Categorical Multi-Combinators

Rafael D.Lins

Computing Laboratory - The University of Kent - Canterbury - CT2 7NF - England.

Chesf - Comp. Hidro Elétrica do São Francisco - Recife - PE - Brazil

Abstract

Categorical Combinators form a formal system similar to Curry's Combinatory Logic. The original system was developed by Curien [4] inspired by the equivalence of the theories of typed λ -calculus and Cartesian Closed Categories as shown by Lambek [6] and Scott [13]. A new system for Categorical Combinators was introduced in [9]. This system uses a more compact notation for the code and needs a smaller set of rewriting rules. In this paper we introduce a new set of categorical combinators which is more efficient than the previous ones, in terms of compilation and execution complexity.

Keywords: Categorical Combinators, lambda calculus, functional programming, complexity

1 Introduction

Categorical Combinators form a formal system developed by Curien in [4], which is similar to Combinatory Logic. An approach to the execution of Categorical Combinators which uses a stack machine is described in [3]. As Turner showed in [14] stack machines do not provide an efficient implementation of lazy functional languages. Aiming to implement lazy functional languages in an efficient way using Categorical Combinators we developed a new system of Categorical Combinators, called Simplified Categorical Combinators [9]. This system is based on the original system by Curien, but has the advantages of presenting a linear relationship between the size of a lambda-expression and its categorical combinator equivalent (the original system is quadratic in the worst case [10]), and working with a very simple and small set of rewriting rules to execute the code.

In [11] we introduced a modification to the compilation algorithm, the **Id**-complementation of A-terms. In this paper we introduce another optimisation to Simplified Categorical Combinators. These modifications at the same time reduce the number of rewriting laws and make rewritings more efficient, keeping the complexity of the pattern matching algorithm unchanged. This system is called Linear Categorical Combinators.

Our aim in this paper is also, based on Linear Categorical Combinators, to generalise the concept of categorical combinators in order to get a more compact notation for the code and, using a very small set of rewriting rules, perform several rewritings at once. This is done by run-length encoding the linear code, changing the syntax of expressions, and analysing the dynamics of this rewriting system. The core of the system of Categorical Multi-Combinators works only with two rewriting laws with a very low pattern-matching complexity and avoids the generation of

trivially reducible sub-expressions. The compilation and execution complexity of this generalised code are analysed here.

We also draw considerations about implementation details of Categorical Multi-Combinators in graph and string reduction machines. Some comparisons between Categorical Multi-Combinators and Hughes' Supercombinators [5] are also presented.

2 Categorical Combinators as a Rewriting System

Categorical Combinators use De Bruijn's representation for variables. In De Bruijn [2] notation for the λ -calculus variables are replaced by a number corresponding to its position relative to the λ by which it is bound in the parse tree of the expression. The compilation algorithm for Categorical Combinators from DeBruijn's λ -calculus as introduced by Curien [4] is:

$$\begin{aligned}
 [\lambda.a] &\rightarrow \Lambda([a]) \\
 [ab] &\rightarrow App \circ ([a], [b]) \\
 [0] &\rightarrow Snd \\
 [n] &\rightarrow Snd \circ Fst^n \text{ if } n \geq 1, \text{ where } Fst^1 = Fst \text{ and } Fst^{n+1} = Fst \circ Fst^n \\
 [c] &\rightarrow \Lambda(Fst)c, \text{ where } c \text{ is a constant}
 \end{aligned}$$

In [4] Curien chooses particular orientations of the axioms and deduced equations of a Cartesian Closed Category ([6] and [13]); different selections of them will generate several different rewriting systems, for reducing the code generated by the compilation algorithm above. The system, which he calls CCL_β uses the following laws, and simulates λ -calculus β -reduction by a sequence of elementary reduction steps of rewritings on the categorical code.

$$(r.1) \quad (x \circ y) \circ z \Rightarrow x \circ (y \circ z)$$

$$(r.2) \quad Id \circ x \Rightarrow x$$

$$(r.3) \quad x \circ Id \Rightarrow x$$

$$(r.4) \quad Fst \circ \langle x, y \rangle \Rightarrow x$$

$$(r.5) \quad Snd \circ \langle x, y \rangle \Rightarrow y$$

$$(r.6) \quad \langle x, y \rangle \circ z \Rightarrow \langle x \circ z, y \circ z \rangle$$

$$(r.7) \quad App \circ \langle \Lambda(x), y \rangle \Rightarrow x \circ \langle Id, y \rangle$$

$$(r.8) \quad \Lambda(x) \circ y \Rightarrow \Lambda(x \circ \langle y \circ Fst, Snd \rangle)$$

The system above is not locally confluent [4].

In [9] we added to this rewriting system the following law, which will simulate a whole sequence of rewritings performed using the laws above:

$$App \circ \langle \Lambda(x) \circ y, z \rangle \Rightarrow x \circ \langle y, z \rangle$$

We also introduced in [9] a new compilation algorithm for Categorical Combinators from DeBruijn λ -calculus [2]. This algorithm is based on the original one with some simplifications concerning the treatment of variables, constants and applications. The following algorithm presents a linear

relationship between the size of source and compiled codes, while the original one, as was shown in [9], is quadratic in the worst case.

$$\begin{aligned}
 \llbracket \lambda.a \rrbracket &\rightarrow \Lambda(\llbracket a \rrbracket) \\
 \llbracket ab \rrbracket &\rightarrow \triangleleft \llbracket a \rrbracket, \llbracket b \rrbracket \triangleright \\
 \llbracket n \rrbracket &\rightarrow n, \text{ where } n \text{ is a variable} \\
 \llbracket c \rrbracket &\rightarrow c, \text{ where } c \text{ is a constant}
 \end{aligned}$$

The rewriting strategy we adopted in [9] is leftmost-outermost, i.e. we will look from left to right in an expression for the syntactically outermost pattern matching the left hand side of any of the rewriting rules. When we find it, this pattern will be rewritten and rewriting will resume from the outermost level of the new expression. In [10] we analysed the dynamics of the rewriting system of Categorical Combinators to avoid moving to the outermost level of an expression after each rewriting. The minimal set of rewriting rules to execute by a leftmost-outermost strategy the code generated by the compilation algorithm above is presented in [9], as follows

$$(R.1) \quad n \circ \langle x, y \rangle \Rightarrow (n - 1) \circ x, \text{ if } n > 0$$

$$(R.2) \quad 0 \circ \langle x, y \rangle \Rightarrow y$$

$$(R.3) \quad \triangleleft x, y \triangleright \circ z \Rightarrow \triangleleft x \circ z, y \circ z \triangleright$$

$$(R.4) \quad \triangleleft \Lambda(x), y \triangleright \Rightarrow x \circ \langle Id, y \rangle$$

$$(R.5) \quad \triangleleft \Lambda(x) \circ y, z \triangleright \Rightarrow x \circ \langle y, z \rangle$$

$$(R.6) \quad c \circ x \Rightarrow c, \text{ where } c \text{ is a constant}$$

The rewriting system which comprises the set of rules above is called **Simplified Categorical Combinators** [9]. In [9] we proved that leftmost-outermost reduction of Simplified Categorical Combinators “mimics” leftmost-outermost reduction of λ -terms, which is known to be a normalising strategy. Simulations presented in [11] show that Simplified Categorical Combinators present a space and time complexity of at least one order of magnitude better than Curien’s CCL_β , and are similar in performance to Turner’s combinators [14].

3 Complementing Λ -terms

We call a categorical combinator expression of the form $\Lambda(x)$, where x is an arbitrary expression, a **Λ -term**. A **complemented Λ -term** is a Λ -term right composed with another expression forming an expression of the type $\Lambda(x) \circ y$. A **highest level Λ -term** is a Λ -term not enclosed within another Λ -term in an expression. We say we **Id-complement** the highest level Λ -terms in an expression if whenever we find the outermost Λ -terms in an expression we **Id** complement them, i.e. if we have an expression of type $\Lambda(x)$ we compose it with the **Id** combinator. For instance, the expression

$$\triangleleft \Lambda(\Lambda(\triangleleft 0, \Lambda(\triangleleft 0, 1 \triangleright) \triangleright)), \Lambda(0) \triangleright$$

when **Id**-complemented will give us

$$\triangleleft \Lambda(\Lambda(\triangleleft 0, \Lambda(\triangleleft 0, 1 \triangleright) \triangleright)) \circ Id, \Lambda(0) \circ Id \triangleright$$

As one can observe the inner Λ -terms are not complemented.

If we analyse our set of rewriting laws we can see that (R.4) is a simplified case of (R.5).

$$\triangleleft \Lambda(x), y \triangleright \Rightarrow x \circ \langle \mathbf{Id}, y \rangle$$

$$\triangleleft \Lambda(x) \circ y, z \triangleright \Rightarrow x \circ \langle y, z \rangle$$

where the Λ -term in the former is not **Id**-complemented. In [11] we prove that if we **Id**-complement all the highest level Λ -terms we can delete (R.4) from our rewriting system.

4 Merging (R.1) and (R.2)

The categorical combinator code suffers a “metamorphosis” under rewriting. As was analysed in [9], the code changes its structure and generates an “environment”, in which to each variable there is associated a local value. The **Id** combinator is a place holder without any active function in the formation of this “environment”. When the leftmost-outermost rewriting pattern matches with the left hand side of rule (R.1), the variable will appear as a sub-expression of an expression like:

$$n \circ \underbrace{\langle \dots \langle \mathbf{Id}, x \rangle, \dots \rangle}_m, y, z \rangle$$

where $m \geq n + 1$. In our set of rewriting laws, the rules which deal with variables are:

$$(R.1) \ n \circ \langle x, y \rangle \Rightarrow (n - 1) \circ x, \text{ if } n > 0$$

$$(R.2) \ 0 \circ \langle x, y \rangle \Rightarrow y$$

The analysis of the behaviour of the categorical code allows us to unify these two rules in a schema representing an infinite set of rules.

$$n \circ \underbrace{\langle \dots \langle x_{m+1}, x_m \rangle, \dots \rangle}_{m \geq n+1}, x_1, x_0 \rangle \Rightarrow x_n$$

So our basic set of rewriting laws now is:

$$(R'.1) \ n \circ \underbrace{\langle \dots \langle x_{m+1}, x_m \rangle, \dots \rangle}_{m \geq n+1}, x_1, x_0 \rangle \Rightarrow x_n$$

$$(R'.2) \ \triangleleft x, y \triangleright \circ z \Rightarrow \triangleleft x \circ z, y \circ z \triangleright$$

$$(R'.3) \ \triangleleft \Lambda(x) \circ y, z \triangleright \Rightarrow x \circ \langle y, z \rangle$$

$$(R'.4) \ c \circ x \Rightarrow c, \quad \text{where } c \text{ is a constant}$$

The practical importance of this optimization is analysed in [10]. The categorical rewriting system which uses the set of reduction laws presented above is called **Linear Categorical Combinators**. From now on we will use a prefix notation for our operations, and replace $\triangleleft x, y \triangleright$ by $A \ x \ y$, $\langle x, y \rangle$ by $P \ x \ y$, and $\Lambda(x)$ by $L(x)$.

5 Arithmetic Operators

The way one introduces constants plays a very important rôle in the efficiency of the system. We work with constants as described in [9]. A constant function with value **k** represents a whole class of arrows with same name and codomain. This polymorphic way of introducing constants provided us a very natural way of expressing the right composition of a constant arrow with any other arrow in the category by rule (R.4). Functions over constants are constant arrows

themselves, which belong to this class. The interactions between constants and functions over constants are ruled by application. For example, the addition of two integers, say **2** and **3** will be represented by:

$$A (A \text{ add } 2) 3 \Rightarrow 5$$

where **add** is an arrow type $\mathbf{1} \rightarrow [I \rightarrow I]$ and an integer is of type $\mathbf{1} \rightarrow I$ (**1** is the terminal object in the category and **I** is an object of the category). In reality arithmetic operations will take place outside the categorical world. We will express numbers in general (integers, reals, etc) as constant arrows.

We will work with conditional expressions as being functions over constants. For the sake of simplicity we will work only with addition, subtraction, multiplication and integer division as arithmetic operations, and *iszero* as a conditional expression. This set of operations can be easily extended to include all in-built machine operations. At the categorical level these operators will be introduced by the following set of laws, where **x** and **y** are constants, and **m** and **n** are of any type:

$$\begin{aligned} A (A \text{ add } x) y &\Rightarrow x + y \\ A (A \text{ sub } x) y &\Rightarrow x - y \\ A (A \text{ mult } x) y &\Rightarrow x * y \\ A (A \text{ div } x) y &\Rightarrow x / y \\ A (A (A \text{ iszero } x) m) n &\Rightarrow \begin{cases} m, & \text{if } x = 0 \\ n \end{cases} \end{aligned}$$

6 Compiling Linear Categorical Combinators from λ -Calculus

In [10] we introduced a simple compilation algorithm for the direct translation of λ -expressions into the original set of Categorical Combinators, which can be easily modified to translate λ -expressions into Linear Categorical Combinators. This algorithm at the same time merges the steps of compiling a λ -expression into De Bruijn λ -calculus with the translation of a De Bruijn λ -expression into Linear Categorical Combinators. As we can observe the **Id**-complementation of Λ -terms is performed in the compilation process.

$$(t'.1) [\lambda x_i. a] = (\circ L(R_0^{x_i} a) \text{ Id})$$

$$(t'.2) [a \ b] = A [a] [b]$$

$$(t'.3) [c] = c, \text{ where } c \text{ is a constant}$$

$$(t'.4) R_{n_i, \dots, n_j}^{x_i, \dots, x_j} \lambda x_k. b = L(R_{n_i+1, \dots, n_j+1, 0}^{x_i, \dots, x_j, x_k} b)$$

$$(t'.5) R_{n_i, \dots, n_j}^{x_i, \dots, x_j} a \ b = A R_{n_i, \dots, n_j}^{x_i, \dots, x_j} a \ R_{n_i, \dots, n_j}^{x_i, \dots, x_j} b$$

$$(t'.6) R_{n_i, \dots, n_j}^{x_i, \dots, x_j} a = \begin{cases} a, & \text{if } a \text{ is a constant} \\ n_k & \text{if } a = x_k \end{cases}$$

We will do an example of the translation of a λ -expression using the algorithm above.

$$\begin{aligned} [\lambda x \lambda y. (y \lambda z. (x z))] &\stackrel{t'.1}{=} (\circ L(R_0^x \lambda y. (y \lambda z. (x z))) \text{ Id}) \\ &\stackrel{t'.4}{=} (\circ L(L(R_{1,0}^{x,y} (y \lambda z. (x z)))) \text{ Id}) \end{aligned}$$

$$\begin{aligned}
& \stackrel{v'.5}{=} (\circ L(L(A R_{1,0}^{x,y} y R_{1,0}^{x,y} \lambda z.(xz))) \text{Id}) \\
& \stackrel{v'.6}{=} (\circ L(L(A 0 R_{1,0}^{x,y} \lambda z.(xz))) \text{Id}) \\
& \stackrel{v'.4}{=} (\circ L(L(A 0 L(R_{2,1,0}^{x,y,z}(xz))) \text{Id}) \\
& \stackrel{v'.5}{=} (\circ L(L(A 0 L(A R_{2,1,0}^{x,y,z} x R_{2,1,0}^{x,y,z} z)) \text{Id}) \\
& \stackrel{v'.6}{=} (\circ L(L(A 0 L(A 2 R_{2,1,0}^{x,y,z} z)) \text{Id}) \\
& \stackrel{v'.6}{=} (\circ L(L(A 0 L(A 2 0)) \text{Id})
\end{aligned}$$

Let us check the result above by compiling the original expression via its De Bruijn's equivalent. The De Bruijn λ -expression of $\lambda x.\lambda y.(y\lambda z.(xz))$ is

$$\lambda\lambda(0 (\lambda(2 0)))$$

compiling this expression we get

$$\begin{aligned}
\llbracket \lambda\lambda(0 (\lambda(2 0))) \rrbracket &= L(\llbracket \lambda(0 (\lambda(2 0))) \rrbracket) \\
&= L(L(\llbracket (0 (\lambda(2 0))) \rrbracket)) \\
&= L(L((A [0] [(\lambda(2 0))])) \\
&= L(L((A 0 \llbracket \lambda(2 0) \rrbracket)) \\
&= L(L((A 0 L(\llbracket (2 0) \rrbracket)) \\
&= L(L((A 0 L(A [2] [0]))) \\
&= L(L((A 0 L(A 2 [0]))) \\
&= L(L((A 0 L(A 2 0)))
\end{aligned}$$

Id-complementing the expression above we get

$$\circ L(L((A 0 L(A 2 0))) \text{Id}$$

the same result was obtained by the direct compilation algorithm.

A natural way of implementing the compiling algorithm presented above seems to be to use a stack. The sequence of indices $n_i, \dots, n_j$ need not be implemented. It is presented here only for the sake of simplicity of notation. The depth of a variable in the stack, which corresponds to its De Bruijn's number, can be established by counting the number of elements we need to pop out of the stack to reach the variable. A more efficient implementation of the algorithm above can be obtained by the use of a hash table associating of $x_i, \dots, x_j$ with $n_i, \dots, n_j$ respectively. This implementation will provide us with constant time look-up, while in the former implementation we have linear time look-up. These implementation techniques are completely compatible with the other version of this algorithm for the direct compilation of Categorical Multi-Combinators we will present further on and should be borne in mind when we examine those.

7 Application as Juxtaposition

Linear Categorical Combinators use the following set of rewriting rules in a leftmost-outermost reduction strategy.

$$(\mathbf{R'.1}) \circ n \underbrace{(P (P (\dots (P}_{m} x_{m+1} x_m) \dots) x_1) x_0)} \Rightarrow x_n$$

$$(\mathbf{R'.2}) \circ (A x y) z \Rightarrow A (\circ x z) (\circ y z)$$

$$(R'.3) \ A (\circ L(x) \ y) \ z \Rightarrow \circ x \ (P \ y \ z)$$

$$(R'.4) \ \circ c \ x \Rightarrow c, \text{ where } c \text{ is a constant}$$

$$(+) \ A \ (A \ \text{add} \ x) \ y \Rightarrow x + y$$

$$(=) \ A \ (A \ (A \ \text{iszero} \ x) \ m) \ n \ \begin{cases} \Rightarrow m, & \text{if } x = 0 \\ \Rightarrow n \end{cases}$$

In the set of rewriting rules above application appears only in (R'.2), (R'.3), and arithmetic operations. We will assume that the result of the evaluation of an expression is of ground type, i.e. not of function type or containing an embedded function type. This assumption is completely reasonable once the execution of functional programs is driven by the printing mechanism and only ground type expressions are printed as result. If we assume the result of the evaluation of an expression is of ground type, in the result of a rewritten expression there are no L's. This means that every time we have in the leftmost-outermost position of the code an expression type $\circ L(x) \ y$ this expression is embedded in a context matching the left hand side of rule (R'.3), i.e.

$$\dots A (\circ L(x) \ y) \ z \dots$$

If we translate application as juxtaposition this would still be the leftmost-outermost redex in the expression, expressed as

$$\dots (\circ L(x) \ y) \ z \dots$$

In the case of expressions matching the left hand side of rule (R'.2), if we replace application by juxtaposition parenthesization will keep compatible the interaction order between expressions (terms are left associated if no parentheses exist). If we rewrite our set of rules using juxtaposition as application our context is still strong enough to tell us which rule is applicable, because any redex in this new system arises from a redex in the old system and the rewriting strategy is unchanged. Let us replace application by juxtaposition in our set of rules.

$$(R''.1) \ \circ n \ \underbrace{(P \ (P \ (\dots (P \ x_{m+1} \ x_m) \ \dots) \ x_1) \ x_0)}_m \Rightarrow x_n$$

$$(R''.2) \ \circ (xy) \ z \Rightarrow (\circ x \ z) (\circ y \ z)$$

$$(R''.3) \ (\circ L(x) \ y) z \Rightarrow \circ x \ (P \ y \ z)$$

$$(R''.4) \ \circ c \ x \Rightarrow c, \text{ where } c \text{ is a constant}$$

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

$$(=) \ =xmn \ \begin{cases} \Rightarrow m, & \text{if } x = 0 \\ \Rightarrow n \end{cases}$$

As one can observe no ambiguity is generated and both systems will always produce expressions of same semantic interpretation. Arithmetic operations in prefix notation in the source code (λ -expression) will now be compiled into themselves in categorical combinators with juxtaposition as application.

Implementations of the set of rewriting laws above in machines which work with the concrete syntax of expressions will increase the performance, because we use a more compact code and their efficiency, in general, depends heavily on the size of the code. This is the case with string-reduction machines. Machines which work with the abstract syntax of expressions, i.e. the

way expressions are parsed, instead of the way they are written, as graph-reduction machines do, will also become more efficient with the use of application as juxtaposition. As we will see later, we will work with larger expressions, instead of a nested sequence of small applicative fixed size sub-expressions. This change to the abstract syntax is suggested by merging left associated applications when these operations become juxtaposition.

8 Introducing Multi β -reduction

Rule (R".3) plays the central rôle in the rewriting of the code - It is β -reduction. A λ -expression will, via (R".3), generate an environment step by step, from which variables will fetch their arguments using rules (R".1) and (R".2).

If we assume the result of the evaluation of an expression is of ground type, as we mentioned above, there are no L 's in the result of a rewritten expression. As one can see, whenever we have a sequence of n nested L 's there must be at least $n+1$ juxtaposed arguments.

For instance, if we have the λ -expression:

$$(\lambda x \lambda y. xy)((\lambda z.+)3)5$$

This expression will be translated into categorical combinators with juxtaposition as application thus:

$$L(L(1\ 0\ 0))((L(+))3)5$$

Complementing Λ -terms, we have:

$$(\circ L(L(1\ 0\ 0))\ \text{Id})((\circ L(+)\ \text{Id})3)5$$

rewriting the expression above we will get:

$$\begin{aligned} &\stackrel{(R''.3)}{\Rightarrow} \circ L(1\ 0\ 0)\ (P\ \text{Id}\ ((\circ L(+)\ \text{Id})3)5) \\ &\stackrel{(R''.3)}{\Rightarrow} \circ (1\ 0\ 0)\ (P\ (P\ \text{Id}((\circ L(+)\ \text{Id})3)5)) \end{aligned}$$

As one can observe in the sequence of reductions above no other rewriting will take place until the local environment is completely formed. Therefore we can generate the environment in one step using a new rewriting rule for expressing β -reduction, stated as :

$$(\circ L^n(y)\ (P\ x_0))x_1 \cdots x_n x_{n+1} \cdots x_z \Rightarrow \circ y\ (P\ x_0 \cdots x_n)x_{n+1} \cdots x_z$$

where $L^n(y)$ is equivalent to $\underbrace{L(L(\cdots (L(y)) \cdots))}_n$ We must make the remaining rules compatible with this new code. Since we generated our environment in one step, we introduced a n -tuple $\mathbf{P}$ to express the sequence of nested pairs we had before. We will call the n -tuple $\mathbf{P}$ the multi-pair combinator, and rule (R".1) will be stated as :

$$\circ n\ (P\ x_m \cdots x_1 x_0) \Rightarrow x_n$$

For the sake of simplicity the sequence of indexes in the rules we newly introduced are in descending order.

Rule (R".2) can be generalised. It distributes the environment through a compound expression. Applications are expressed here as left-associated juxtapositions. Therefore when we have as our leftmost-outermost rewriting pattern

$$\cdots \circ (x_0 x_1 x_2 \cdots x_n)\ y \cdots$$

this is equivalent to

$$\dots \circ \{ \dots ((x_0 x_1) x_2) \dots \} x_n \} y \dots$$

which will always follow the reduction path:

$$\begin{aligned} & \xrightarrow{(R''_2)} \dots (\circ \{ \dots ((x_0 x_1) x_2) \dots \}) y (\circ x_n y) \dots \\ & \xrightarrow{(R''_2)} \dots ((\circ \{ \dots ((x_0 x_1) x_2) \dots \}) y (\circ x_{n-1} y)) (\circ x_n y) \dots \end{aligned}$$

successively, until becoming

$$\xrightarrow{(R''_2)} \dots (\circ x_0 y) (\circ x_1 y) (\circ x_2 y) \dots (\circ x_{n-1} y) (\circ x_n y) \dots$$

The intermediate patterns generated by rewriting of the code step-by-step, as we can observe in the sequence of reductions above, do not interact with the context they are within, until the environment is fully distributed. If we replace (R''_2) by

$$\circ (x_0 x_1 x_2 \dots x_n) y \Rightarrow (\circ x_0 y) (\circ x_1 y) (\circ x_2 y) \dots (\circ x_{n-1} y) (\circ x_n y)$$

we do not produce any theoretical problems, because we do not alter the rewriting strategy. The introduction of application as juxtaposition brings us simplicity in stating multi β -reduction without the necessity of introducing extra laws for unifying sequential applications. For example, if we have an expression like

$$\dots A (A (\circ L^2(1) (P \text{ Id})) \mathbf{2}) \mathbf{3} \dots$$

in order to apply multi β -reduction we would need to unify the expression above to

$$\dots A^2 (\circ L^2(1) (P \text{ Id})) \mathbf{2} \mathbf{3} \dots$$

Juxtaposition as application removes this inconvenience in the implementation of machines working with concrete syntax. In the case of abstract syntax machines this unification can occur at the implementation level, without the need to perform unification as a separate rewriting step. For instance, in the example above

$$\dots A (A (\circ L^2(1) (P \text{ Id})) \mathbf{2}) \mathbf{3} \dots$$

we can work with this expression as if it were already unified and backtrack to fetch all the arguments needed to perform the multi β -reduction.

We will summarize the core of our categorical multi-combinator machine.

$$(M.1) \quad \circ n (P x_m \dots x_1 x_0) \Rightarrow x_n$$

$$(M.2) \quad \circ (x_0 x_1 x_2 \dots x_n) y \Rightarrow (\circ x_0 y) (\circ x_1 y) (\circ x_2 y) \dots (\circ x_{n-1} y) (\circ x_n y)$$

$$(M.3) \quad (\circ L^n(y) (P x_0)) x_1 \dots x_n x_{n+1} \dots x_z \Rightarrow \circ y (P x_0 \dots x_n) x_{n+1} \dots x_z$$

$$(M.4) \quad \circ c x \Rightarrow c, \text{ where } c \text{ is a constant}$$

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

$$(=) = xmn \begin{cases} \Rightarrow m, & \text{if } x = 0 \\ \Rightarrow n \end{cases}$$

9 Removing Id-Complementation

The concepts introduced in the previous sections will lead us to a new compilation algorithm from DeBruijn λ -calculus into categorical multi-combinators:

$$\begin{aligned} \llbracket \underbrace{\lambda \cdots \lambda}_m . a \rrbracket &\rightarrow L^m(\llbracket a \rrbracket) \\ \llbracket ab \cdots z \rrbracket &\rightarrow \llbracket a \rrbracket \llbracket b \rrbracket \cdots \llbracket z \rrbracket \\ \llbracket n \rrbracket &\rightarrow n, \text{ where } n \text{ is a variable} \\ \llbracket c \rrbracket &\rightarrow c, \text{ where } c \text{ is a constant} \end{aligned}$$

We need to complement the highest level Λ -term, as described in section 3 above, with $(P \text{ Id})$.

Now let us analyse the role performed by the **Id** combinator in our rewriting system as it stands. Let us follow a sequence of reductions in an example. The λ -expression (we use boldface to denote constants),

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

translates into De Bruijn's λ -calculus as

$$((\lambda \lambda. 0((\lambda. 0)1)1) \mathbf{2}) +$$

This expression will translate into categorical multi-combinators, using the algorithm above, as

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

which when **Id**-complemented yields,

$$(\circ L^2(0(L^1(0)1)1)(P \text{ Id}) \mathbf{2} +$$

Using the rewriting rules we presented in the last section, the expression above rewrites

$$\begin{aligned} &\stackrel{M^3}{\Rightarrow} \circ(0(L^1(0)1)1) (P \text{ Id } \mathbf{2} +) \\ &\stackrel{M^2}{\Rightarrow} (\circ 0 (P \text{ Id } \mathbf{2} +))(\circ (L^1(0)1) (P \text{ Id } \mathbf{2} +)) (\circ (P \text{ Id } \mathbf{2} +)) \\ &\stackrel{M^1}{\Rightarrow} +(\circ (L^1(0)1) (P \text{ Id } \mathbf{2} +)) (\circ 1 (P \text{ Id } \mathbf{2} +)) \\ &\stackrel{M^2}{\Rightarrow} +((\circ L^1(0) (P \text{ Id } \mathbf{2} +)) (\circ 1 (P \text{ Id } \mathbf{2} +))) (\circ 1 (P \text{ Id } \mathbf{2} +)) \end{aligned}$$

As we analysed in [9] the **Id**-combinator is an “environment” initialiser without any “meaning”. The dynamics of our rewriting system allows us to state that no variable will select the **Id** combinator from an environment. The **Id** combinator is not the only environment initialiser, however. One environment can initialise another. This is the case we have in the last expression above. The environment $(P \text{ Id } \mathbf{2} +)$ will initialise the new environment which will be local to the variables in the Λ -term $L^1(0)$, i.e. 0. The expression above rewrites as follows,

$$\begin{aligned} &+((\circ L^1(0) (P \text{ Id } \mathbf{2} +)) (\circ 1 (P \text{ Id } \mathbf{2} +))) (\circ 1 (P \text{ Id } \mathbf{2} +)) \\ &\stackrel{M^3}{\Rightarrow} +(\circ 0 (P \text{ Id } \mathbf{2} + (\circ 1 (P \text{ Id } \mathbf{2} +)))) (\circ 1 (P \text{ Id } \mathbf{2} +)) \end{aligned}$$

At this point of the rewriting process we can observe the merging of the old environment with the arguments of the Λ -term to form the new environment. The expression $(\circ 1 (P \text{ Id } \mathbf{2} +))$ remains unchanged. The rewriting process resumes as follows,

$$\begin{aligned} &\stackrel{M^1}{\Rightarrow} +(\circ 1 (P \text{ Id } \mathbf{2} +)) (\circ 1 (P \text{ Id } \mathbf{2} +)) \\ &\stackrel{M^1}{\Rightarrow} + \mathbf{2} (\circ 1 (P \text{ Id } \mathbf{2} +)) \\ &\stackrel{M^1}{\Rightarrow} + \mathbf{2} \mathbf{2} \\ &\stackrel{\pm}{\Rightarrow} \mathbf{4} \end{aligned}$$

Another way of analysing the role of the **Id** combinator is as an argument for a highest level Λ -term. Instead of providing this argument with no information we will decrease by one the arity of the all highest level Λ -terms. This implies we should change rule M.3 to

$$(M'.3) \quad L^n(y) x_0 x_1 \cdots x_n x_{n+1} \cdots x_z \Rightarrow \circ y (P x_0 \cdots x_n) x_{n+1} \cdots x_z$$

Rule M.2 generates compositions which were dealt by M.3 directly and cannot be dealt by rule M'.3. An extra rule must be added to our rewriting system in order to make the environment distribution mechanism, expressed in rule M.2, compatible with the way we stated rule M'.3 above. This new rewriting law will remove compositions of Λ -terms originated by distributing environments to expressions of type $L^n(y)$. Instead of letting one environment initialise another directly we will perform an artifice. We will decompose the original environment and "pass" its information to the Λ -term it complements via application. The arity of this Λ -term must be increased in order to allow it to "fetch" the new arguments (originated by decomposing the environment) as well as the old ones. This transformation will be performed by the following rewriting rule,

$$(M'.5) \quad \circ L^n(y) (P x_0 \cdots x_m) \Rightarrow L^{n+m}(y) x_0 \cdots x_m$$

An application of rule M'.5 followed by rule M'.3, not necessarily immediatly, will be equivalent to the application of rule M.3 in the system with **Id**-complementation.

Now let us do the same example we did before using this new system. Our original categorical multi-combinator expression

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

will, instead of having the highest level Λ -terms **Id**-complemented, have them decreased yielding the following expression

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

this expression rewrites as

$$\begin{aligned} & \xRightarrow{M'.5} \circ (0(L^1(0)1)1) (P \mathbf{2} +) \\ & \xRightarrow{M.2} (\circ 0 (P \mathbf{2} +)) (\circ (L^1(0)1) (P \mathbf{2} +)) (\circ 1 (P \mathbf{2} +)) \\ & \xRightarrow{M.1} + (\circ (L^1(0)1) (P \mathbf{2} +)) (\circ 1 (P \mathbf{2} +)) \\ & \xRightarrow{M.2} + ((\circ L^1(0) (P \mathbf{2} +)) (\circ 1 (P \mathbf{2} +))) (\circ 1 (P \mathbf{2} +)) \end{aligned}$$

If we compare the rewriting steps performed so far with the first four rewriting steps in the previous sequence of reductions we can observe that they are basically the same. The latter one without the presence of the **Id** combinators. At this point in the former example the environment was merged. Here it will be decomposed using rule (M'.5) as follows,

$$\begin{aligned} & \xRightarrow{M'.5} + (L^2(0) \mathbf{2} + (\circ 1 (P \mathbf{2} +))) (\circ 1 (P \mathbf{2} +)) \\ & \xRightarrow{M'.5} + (\circ 0 (P \mathbf{2} + (\circ 1 (P \mathbf{2} +)))) (\circ 1 (P \mathbf{2} +)) \\ & \xRightarrow{M.1} + (\circ 1 (P \mathbf{2} +)) (\circ 1 (P \mathbf{2} +)) \\ & \xRightarrow{M.1} + \mathbf{2} (\circ 1 (P \mathbf{2} +)) \\ & \xRightarrow{M.1} + \mathbf{2} \mathbf{2} \\ & \xRightarrow{\pm} \mathbf{4} \end{aligned}$$

The remaining rewriting steps are equivalent to the last rewriting steps in the previous example if we delete the **Id** combinators.

Removing **Id** complementation gave us a more compact notation for the categorical multi-combinator code at the cost of introducing one extra rewriting law into our system. In the next section we will analyse the application of this extra rewriting law and show that it can be simplified.

10 Local and Global Variables

Rule (M'.5) above is used to distribute (via application) the components of an environment to a Λ -term which is not of the highest-level. We will analyse here how these Λ -terms are generated. In “real” functional programming languages with no explicit abstraction operation a script (program) is formed by a series of function definitions. These function definitions form closed λ -expressions, i.e there are no free variables. This means that each function in the script can be seen as a “block” which during rewriting will generate its own environment. For instance, the example from last section could have been generated by the following function definitions in our script,

$$\begin{aligned} exp &= (\lambda x \lambda y. y(idt\ x)\ x)\ 2 + \\ idt &= \lambda z. z \end{aligned}$$

If we replace the identifier *idt* by its λ -expression in *exp* then we get,

$$(\lambda x \lambda y. y(\lambda z. z\ x)\ x)\ 2 +$$

In section 9 we reduce this expression. Let us analyse what happens when the λ -expression of the *idt* function is the leftmost-outermost redex.

$$\begin{aligned} &= +((\circ L^1(0)\ (P\ \mathbf{Id}\ 2\ +))\ (\circ 1\ (P\ \mathbf{Id}\ 2\ +)))\ (\circ 1\ (P\ \mathbf{Id}\ 2\ +)) \\ &\stackrel{M'.5}{\Rightarrow} +(\circ 0\ (P\ \mathbf{Id}\ 2\ +\ (\circ 1\ (P\ \mathbf{Id}\ 2\ +))))\ (\circ 1\ (P\ \mathbf{Id}\ 2\ +)) \end{aligned}$$

As we can see above the environment $(P\ \mathbf{Id}\ 2\ +)$ does not “carry” any information necessary for the evaluation of the internal Λ -term $L^1(0)$. The variable 0 in this Λ -term will discard the information contained in $(P\ \mathbf{Id}\ 2\ +)$.

Not all λ -expressions compiled in a functional language present this “behaviour”. If we have local recursion in our language functions are closed λ -expressions if seen as a whole, but there may occur free variables in some of the functions defined internally. This is an exceptional case. The way we compile local recursion [12] uses a similar mechanism to that presented in rule M'.5. Non-local references to variables are resolved by passing values in explicitly. This suggests that we can simplify rule M'.5 stating it as

$$(M''.5)\ \circ L^a(b)\ (P\ x_0 \cdots x_m) \Rightarrow L^a(b)$$

The rule above can be merged with rule M.4. We will present the core of our Categorical Multi-Combinator rewriting system as

$$(M^*.1)\ \circ n\ (P\ x_m \cdots x_1 x_0) \Rightarrow x_n$$

$$(M^*.2)\ \circ (x_0 x_1 x_2 \dots x_n)\ y \Rightarrow (\circ x_0\ y)(\circ x_1\ y) \dots (\circ x_{n-1}\ y)(\circ x_n\ y)$$

$$(M^*.3)\ L^n(y) x_0 x_1 \cdots x_n x_{n+1} \cdots x_z \Rightarrow \circ y\ (P\ x_0 \cdots x_n) x_{n+1} \cdots x_z$$

$$(M^*.4)\ \circ c\ x \Rightarrow c, \text{ where } c \text{ is a constant or of type } L^a(b)$$

11 Compiling Multi-Combinators from λ -Calculus

Adding the concepts of categorical multi-combinators to the algorithm presented in section 6 we get a very simple and natural compiling algorithm for λ -expressions.

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

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

$$(T.3) [c] = c, \text{ where } c \text{ is a constant}$$

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

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

$$(T.6) R_{n_i \dots n_j}^{x_i \dots x_j} a = \begin{cases} a, & \text{if } a \text{ is a constant} \\ n_k, & \text{if } a = x_k \end{cases}$$

We will present an example of the translation of a λ -expression into categorical multicombinators using the algorithm above.

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

12 Code Complexity

Let us analyse the size complexity of an expression translated from λ -calculus into categorical multi-combinators. The size of a λ -expression is the same as its De Bruijn's equivalent. For the sake of simplicity we will analyse the complexity of the translation of a De Bruijn expression into Categorical Multi-Combinators.

The compilation algorithm from DeBruijn λ -calculus into categorical multi-combinators is:

$$\begin{aligned} [\underbrace{\lambda \dots \lambda}_m . a] &\rightarrow L^{m-1}([a]) \\ [ab \dots z] &\rightarrow [a] [b] \dots [z] \\ [n] &\rightarrow n, \text{ where } n \text{ is a variable} \\ [c] &\rightarrow c, \text{ where } c \text{ is a constant} \end{aligned}$$

The highest level Λ -terms must be decremented as explained in section 9 above. Given a λ -expression, we have:

$$\underbrace{\lambda \dots \lambda}_m . \underbrace{x_0 x_1 \dots x_{n-1}}_n$$

the size of this expression is the sum of the number of binders to the size of body of the expression.

$$\#(\underbrace{\lambda \dots \lambda}_m . \underbrace{x_0 x_1 \dots x_{n-1}}_n) = m + \sum_{i=0}^{n-1} \#(x_i)$$

where $\#(x_i) = 1$ if x_i is a variable or a constant.

Its categorical multi-combinator translation will be of size:

$$\#(\underbrace{\lambda \cdots \lambda}_m . \underbrace{x_0 x_1 \cdots x_{n-1}}_n) = 1 + \sum_{i=0}^{n-1} \#(x_i)$$

where $\#(x_i) = 1$ if x_i is a variable or a constant. All binders will be represented by only one categorical multi-combinator. As we see, Categorical Multi-Combinators gives us linear code with a smaller constant of proportionality on the upper bound than a λ -expression.

13 Efficiency Criterion

In the λ -calculus an optimal reduction strategy is a strategy, which always chooses the shortest reduction path leading to normal form, if it exists, in the reduction tree of a λ -expression. In the standard λ -calculus there is no optimal reduction strategy [1]. A pioneer work on optimality was the shared λ -calculus [15] developed by Wadsworth. This system, unfortunately, is not optimal in all cases.

Let us analyse the behaviour of Categorical Multi-combinators vis à vis this optimality criterion. Given a De Bruijn λ -expression it's translation into categorical multi-combinators is:

$$\underbrace{\lambda \cdots \lambda}_n . \underbrace{x \cdots z}_m \rightarrow L^{n-1} . \underbrace{x \cdots z}_m$$

We will call the sequence of reductions in categorical multi-combinators which takes a λ -equivalent expression into another λ -equivalent expression, only using multi β -reduction once, a **complete β -reduction step**.

A complete β -reduction step in categorical multi-combinators will in the best case (the case in which a function is presented with all arguments in a left associated way) comprise:

1. One step for multi β -reduction (using rule (M*.3)).
2. One step for environment distribution through the body of the function (using rule (M*.2)).
3. m steps for “argument fetching” (using rules (M*.1) and (M*.4)).

In the worst case, the case in which a function is presented with all arguments in a right associated way, a complete β -reduction step will take:

1. One step for multi β -reduction (using rule (M*.3)).
2. $(m-1)$ steps for environment distribution through the body of the function (using rule (M*.2)).
3. m steps for “argument fetching” (using rules (M*.1) and (M*.4)).

Let us summarize:

best case: $1 + 1 + m = m + 2$ steps

worst case: $1 + (m - 1) + m = 2m$ steps

In some circumstances the execution of a categorical multi-combinator expression may need fewer reduction steps then the optimal number of β -reductions involved in reducing its equivalent to normal form in the λ -calculus. For example, if we have the following λ -expression:

$$(\lambda x \lambda u \lambda v \lambda x \lambda y \lambda z . x) a b c d e f$$

The optimal reduction path will take six reduction steps, while the categorical multi-combinator equivalent will take only two reduction steps:

$$(L^5(2) \ a \ b \ c \ d \ e \ f) \xRightarrow{(M^{\bullet,3})} \circ 2 \ (P \ a \ b \ c \ d \ e \ f) \xRightarrow{(M^{\bullet,2})} d$$

In the example above if the body of our function were of size five the number of categorical multi-combinator reductions will be greater than the optimal in the λ -calculus.

14 Reduction Order & Optimisations

If we examine the core of our machine we can see that rule $(M^{\bullet,4})$

$$\circ c \ x \Rightarrow c, \text{ where } c \text{ is a constant or of type } L^a(b)$$

can be incorporated by the other rewriting laws in order to fully reduce the term and avoid the generation of unnecessary sub-expressions. For instance, if we have an expression like

$$L^n(y) \ x_0 \ x_1 \ \dots \ x_n \ x_{n+1} \ \dots \ x_z$$

in which y is a constant, it will always reduce in leftmost-outermost order to

$$\Rightarrow \circ y \ (P \ x_0 \ \dots \ x_n) \ x_{n+1} \ \dots \ x_z$$

$$\Rightarrow y \ x_{n+1} \ \dots \ x_z$$

If we analyse the compilation algorithm and the dynamics of the reduction machine we can see that y cannot be of type $L^a(b)$ in the expression above. In the case of a compound expression under composition distributivity we have

$$\circ (x_0 x_1 \dots x_n) y \Rightarrow (\circ x_0 \ y) (\circ x_1 \ y) \dots (\circ x_n \ y)$$

The sub-expressions $(\circ x_i y)$ in which x_i is a constant or of type $L^a(b)$ will either not be reduced (the sub-expression is disposed of during the rewriting process) or will be reduced to the expression x_i itself. In the case of both expressions above we see that if during rewriting we test the type of the sub-expressions before copying we can save space. Less space is needed, because we do not generate trivially reducible sub-expressions. For the reasons presented above we will present the core of our machine as :

$$(G.1) \ \circ n \ (P \ x_m \ \dots \ x_1 x_0) \Rightarrow x_n$$

$$(G.2) \ \circ (x_0 x_1 x_2 \dots x_n) \ z \Rightarrow (x'_0 x'_1 \dots x'_n),$$

$$\text{where } \begin{cases} x_i = x_i \text{ if } x_i \text{ is a constant or of type } L^a(b) \\ x'_i = \circ x_i \ z \end{cases}$$

$$(G.3) \ L^n(y) \ x_0 \ x_1 \ \dots \ x_n x_{n+1} \ \dots \ x_z \begin{cases} \Rightarrow y \ x_{n+1} \ \dots \ x_z \text{ if } y \text{ is a constant} \\ \Rightarrow \circ y \ (P \ x_0 \ \dots \ x_n) x_{n+1} \ \dots \ x_z \end{cases}$$

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

$$(=) \ =xmn \begin{cases} \Rightarrow m, \text{ if } x = 0 \\ \Rightarrow n \end{cases}$$

Another optimization will be introduced into the set of rewriting rules above. An expression of the form

$$L^n(y) \ x_0 \ x_1 \ \dots \ x_n \ \dots \ x_z$$

in which y is a variable will always leftmost-outermost reduce to

$$\begin{aligned} &\Rightarrow \circ y (P x_0 x_1 \dots x_n) x_{n+1} \dots x_z \\ &\Rightarrow x_{n-y} x_{n+1} \dots x_z \end{aligned}$$

In a similar way, whenever we have the leftmost-outermost pattern matching with the left hand side of rule (G.2) we know that z is a multi-pair combinator. This expression is not obtainable from the translation of a λ -expression, but is produced by applying rule (G.3). We will be eager in fetching arguments of variables in expressions as the presented above. Our set of rewriting rules will be stated as:

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

$$\text{where } \begin{cases} x'_i = x_i \text{ if } x_i \text{ is a constant or of type } L^a(b) \\ x'_i = y_k \text{ if } x_i \text{ is a variable } k \\ x'_i = \circ x_i (P y_m \dots y_1 y_0) \end{cases}$$

$$(G'.2) L^n(y) x_0 x_1 \dots x_n x_{n+1} \dots x_z \begin{cases} \Rightarrow y x_{n+1} \dots x_z \text{ if } y \text{ is a constant} \\ \Rightarrow x_{n-y} x_{n+1} \dots x_z \text{ if } y \text{ is a variable} \\ \Rightarrow (\circ y (P x_0 \dots x_n)) x_{n+1} \dots x_z \end{cases}$$

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

$$(=) \quad = xmn \begin{cases} \Rightarrow m, \text{ if } x = 0 \\ \Rightarrow n \end{cases}$$

Let us now resume the example we presented in section 9 above. The λ -expression

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

translates into Categorical Multi-Combinators using the compilation algorithm presented in section 11 as,

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

using the laws above this expression rewrites as,

$$\begin{aligned} &\stackrel{G'.2}{\Rightarrow} \circ (0(L^0(0)1)1) (P \mathbf{2} +) \\ &\stackrel{G'.1}{\Rightarrow} + (\circ (L^0(0)1) (P \mathbf{2} +)) \mathbf{2} \\ &\stackrel{G'.1}{\Rightarrow} + (L^0(0) \mathbf{2}) \mathbf{2} \\ &\stackrel{G'.2}{\Rightarrow} + \mathbf{2} \mathbf{2} \\ &\stackrel{\pm}{\Rightarrow} \mathbf{4} \end{aligned}$$

15 A More Complex Example

We will present an example of a practical program analysing its compilation and execution. Let us assume we have the following script:

$$\begin{aligned} \text{twice } f \ x &= f (f \ x) \\ \text{suc } n &= 1 + n \\ \text{exp } a \ b \ c &= (a \ b) \ c \end{aligned}$$

the script above can be represented by the following λ -expressions:

$$\begin{aligned} twice &= \lambda f \lambda x. f (f x) \\ suc &= \lambda n. 1 + n \\ exp &= \lambda a \lambda b \lambda c. (a b) c \end{aligned}$$

which translate into Categorical Multi-Combinators using the compilation algorithm presented section 11 above as (we use boldface to denote constants),

$$\begin{aligned} twice &= L^1(1(1\ 0)) \\ suc &= L^0(+\ 1\ 0) \\ exp &= L^2(2\ 1\ 0) \end{aligned}$$

Now let us evaluate the expression

$$exp\ twice\ suc\ 1$$

which translates into Categorical Multi-Combinators as

$$L^2(2\ 1\ 0)\ L^1(1(1\ 0))\ L^0(+\ 1\ 0)\ 1$$

Using the set of rewriting rules presented in the last section this expression rewrites as

$$\begin{aligned} &\xrightarrow{G',2} \circ(2\ 1\ 0)\ (P\ L^1(1(1\ 0))\ L^0(+\ 1\ 0)\ 1) \\ &\xrightarrow{G',1} L^1(1(1\ 0))\ L^0(+\ 1\ 0)\ 1 \\ &\xrightarrow{G',2} \circ(1(1\ 0))\ (P\ L^0(+\ 1\ 0)\ 1) \\ &\xrightarrow{G',1} L^0(+\ 1\ 0)\ (\circ(1\ 0)\ (P\ L^0(+\ 1\ 0)\ 1)) \\ &\xrightarrow{G',2} \circ(+\ 1\ 0)\ (P\ (\circ(1\ 0)\ (P\ L^0(+\ 1\ 0)\ 1))) \\ &\xrightarrow{G',1} +\ 1\ (\circ(1\ 0)\ (P\ L^0(+\ 1\ 0)\ 1)) \\ &\xrightarrow{G',1} +\ 1\ (L^0(+\ 1\ 0)\ 1) \\ &\xrightarrow{G',2} +\ 1\ (\circ(+\ 1\ 0)\ (P\ 1)) \\ &\xrightarrow{G',1} +\ 1\ (+\ 1\ 1) \\ &\xrightarrow{+} +\ 1\ 2 \\ &\xrightarrow{+} 3 \end{aligned}$$

This same example if reduced using the non-optimised set of rules presented in section 10 would take twice as many rewriting steps and, in average, four times as much space as above.

16 Graph Reduction

The implementation of categorical combinators on Graph-Reduction machines was discussed in [10]. We will present here some of the desirable features to which in our opinion, implementations of Categorical Multi-Combinators should attend, in addition to the comments already formulated during the development of this system on the implementation of this code on abstract syntax machines.

16.1 Cells

A cell is the basic constituent of the dynamic storage system which allows the representation of the graph. A cell is divided into fields in which information (pointer, type, data, etc) is stored. In

order to obtain an efficient implementation of categorical multi-combinators the use of variable length cells seems to be desirable for the multi-pair combinator and multiple application. Each cell is formed by a contiguous block of memory. This will allow us to have constant time access to any element of a multi-pair combinator via

$$\circ n (P x_m \dots x_1 x_0) \Rightarrow x_n$$

We will present the structure of the different cells used in our graph-reduction machine for categorical multi-combinators.

A and **P** are positive integers and stand for the number of contiguous memory spaces needed by the multi-application and multi-pair combinators respectively (an additional flag is needed to distinguish between them). **N** is a negative integer and stands for the number of L's in the Λ -term. **O** is the composition cell. The information allocated in each memory space can be either a pointer, a variable, or a constant. In the rewriting process we will follow the leftmost sequence of pointers in the graph of an expression looking for a subgraph which pattern-matches with any of our rewriting laws. The optimisations presented in the last section have shown to be [8] of great importance in terms of the performance of Categorical Multi-Combinators. Less cells are needed, because we do not generate trivially reducible sub-graphs. Execution time is also better, because the time elapsed in testing the type of an expression is always less than the time involved in allocating cells, and copying pointers or information.

16.2 Garbage Collection

A major problem in the use of variable-length cell machines is the increase in the complexity of the garbage collector. The garbage collection routine is responsible for the re-cycling of memory space which contains information no longer needed for computation. Since our cells use a contiguous block of memory we should use a compacting garbage collector in the implementation of a graph reduction machine for categorical multi-combinators. In so doing we avoid memory fragmentation problems. For instance, if we need a contiguous block of memory to generate a cell of size 10 and after scanning all memory heap the required space is unavailable we are forced to garbage collect, even if in our memory heap there are several gaps which could provide smaller size cells. This situation can be repeated several times during the execution process. With a compacting garbage collector we can keep a pointer to the beginning of the free-memory block in the heap. If a new cell is needed we can allocate this information in the free-memory block, or in case of insufficient space garbage collect and compact all the information into another contiguous block of memory.

17 String Reduction

String reduction is a very simple implementation technique. The machine sequentially reads a string and outputs another in which the leftmost-outermost pattern matchable with any of the reduction laws is rewritten. The process will continue until reaching an irreducible expression (normal form). The string is always allocated in a sequential block of memory, therefore no garbage collection is necessary. One of the major inconveniences of string reduction is not allowing sharing (or at least not in a natural way). Sharing is a great advantage of graph-reduction machines, i.e. we allow multiple access to a subgraph. In [8] we use a string reduction machine to analyse the comparative efficiency of three sets of categorical combinators and multi-combinators, and we present some of the advantages and drawbacks of the use of string machines as an implementation basis for functional languages.

18 Multi-Combinators & Supercombinators

A work which has similar aims to the system of Categorical Multi-Combinators is Hughes' Supercombinators [5]. Both systems have the power to perform the equivalent of several β -reductions at once and in both of them an expression needs to have all its arguments present before evaluation.

There are, however, differences between these two systems. Categorical Multi-Combinators work with a fixed set of combinators which allows us the possibility of a hardware implementation. Supercombinators are generated during the compilation process and do not give us this flexibility.

The compilation algorithm for Categorical Multi-Combinators is extremely simple and generates expressions with size linear to the source code. Supercombinators use a more complex compilation algorithm due to the necessity of detecting maximal free expressions [5] (m.f.e's) in the code. The Supercombinator translation of a program of size N has size $O(N \log N)$ in the worst case.

Supercombinators are fully lazy [5]. This means that any sub-expression will be reduced at most once. Categorical Multi-Combinators as presented here are not fully lazy. If in our code we have a shared occurrence of a partial application of a function this sub-expression cannot be evaluated before being copied and therefore it may be evaluated more than once. However, if we use a "reverse" η -reduction introducing dummy variables to complete the arity of functions under partial application we can obtain a fully lazy implementation of Categorical Multi-Combinators.

19 Conclusions

Categorical Multi-Combinators use a very compact code, which is produced by a simple compilation algorithm. This code is reduced by applying a very small set of rewriting rules with a very low pattern-matching complexity. The equivalent of several β -reductions is performed at once. Categorical Multi-Combinators use a fixed set of combinators and can be implemented in hardware. For these reasons the author is convinced this system can provide an efficient basis for the implementation of lazy functional languages. This is borne out by some experimental results which will appear in [8, 12].

Acknowledgements

This work was developed under the supervision of Dr.Simon Thompson to whom the author is indebted for many discussions, suggestions, and comments.

The financial support for this work was provided by CNPq, Brazil, Grant 20.2744/84.

References

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. 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).
7. P.J.Landin, The Mechanical Evaluation of Expressions, Comput.J.6, (1963-4)
8. S.J.Levisseur & R.D.Lins, On The Efficiency of Categorical Combinators & Multi-Combinators, in preparation.
9. 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.
10. R.D.Lins, On the Efficiency of Categorical Combinators in Applicative Languages, Ph.D. Thesis, The University of Kent at Canterbury, October 1986.
11. R.D.Lins, On the Efficiency of Categorical Combinators as a Rewriting system, to appear in Software Practice & Experience, (1987).
12. R.D.Lins, Implementing SASL using Categorical Multi-Combinators, in preparation.
13. 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).
14. D.A.Turner, A New Implementation Technique for Applicative Languages, Software Practice and Experience, Vol 9, 31-49 (1979).
15. C.P.Wadsworth, Semantics and Pragmatics of the Lambda Calculus, Oxford Univ. D.Phil. Thesis, 1971.