. R.J.M.Hughes, The Design and Implementation of Programming Languages, Ph.D. Thesis, Oxford Univ. Comp.Lab., July 1983.

6. T.Johnsson, Compiling Lazy Functional Languages, PhD. Thesis, Chalmers University of Technology, Göteborg, 1987.

7. J.Lambek, From Lambda-calculus to Cartesian Closed Categories, in To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, ed J.P.Seldin and J.R.Hindley, Academic Press (1980).

8. R.D.Lins, A New Formula for The Execution of Categorical Combinators, Proceedings of 8th. International Conference on Automated Deduction, Oxford, England, July 1986, LNCS 230, p 89 - 98, Springer Verlag.

9. R.D.Lins, On the Efficiency of Categorical Combinators as a Rewriting System, Software Practice & Experience, Vol 17(8), 547-559, (August 1987).

10. R.D.Lins, Categorical Multi-Combinators, Proceedings of Third International Conference on Functional Programming and Computer Architecture, Portland, USA, September 1987, LNCS 274, p 60-79, Springer Verlag.

11. R.D.Lins & S.J.Thompson, Implementing SASL using Categorical Multi-Combinators, Computing Lab. Report N.47, The Univ. of Kent, at Canterbury, 1st. revision, June/88. (a ser publicado por Software Practice & Experience).

12. R.D.Lins & S.J.Thompson, Reference Counting Garbage Collection for Cyclic Structures, June/88 (a ser publicado como U.K.C. Lab. Report e submetido a Software Practice & Experience).

13. R.D.Lins, Compiling Categorical Multi-Combinators, en preparación.

14. M.A.Macedo, Uma Especificação VDM para uma máquina de Multi-Combinadores Categóricos, en preparación.

15. S.Peyton-Jones, The Implementation of Functional Languages, Prentice Hall, (1987).

16. D.Scott, Relating Theories of the Lambda-Calculus, in To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, ed. J.P.Seldin and J.R.Hindley, Academic Press (1980).

17. D.A.Turner, SASL Language Manual, Revised Version: Nov/83, Computing Lab., UNIKENT, (1983).

18. D.A.Turner, A New Implementation Technique for Applicative Languages, Software Practice and Experience, Vol 9, 31-49 (1979).

19. D.A.Turner, Recursion Equations as a Programming Language, Functional Programming and its Applications, J.Darlington, P.Henderson, and D.A.Turner eds., Cambridge University Press (1982).