Journal of Universal Computer Science, vol. 10, no. 7 (2004), 769-788
submitted: 16/2/04, accepted: 21/6/04, appeared: 28/7/04 © J.UCS

Partial Categorical Multi-Combinators and Church-Rosser Theorems

Rafael Dueire Lins
(Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
50.740-530, Recife, PE, Brazil
rdl@ee.ufpe.br)

Abstract: Categorical Multi-Combinators form a rewriting system developed with the aim of providing efficient implementations of lazy functional languages. The core of the system of Categorical Multi-Combinators consists of only four rewriting laws with a very low pattern-matching complexity. This system allows the equivalent of several β-reductions to be performed at once, as functions form frames with all their arguments. Although this feature is convenient for most cases of function application it does not allow partially parameterised functions to fetch arguments. This paper presents Partial Categorical Multi-Combinators, a new rewriting system, which removes this drawback.

Key Words: functional programming, categorical combinators, explicit substitutions.
Category: SD.F.4.1, SD D.3.2

1 Introduction

The method of compilation of functional languages into combinators, first explored by Turner in [Turner 1979], provides a way of removing the variables from a program, transforming it into an applicative combination of constant functions or **combinators**. Turner used a set of combinators based on Curry's Combinatory Logic. To each combinator there is associated a rewriting law. In rewriting a combinator expression, Turner rewrites the leftmost-outermost reducible subexpression (or *redex*) at each stage. When no further rewriting can take place the expression is said to be in **normal form**.

Another theory of functions is provided by Category Theory [Lambek 1980], and we can see the notation used herein as providing an alternative set of combinators. The original system of Categorical Combinators was developed by Curien [Curien 1986]. This work was inspired by the equivalence of the theories of typed λ-calculus and Cartesian Closed Categories as shown by Lambek [Lambek 1980] and Scott [Scott 1980].

Aiming to implement lazy functional languages in an efficient way using rewriting of Categorical Combinators we developed a number of optimisations [Lins 1996, Lins 1987a] of the naïve system, the most refined of which was the system of Linear Categorical Combinators [Lins 1987a]. The modifications introduced reduce the number of rewriting laws and increase the efficiency of the system by reducing the number of rewriting steps involved in taking an expression to normal form, whilst leaving the complexity of the pattern matching algorithm unchanged.

Categorical Multi-Combinators are a generalisation of Linear Categorical Combinators. Each rewriting step of the Multi-Combinator code is equivalent to several rewritings of Linear Categorical Combinators, since an application of a function to several arguments can be reduced in a single step. The core of the system of Categorical Multi-Combinators consists only of four rewriting laws with a very low pattern-matching complexity and avoids the generation of trivially reducible sub-expressions. In [Lins et al. 1994b] we have shown the equivalence between the operational semantics of the TIM [Fairbairn and Wray 1987] machine and rewriting of Categorical Multi-Combinator expressions: every TIM state is equivalent to a Categorical Multi-Combinator expression and *vice versa*; equivalent expressions are transformed into equivalent expressions by rewriting.

Independently, there has been much interest in compiled versions of functional languages which run much more quickly on von Neumann machines than do interpreters. Johnsson, with his implementation of Lazy ML [Johnsson 1987], showed that it is possible to get fast implementations of lazy functional languages. The basic principle of the G-Machine is to avoid generating graphs, as much as possible. An analysis of the G-Machine and its optimisations can be found in [Lins and Soares 1993]. Categorical Multi-Combinators served as a basis for several compiled machines [Lins and Lira 1993, Lins et al. 1994a, Musicante and Lins 1991, Thompson and Lins 1992]. The latest abstract machine, ΓCMC [Lins and Lira 1993], has already shown very good performance figures [Hartel et al.1996].

The system of Categorical Multi-Combinators allows the equivalent of several β-reductions to be performed at once, as functions form frames with all their arguments. This feature is convenient for most cases of function application because a coarser granularity of computation allows better compiled code. On the other hand, full lazyness is lost because partially parameterised functions are not reducible as such. Partial applications need to wait until the evaluation reaches a point in which all arguments are present, becoming a total application. If a partial application becomes shared in Categorical Multi-Combinators a copy of it is made for each instance of the variable to be replaced, losing the sharing of computations. Pseudoknot [Hartel et al.1996] is an example of a benchmark in which there is a large number of shared partial applications. In this paper we present a new set of Categorical Multi-Combinators, called Partial Categorical Multi-Combinators, which allows partial applications to be evaluated. We prove that Partial Categorical Multi-Combinators have the Church-Rosser properties of uniqueness of normal forms and that they are normalising, i.e. rewriting the leftmost-outermost pattern at each point of the reduction sequence leads to normal form, if it exists. Although, in this paper, the focus is on the formal aspects of Partial Categorical Multi-Combinators, the reader can find in appendix their compilation schemes and state transition laws for the ΓCMC machine [Lins and Lira 1993].

2 Categorical Multi-Combinators

In this section we present the compilation algorithm and rewriting laws for Categorical Multi-Combinators.

2.1 Compilation Algorithm

In Categorical Multi-Combinators, function application is denoted by juxtaposition, taken to be left-associative. The compilation algorithm for translating λ-expressions into Categorical Multi-Combinators is:

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

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

(T.3) $[c] = c$, where c is a constant.

(T.4) $R^{x_i \cdots x_j} \underbrace{\lambda x_k \ldots \lambda x_l}_{m} .a = L^{m-1}(R^{x_i \cdots x_j x_k \cdots x_l} a)$

(T.5) $R^{x_i \cdots x_j}(a \ldots b) = (R^{x_i \cdots x_j} a \ldots R^{x_i \cdots x_j} b)$

(T.6) $R^{x_i \cdots x_j} b = \begin{cases} b, & \text{if } b \text{ is a constant} \\ n_k, & \text{if } b = x_k \end{cases}$

In the case of rules T.1 and T.4 above, n and m stand for the largest possible sequence of binders, i.e. a may not be an abstraction. $R^{x_i \cdots x_j}$ is an auxiliary function which at T.6 replaces variables by its deBruijn number, the depth in the list of bound variables generated by T.1 and expanded by T.4. Rule T.5 above, simply distributes the environment (list of bound variables) through applications. Whenever applying rule T.6 above a variable b can be associated with more than one x_k one must choose the minimum corresponding n_k, keeping locality of binding.

2.2 Categorical Multi-Combinator Rewriting Laws

The core of the Categorical Multi-Combinator machine is presented on page 71 of [Lins 1987b]. For a matter of convenience the multi-pair combinator, which forms evaluation environments, is written as $(x_0, \ldots, x_n)$. Compositions, which represent closures, are denoted as $\langle a, b\rangle$. Using this notation the kernel of the Categorical Multi-Combinator rewriting laws is expressed as:

(M*.1) $\langle n, (x_m, \ldots, x_1, x_0)\rangle \Rightarrow x_n$

(M*.2) $\langle x_0 x_1 x_2 \ldots x_n, y\rangle \Rightarrow \langle x_0, y\rangle \ldots \langle x_n, y\rangle$

(M*.3) $\langle L^n(y), (w_0, \ldots)\rangle x_0 x_1 \ldots x_n x_{n+1} \ldots x_z \Rightarrow \langle y, (x_0, \ldots, x_n)\rangle x_{n+1} \ldots x_z$

(M*.4) $\langle k, (x_m, \ldots, x_1, x_0)\rangle \Rightarrow k,$ where k is a constant

The state of computation of a Categorical Multi-Combinator expression is represented by the expression itself. Rule (M*.1) performs environment look-up, this is the mechanism by which a variable fetches its value in the corresponding environment. (M*.2) is responsible for environment distribution. Rule (M*.3) performs environment formation. It is called multi-β reduction, because it is equivalent to performing several β-reductions in the λ-calculus. Rule (M*.4) discards the environment associated with a constant.

$(\lambda a.a\ a)((\lambda c\lambda d.d)B)C$ is translated into Categorical Multi-Combinators and rewritten as (we assume that $[B] = B'$ and $[C] = C'$),

$$\begin{aligned}
\langle L^0(0\ 0), ()\rangle(\langle L^1(0), ()\rangle\ B')C' &\overset{M*.3}{\Rightarrow} \langle (0\ 0), (\langle L^1(0), ()\rangle\ B')\rangle\ C' \\
&\overset{M*.2}{\Rightarrow} (\langle 0, (\langle L^1(0), ()\rangle\ B')\rangle \langle 0, (\langle L^1(0), ()\rangle\ B')\rangle)\ C' \\
&\overset{M*.1}{\Rightarrow} \langle L^1(0), ()\rangle\ B'\ \langle 0, (\langle L^1(0), ()\rangle\ B')\rangle)\ C' \\
&\overset{M*.3}{\Rightarrow} \langle 0, (B', \langle 0, (\langle L^1(0), ()\rangle\ B')\rangle)\rangle C' \\
&\overset{M*.1}{\Rightarrow} \langle 0, (\langle L^1(0), ()\rangle\ B')\rangle\ C' \\
&\overset{M*.1}{\Rightarrow} \langle L^1(0), ()\rangle\ B'\ C' \\
&\overset{M*.3}{\Rightarrow} \langle 0, (B', C')\rangle \\
&\overset{M*.1}{\Rightarrow} C'
\end{aligned}$$

2.3 Reduction Order

It is a well known fact that leftmost-outermost reduction of λ-expressions is a safe but non-optimal reduction strategy. In the λ-expression

$$(\lambda a.a\ a)((\lambda c\lambda d.d)B)\ C$$

the reduction of the rightmost redex yields

$$\begin{aligned}
&\overset{\beta}{\Rightarrow} (\lambda a.a\ a)(\lambda d.d)\ C \\
&\overset{\beta}{\Rightarrow} (\lambda d.d)(\lambda d.d)\ C \\
&\overset{\beta}{\Rightarrow} (\lambda d.d)\ C \\
&\overset{\beta}{\Rightarrow} C
\end{aligned}$$

The sequence of reductions above is shorter than the leftmost-outermost one, because the partial application, which forms the rightmost redex in the expression above, was reduced before being copied. Functional programs often make use of partially applied functions [Hartel et al.1996]. A program that makes intensive use of partial applications which become shared during execution calls for an efficient mechanism allowing the sharing of computation to be kept.

If we analyse the sequence of reductions for Categorical Multi-Combinators we can see that the Categorical Multi-Combinator sub-expression equivalent to the rightmost redex in the λ-expression is not reducible by applying any of the rewriting laws above. Categorical Multi-Combinators will make copies of the partial application and "wait" until all arguments are present to perform multi-β reduction. As functional languages only print expressions of ground type, we know that the extra arguments needed will be in place whenever the partial application becomes the leftmost-outermost redex, thus making multi-β reduction possible. However, not being able to share the result of evaluation of partial applications has performance implications.

3 Partial Categorical Multi-Combinators

In this section we introduce Partial Categorical Multi-Combinators, a rewriting system which allows to reduce partially applied functions.

3.1 Compilation Algorithm

The compilation algorithm for translating λ-expressions into Partial Categorical Multi-Combinators is different from that presented above for Categorical Multi-Combinators. Now, instead of working with the deBruijn representation for variables we work with the co-deBruijn number, as we want variables to which arguments are passed first to be represented by smaller numbers than the ones which correspond to arguments passed later on. Parenthesisation of expressions is also made explicit. Thus the compilation algorithm for Partial Categorical Multi-Combinators from fully parenthesised λ-lifted expressions in the λ-Calculus is:

(T'.1) $[\underbrace{\lambda x_i \ldots \lambda x_j}_{n}.a] = \langle L^{n-1}(R^{x_j \ldots x_i} a), ()\rangle$

(T'.2) $[(\ldots(a\ b)\ldots c)] = (\ldots([a][b])\ldots[c])$

(T'.3) $[c] = c$, where c is a constant.

(T'.4) $R^{x_j \ldots x_i} \underbrace{\lambda x_k \ldots \lambda x_l}_{m}.a = L^{m-1}(R^{x_l \ldots x_k x_j \ldots x_i} a)$

(T'.5) $R^{x_j \dots x_i}(\dots(a\ b)\dots c) = (R^{x_j \dots x_i}a \dots R^{x_j \dots x_i}b)\dots R^{x_j \dots x_i}c$

(T'.6) $R^{x_j \dots x_i}b = \begin{cases} b, \text{ if } b \text{ is a constant} \\ n_k, \text{if } b = x_k \end{cases}$

The compilation algorithm for Partial Categorical Multi-Combinators follows the same remarks made on the compilation algorithm of Categorical Multi-Combinators. Again, if whenever applying rule T'.6 above a variable b can be associated with more than one x_k, then one must choose the maximum corresponding n_k. This enforces the locality of binding of variables. Observing the compilation algorithm above one can see that the only difference to the Categorical Multi-Combinators (rules T.1 to T.6) is the representation of variables by the co-deBruijn number.

3.1.1 Example of Compilation

Here follows an example of the compilation of a λ-expression into Partial Categorical Multi-Combinators, using the algorithm above:

$$
\begin{aligned}
[(((\lambda a.a\ a)((\lambda c\lambda d.d)B))\ C)] &\overset{T'.2}{=} (([(\lambda a.a\ a)]\ [((\lambda c\lambda d.d)B)])\ [C]) \\
&\overset{T'.1}{=} ((\langle L^0(R^a(a\ a)), ()\rangle\ [((\lambda c\lambda d.d)B)])\ [C]) \\
&\overset{T'.5}{=} ((\langle L^0(R^a a\ R^a a), ()\rangle\ [((\lambda c\lambda d.d)B)])\ [C]) \\
&\overset{T'.6}{=} ((\langle L^0(0\ R^a a), ()\rangle\ [((\lambda c\lambda d.d)B)])\ [C]) \\
&\overset{T'.6}{=} ((\langle L^0(0\ 0), ()\rangle\ [((\lambda c\lambda d.d)B)])\ [C]) \\
&\overset{T'.2}{=} ((\langle L^0(0\ 0), ()\rangle\ ([\lambda c\lambda d.d]\ [B]))\ [C]) \\
&\overset{T'.1}{=} ((\langle L^0(0\ 0), ()\rangle\ (\langle L^1(R^{d,c}d), ()\rangle\ [B]))\ [C]) \\
&\overset{T'.6}{=} ((\langle L^0(0\ 0), ()\rangle\ (\langle L^1(1), ()\rangle\ [B]))\ [C])
\end{aligned}
$$

The size of compiled expressions in Partial and Categorical Multi-Combinators is exactly the same and is linear with their λ-calculus equivalent.

3.2 Partial Categorical Multi-Combinators Rewriting Laws

In this section we generalise multi-β reduction to allow a function to fetch fewer arguments than its arity passed to it. Thus one has,

$$(\dots(\langle L^n(y), (w_1, \dots)\rangle)x_0)\dots)x_m) \Rightarrow \langle L^{n-m-1}(\langle y, (x_0, \dots, x_m)\rangle), ()\rangle \ \textit{if}\ \ m < n$$

Now, one needs to adjust the argument fetching mechanism in such a way to allow variables to work with partial multi-β reduction.

$$\langle n, (x_m, \dots, x_1, x_0)\rangle \Rightarrow \begin{cases} x_n, \text{ if } n \le m \\ n - m - 1, \textit{otherwise} \end{cases}$$

The complete set of rewriting laws for Partial Categorial Multi-Combinators is:

(P.1) $\langle n, (x_m, \ldots, x_1, x_0)\rangle \Rightarrow \begin{cases} x_n, \text{ if } n \leq m \\ n - m - 1, \textit{otherwise} \end{cases}$

(P.2) $\langle x_0 x_1 x_2 \ldots x_n, y\rangle \Rightarrow \langle x_0, y\rangle \ldots \langle x_n, y\rangle$

(P.3) $(\ldots(\langle L^n(y), (w_1, \ldots)\rangle)x_0)\ldots)x_n)x_{n+1})\ldots)x_z) \Rightarrow$
$(\ldots\langle y, (x_0, \ldots, x_n)\rangle), x_{n+1})\ldots)x_z)$

(P.4) $(\ldots(\langle L^n(y), (w_1, \ldots)\rangle x_0)\ldots)x_m) \Rightarrow$
$\langle L^{n-m-1}(\langle y, (x_0, \ldots, x_m)\rangle), ()\rangle \ \textit{if} \ m < n$

(P.5) $\langle k, (x_m, \ldots, x_1, x_0)\rangle \Rightarrow k,$ where k is a constant

The fundamental difference between Partial and Categorical Multi-Combinators above is rewriting law P.4 above. It allows a function with less arguments than its arity to process the existing arguments yielding another function on the remaining arguments. Law P.4 restores an adequate degree of currying to the system of Categorical Multi-Combinators lost by λ-lifting, without incurring the penalty of having redundant laziness.

3.3 Example of Evaluation

Let us analyse the Partial Categorical Multi-Combinator expression presented in the example above under a reduction strategy similar to the one adopted for the reduction of the λ-expression in the last section, i.e. reducing the rightmost redex first.

$$(((\langle L^0(0\ 0), ()\rangle\ (\langle L^1(1), ()\rangle\ B'))\ C') \overset{P.4}{\Rightarrow} (((\langle L^0(0\ 0), ()\rangle\ (\langle L^0(\langle 1, B'\rangle\ ), ()\rangle)\ C')$$
$$\overset{P.1}{\Rightarrow} (((\langle L^0(0\ 0), ()\rangle\ \langle L^0(0), ()\rangle\ )\ C')$$

at this point the partial parameterisation of the function on the right hand side of the expression above was fully reduced, giving rise to a new function. Evaluation proceeds as follows:

$$\overset{P.3}{\Rightarrow} (\langle (0\ 0), (\langle L^0(0), ()\rangle\ )\rangle\ C')$$
$$\overset{P.2}{\Rightarrow} ((\langle 0, (\langle L^0(0), ()\rangle\ )\rangle)\ \langle 0, (\langle L^0(0), ()\rangle)\rangle\ C')$$
$$\overset{P.1}{\Rightarrow} (\langle L^0(0), ()\rangle\ \ \langle 0, (\langle L^0(0), ()\rangle)\rangle\ C')$$
$$\overset{P.3}{\Rightarrow} (\langle 0, (\langle\ 0, (\langle L^0(0), ()\rangle)\rangle)\rangle\ C')$$
$$\overset{P.1}{\Rightarrow} (\langle 0, (\langle L^0(0), ()\rangle)\rangle)\ C')$$
$$\overset{P.1}{\Rightarrow} (\langle L^0(0), ()\rangle)))\ C')$$
$$\overset{P.3}{\Rightarrow} \langle 0, (C')\rangle$$
$$\overset{P.1}{\Rightarrow} C'$$

Below, we prove that Partial Categorical Multi-Combinators have the Church-Rosser property allowing rewritings to take place in any order to reach normal form, if it exists. Notice that applicative order in Categorical Multi-Combinators yields expressions equivalent to λ-expressions in *head-normal forms.* Applicative order reduction of Partial Multi-Combinator expressions yields expressions equivalent to expressions in *normal* form in the λ-Calculus.

4 Church-Rosser Theorems

The first Church-Rosser theorem for the λ-Calculus proves the uniqueness of normal forms of λ-expressions, if they exist. This means that all terminating sequences of reductions of a λ-expression will lead to the same result. A rewriting system to which the Church-Rosser property is valid is called *confluent* or Church-Rosser. The second Church-Rosser theorem for the λ-Calculus shows that the reduction of the leftmost-outermost redex at each point of the reduction sequence leads to normal form, if it exists.

In this section, we show that Partial Categorical Multi-Combinators have the properties stated by the two Church-Rosser theorems.

4.1 Normal Forms

Here we prove that Partial Categorical Multi-Combinators have the property that if one starts from a Partial Categorical Multi-Combinators expression any terminating sequence of reductions leads to the same expression.

Our strategy for proving this property is based on Huet's version of the Knuth-Bendix algorithm [Huet 1980]. Huet proves that if a rewriting system if *left-linear* and has no *critical pairs* it is confluent. A rewriting system is said to be left-linear if no variable appears more than once on the left-hand side of any of its rewriting rules. Critical pairs are computed by a superposition algorithm, where one attempts to match in a most general way the left-hand side of some rewriting rule with a nonvariable subterm of all rewriting rules in the system, including itself. Critical pairs show the possibility of reduction sequences diverging.

The analysis of the of rewriting laws of Partial Categorical Multi-Combinators shows that there is no repeated variable on the left-hand side of any of the rewriting rules. Considering that rules P.3 and P.4 are mutually exclusive, there is no possible overlapping of patterns on the left hand side of any of the rewriting laws. Any rewritable pattern matches trivially with a variable of any of the rewriting laws in the system, therefore there are no critical pairs. We have proved that Partial Categorical Multi-Combinators form a confluent rewriting system, thus the Church-Rosser property of uniqueness of normal forms holds.

4.2 Normalisation Property

This section presents the proof that the reduction of the leftmost-outermost redex at each point of the reduction sequence leads to normal form, if it exists. A direct proof of this theorem is not simple. Our strategy is to produce a proof in three steps. First, we present the λ-Calculus with lazy explicit substitutions [Lins 1996], a rewriting system which performs β-reductions with explicit, on demand, variable substitution. The second step is to introduce the λ-Calculus with Multiple Substitutions, a rewriting system in which each rewriting step is equivalent to several β-reductions. Variable substitution is also performed explicitly and on demand. Then, we show that leftmost-outermost rewritings of Partial Categorical Multi-Combinators are equivalent to leftmost-outermost rewritings on the λ-Calculus with Multiple Substitutions, therefore equivalent in each step to a sequence of leftmost-outermost β-reductions on the λ-Calculus.

4.2.1 The λ-Calculus with Lazy Substitutions

The rewriting system called the λ-Calculus with lazy substitutions was introduced in 1986 by the author [Lins 1996] as a way to prove that the leftmost-outermost rewriting of Linear Categorical Combinators [Lins 1996, Lins 1987a] was equivalent to performing leftmost-outermost β-reductions in the λ-Calculus. Explicit substitutions were "rediscovered" and extended in [Abadi et al. 1991], but reference [Ferreira et al. 1996] and all other references afterwards acknowledge Lins as its original developer.

The rewriting laws in the λ-Calculus with lazy substitutions are:

l.1 $(\lambda x.a)A \Rightarrow [A/x]a$

l.2 $[A/x]\lambda z.a \Rightarrow \begin{cases} \lambda z.a, & \textit{if } x \neq z \\ \lambda z.[A/x]a, & \textit{if } x \neq z, \textit{and } z \textit{ not free in } A \\ \lambda z.[A/x][w/z]a, & \textit{where } w \textit{ is a new variable} \end{cases}$

l.3 $[A/x]\ (a\ b) \Rightarrow ([A/x]a)\ ([A/x]b)$

l.4 $[A/x]\ x \Rightarrow A$

l.5 $[A/x]\ z \Rightarrow z$

Rule 1.1 above is β-reduction with an explicit variable substitution operator $[A/x]$. Rules l.2 and l.3 shifts the substitution operator into the body of an abstraction and distributes it through an application, respectively. Rules l.4 and l.5 perform actual substitution of formal parameters for real parameters. It is obvious that the leftmost-outermost reduction in the λ-Calculus with lazy substitutions is equivalent to leftmost-outermost β-reduction in the λ-Calculus.

4.2.2 The λ-Calculus with Multiple Substitutions

Assuming we have the following λ-expression,

$$(\lambda x.\lambda y.\lambda z.a)\ T\ U\ V \ldots$$

applying the rules of the λ-Calculus with lazy substitutions it leftmost-outermost reduces to:

$$\begin{array}{l}
\overset{\Rightarrow}{l.1}\ ([T/x]\lambda y.\lambda z.a)\ U\ V \ldots \\
\overset{\Rightarrow}{l.2}\ (\lambda y.[T/x]\lambda z.a)\ U\ V \ldots \\
\overset{\Rightarrow}{l.1}\ ([U/y][T/x]\lambda z.a)\ V \ldots \\
\overset{\Rightarrow}{l.2}\ ([U/y]\lambda z.[T/x]a)\ V \ldots \\
\overset{\Rightarrow}{l.2}\ (\lambda z.[U/y][T/x]a)\ V \ldots \\
\overset{\Rightarrow}{l.1}\ ([V/z][U/y][T/x]a) \ldots
\end{array}$$

One can observe in the sequence of reductions above that no other rewriting takes place until all the substitution operators appear. There is no reason for not rewriting the top expression directly into the bottom one, as this is always the leftmost-outermost rewriting path, yielding:

$$(\lambda x.\lambda y.\lambda z.a)\ T\ U\ V \ldots \Rightarrow ([V/z][U/y][T/x]a) \ldots$$

Making this a new rewriting law and adopting a more convenient notation for the substitution operator we present a new rewriting system called the λ-Calculus with Multiple Substitutions:

m.1 $(\lambda x_1 \ldots \lambda x_n.a)A_1 \ldots A_m \Rightarrow \lambda x_{m+1} \ldots \lambda x_n.[A_1/x_1, \ldots, A_m/x_m]a$, if $m < n$

m.2 $(\lambda x_1 \ldots \lambda x_n.a)A_1 \ldots A_n A_{n+1} \ldots \Rightarrow [A_1/x_1, \ldots, A_n/x_n]aA_{n+1} \ldots$, otherwise

m.3 $[A_1/x_1, \ldots, A_n/x_n]\lambda z.a \Rightarrow \lambda z.[A_1/x_1, \ldots, A_n/x_n]a$

m.4 $[A_1/x_1, \ldots, A_n/x_n](a \ldots b) \Rightarrow ([A_1/x_1, \ldots, A_n/x_n]a \ldots [A_1/x_1, \ldots, A_n/x_n]b)$

m.5 $[A_1/x_1, \ldots, A_n/x_n]x_i \Rightarrow A_i$

m.6 $[A_1/x_1, \ldots, A_n/x_n]z \Rightarrow z$

In rule m.3 we assume that for all i we have $x_i \neq z$ and that z does not appear free in any expression A_i, α-conversion may be needed to guarantee this condition.

One can see by construction that the leftmost-outermost reduction in the λ-Calculus with Multiple Substitutions is equivalent to a sequence of leftmost-outermost reductions in the λ-Calculus with Lazy Substitutions, therefore any terminating sequence in the former gives rise to one in the latter; thus this reduction strategy is normalising.

4.2.3 Final Step

Now we prove that the rewriting of the leftmost-outermost pattern of Partial Categorical Multi-Combinators corresponds to rewriting the leftmost-outermost redex in the λ-Calculus with Multiple Substitutions. We first introduce a translation function $\mathcal{T}$, which translates Partial Categorical Multi-Combinator expressions into expressions of the λ-Calculus with Multiple Substitutions. The translation function $\mathcal{T}$ is defined as:

(t.1) $\mathcal{T}^{w_m \ldots w_1}(\ldots(a\ b)\ldots)l) = \mathcal{T}^{w_m \ldots w_1} a \mathcal{T}^{w_m \ldots w_1} b \ldots \mathcal{T}^{w_m \ldots w_1} l$

(t.2) $\mathcal{T}^{w_m \ldots w_1} L^n(\langle y, (x_1, \ldots, x_m)\rangle) =$
$$(\lambda w_{m+1} \ldots \lambda w_{n-m+1} . \mathcal{T}^{w_{m+n+1} \ldots w_1} \langle y, (x_0, \ldots, x_m)\rangle)$$

(t.3) $\mathcal{T}^{w_m \ldots w_1} \langle L^n(y), (x_m, \ldots, x_1)\rangle) = (\lambda w_1 \ldots \lambda w_{m+1} . \mathcal{T}^{w_{m+n} \ldots w_1} y)$

(t.4) $\mathcal{T}^{w_m \ldots w_1} \langle y, (x_n, \ldots, x_1)\rangle = [\mathcal{T}^{[]} x_1/w_1, \ldots, \mathcal{T}^{[]} x_n/w_n] \mathcal{T}^{w_{m+n} \ldots w_1} y$

(t.5) $\mathcal{T}^{w_m \ldots w_1} n = w_{n+1}$, if n is a variable.

(t.6) $\mathcal{T}^{w_m \ldots w_1} k = k$, if k is a constant.

One can observe that the translation function $\mathcal{T}$ is a correct mapping between the two rewriting systems by analysing the behaviour of original and translated expressions. One can see that $\mathcal{T}$ and $\mathcal{R}$ behave almost as inverses of each other. We show that if a Partial Categorical Multi-Combinator expression A leftmost-outermost rewrites in one step to an expression A', then the translation of A into the λ-Calculus with Multiple Substitutions, $\mathcal{T}\ A$, leftmost-outermost rewrites to $\mathcal{T}\ A'$, in one step. So, the following diagram commutes:

$$\begin{array}{ccc} A & \xrightarrow{\mathcal{T}} & \mathcal{T}\ A \\ \Downarrow & & \Downarrow \\ A' & \xrightarrow{\mathcal{T}} & \mathcal{T}\ A' \end{array}$$

We will analyse each of the rewritable patterns for Partial Categorical Multi-Combinators.

P.1

$$\begin{array}{lcl} \mathcal{T}^{[]}\langle n, (x_0, \ldots, x_m)\rangle & \stackrel{t.4}{=} & [\mathcal{T}^{[]} x_0/w_1, \ldots, \mathcal{T}^{[]} x_m/w_{m+1}] \mathcal{T}^{w_{m+1} \ldots w_1} n \\ \Downarrow P.1 & & \Downarrow t.5 \\ \mathcal{T}^{[]} x_n & & [\mathcal{T}^{[]} x_0/w_1, \ldots, \mathcal{T}^{[]} x_m/w_{m+1}] w_{n+1} \\ & & \Downarrow m.5 \\ & & \mathcal{T}^{[]} x_n \end{array}$$

The second clause in rule P.1 is never the leftmost-outermost redex in a Partial Multi-Combinator expression. If it were, there would be a situation equivalent to existing a free variable in the code.

P.2

$$
\begin{array}{r}
\mathcal{T}^{[]}(\langle(x_0,\ldots,x_n),(v_1,\ldots,v_m)\rangle) \stackrel{t.4}{=} \\
\Downarrow P.2 \\
\mathcal{T}^{[]}(\langle x_0,(v_1,\ldots,v_m)\rangle)\ldots(\langle x_n,(v_1,\ldots,v_m)\rangle) \\
\Downarrow t.1 \\
\mathcal{T}^{[]}(\langle x_0,(v_1,\ldots,v_m)\rangle)\ldots\mathcal{T}^{[]}(\langle x_n,(v_1,\ldots,v_m)\rangle) \\
\Downarrow t.4 \\
[\mathcal{T}^{[]}v_1/w_1,\ldots]\mathcal{T}^{w_m\cdots}x_0\ldots[\mathcal{T}^{[]}v_1/w_1,\ldots]\mathcal{T}^{w_m\cdots}x_n
\end{array}
$$

$$
\begin{array}{l}
\stackrel{t.4}{=} [\mathcal{T}^{[]}v_1/w_1,\ldots,\mathcal{T}^{[]}v_m/w_m]\mathcal{T}^{w_m\cdots}(x_0,\ldots,x_n) \\
\qquad \Downarrow t.1 \\
\quad [\mathcal{T}^{[]}v_1/w_1,\ldots]\mathcal{T}^{w_m\cdots}x_0\ldots\mathcal{T}^{w_m\cdots}x_n \\
\qquad \Downarrow m.4 \\
\quad [\mathcal{T}^{[]}v_1/w_1,\ldots]\mathcal{T}^{w_m\cdots}x_0\ldots[\mathcal{T}^{[]}v_1/w_1,\ldots]\mathcal{T}^{w_m\cdots}x_n
\end{array}
$$

P.3

$$
\begin{array}{r}
\mathcal{T}^{[]}(\ldots(\langle L^n(y),(v_l,\ldots,v_1)\rangle x_0)\ldots)x_n)\ldots)x_z) \stackrel{=}{t.1} \\
\Downarrow P.3 \\
\mathcal{T}^{[]}(\ldots(\langle y,(x_0,\ldots,x_n)\rangle x_{n+1})\ldots)x_z) \\
\Downarrow t.1 \\
\mathcal{T}^{[]}(\ldots(\langle y,(x_0,\ldots,x_n)\rangle\ \mathcal{T}^{[]}x_{n+1}\ldots\ \mathcal{T}^{[]}x_z \\
\Downarrow t.4 \\
[\mathcal{T}^{[]}(x_0/w_1,\ldots,\mathcal{T}^{[]}x_n/w_{n+1})]\ \mathcal{T}^{w_{n+1}\ldots w_1}y\mathcal{T}^{[]}x_{n+1}\ldots\ \mathcal{T}^{[]}x_z
\end{array}
$$

$$
\begin{array}{l}
\stackrel{=}{t.1}\ \mathcal{T}^{[]}(\ldots(\langle L^n(y),(v_l,\ldots,v_1)\rangle\mathcal{T}^{[]}x_0\ldots\ \mathcal{T}^{[]}x_n\ldots\mathcal{T}^{[]}x_z \\
\qquad\quad \Downarrow t.3 \\
\quad (\lambda w_1\ldots\lambda w_{n+1}.\mathcal{T}^{w_{n+1}\ldots w_1}y)\mathcal{T}^{[]}x_0\ldots\ \mathcal{T}^{[]}x_n\ldots\mathcal{T}^{[]}x_z \\
\qquad\quad \Downarrow m.2 \\
\quad [\mathcal{T}^{[]}(x_0/w_1,\ldots,\mathcal{T}^{[]}x_n/w_{n+1})]\ \mathcal{T}^{w_{n+1}\ldots w_1}y\mathcal{T}^{[]}x_{n+1}\ldots\ \mathcal{T}^{[]}x_z
\end{array}
$$

P.4

$$\begin{array}{r}
\mathcal{T}^{[]}(\ldots(\langle L^n(y),(v_l,\ldots,v_1)\rangle x_0)\ldots)x_m)\ \overset{=}{t.1} \\
\Downarrow P.4 \\
\mathcal{T}^{[]}(\ldots(L^{n-m-1}(\langle y,(x_0,\ldots,x_m)\rangle) \\
\|t.2 \\
(\lambda w_{m+1}\ldots\lambda w_{n-m-1}.\mathcal{T}^{w_{n-m}\ldots w_1}\langle y,(x_0,\ldots,x_m)\rangle) \\
\|t.4 \\
(\lambda w_1\ldots\lambda w_{n-m}.[\mathcal{T}^{[]}x_0/w_1,\ldots,\mathcal{T}^{[]}x_m/w_{m+1}]\ \mathcal{T}^{w_{n+1}\ldots w_1}y)
\end{array}$$

$$\begin{array}{l}
\overset{=}{t.1}\ \mathcal{T}^{[]}(\ldots(\langle L^n(y),(v_l,\ldots,v_1)\rangle\mathcal{T}^{[]}x_0\ldots\ \mathcal{T}^{[]}x_m \\
\qquad\Downarrow t.3 \\
\quad(\lambda w_1\ldots\lambda w_{n+1}.\mathcal{T}^{w_{n+1}\ldots w_1}y)\mathcal{T}^{[]}x_0\ldots\ \mathcal{T}^{[]}x_m \\
\qquad\Downarrow m.1 \\
\quad(\lambda w_1\ldots\lambda w_{n-m}.[\mathcal{T}^{[]}x_0/w_1,\ldots,\mathcal{T}^{[]}x_m/w_{m+1}]\ \mathcal{T}^{w_{n+1}\ldots w_1}y)
\end{array}$$

P.5

$$\begin{array}{ccl}
\mathcal{T}^{[]}\langle k,(x_1,\ldots,x_m)\rangle & \overset{=}{t.4} & [\mathcal{T}^{[]}x_1/w_1,\ldots,\mathcal{T}^{[]}x_m/w_m]\ \mathcal{T}^{w_m\ldots w_1}k \\
\Downarrow P.5 & & \|t.6 \\
\mathcal{T}^{[]}k & \overset{=}{t.4} & [\mathcal{T}^{[]}x_1/w_1,\ldots,\mathcal{T}^{[]}x_m/w_m]\ k \\
\|t.6 & & \Downarrow m.6 \\
k & & k
\end{array}$$

The translation function $\mathcal{T}$ makes no redex transposition during translation, thus the leftmost-outermost Partial Categorical Multi-Combinator redex corresponds to the leftmost-outermost redex in the λ-Calculus with Multiple Substitutions, in the sense that the rewriting of the leftmost-outermost redexes in both systems are equal modulo translation. We can conclude that leftmost-outermost rewritings of Partial Categorical Multi-Combinators is a normalising strategy.

5 Conclusions

Partial Categorical Multi-Combinators form a rewriting system, which performs the equivalent to a sequence of β-reductions in one rewriting step, but also allow for the reduction of partially parameterised functions. This paper shows that Partial Categorical Multi-Combinators have the Church-Rosser properties of uniqueness of normal forms and that the rewriting of the leftmost-outermost pattern leads to normal form, if it exists. The introduction of Partial Categorical Multi-Combinator to ΓCMC brought full lazyness to the machine, allowing for partial applications to be shared. This strategy has proved efficient in our implementation of Haskell [Carvalho Jr. et al. 2002, Lima et al. 2004], yielding a performance improvement of about 20% for the Pseudoknot benchmark [Hartel et al.1996].

Acknowledgements

The author is grateful to Simon Thompson (The University of Kent, U.K.) and to three anonymous referees for their comments on an earlier versions of this paper. The author acknowledges the financial support of CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico (Brazil).

References

[Abadi et al. 1991] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit Substitutions. Journal of Functional Programming, 1(4):375–416, 1991.

[Carvalho Jr. et al. 2002] F.H.Carvalho Jr., R.M.F.Lima and R.D.Lins. Coordinating Functional Processes with Haskell$_{\#}$. Proceedings of 17th ACM Symposium on Applied Computing, Madrid, Spain, March 2002.

[Curien 1986] P-L.Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Research Notes in Theoretical Computer Science. Pitman Publishing Ltd., 1986.

[Fairbairn and Wray 1987] J.Fairbairn and S.Wray. TIM: A simple, lazy abstract machine to execute supercombinators. In *Proc. of FP&CA'87*, pages 34–45. LNCS 274, Springer Verlag, 1987.

[Ferreira et al. 1996] M. C. F. Ferreira, D.Kesner and L.Puel. Lambda-Calculi with Explicit Substitutions and Composition Which Preserve Beta-Strong Normalization. in Proceedings of 5th International Conference on Algebraic and Logic Programming, pages 284-298, LNCS vol. 1139, Aachen, Germany, 25–27 September 1996.

[Huet 1980] G.Huet. Confluent Reductions: Abstract Properties and Applications to Term-Rewriting Systems. Journal of the ACM, 27(4):797-821, October 1980.

[Johnsson 1987] T.Johnsson. *Compiling Lazy Functional Languages*. PhD thesis, Chalmers Univ., Sweden, 1987.

[Lambek 1980] J.Lambek. ¿From lambda-calculus to cartesian closed categories. In J.P.Seldin and J.R.Hindley, editors, *To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, 1980.

[Lins 1996] R.D.Lins. A New Formula for the Execution of Categorical Combinators, in *Proceedings of 8th International Conference on Automated Deduction*, LNCS 230, pg 89-98, Springer Verlag, July/86.

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

[Lins 1987b] R.D.Lins. Categorical multi-combinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 60–79. Springer-Verlag, September 1987. LNCS 274.

[Lins and Lira 1993] R.D.Lins & B.O.Lira. ΓCMC: A Novel Way of Implementing Functional Languages, *Journal of Programming Languages*, 1:19-39, Chapmann & Hall, January 1993.

[Lins et al. 1994a] R.D.Lins, G.G.Cruz Neto & R.F.Lima. Implementing and Optimising ΓCMC, Proceedings of Euromicro'94, pp.353-361, IEEE Computer Society Press, Sep. 1994.

[Lins et al. 1994b] R.D.Lins, S.J.Thompson and S.Peyton Jones, On the Equivalence between CM-C and TIM, *Journal of Functional Programming*, 4(1):47-63, Cambridge University Press, January/1994.

[Lins and Soares 1993] R.D.Lins & P.G.Soares. Some Performance Figures for the G-Machine and its Optimisations. Microprocessing and Microprogramming 37(1993) 163-166, North-Holland.

[Lins and Thompson 1990] R.D.Lins & S.J.Thompson. Implementing SASL using categorical multi-combinators. *Software — Practice and Experience*, 20(8):1137–1165, November 1990.

[Lima et al. 2004] R.M.F.Lima, R.D.Lins & A.L.M.Santos. A back-end for GHC based on Categorical Multi-Combinators. *in Proceedings of 19th ACM Symposium on Applied Computing, vol(2):1482–1489, Cyprus, March 2004*

[Musicante and Lins 1991] M.A.Musicante & R.D.Lins. GMC: A Graph Multi-Combinator Machine. *Microprocessing and Microprogramming*, 31:31–35, North-Holland, April 1991.

[Thompson and Lins 1992] S.J.Thompson & R.D.Lins. The Categorical Multi-Combinator Machine: CMCM, *The Computer Journal*, vol 35(2): 170-176, Cambridge University Press, April 1992.

[Hartel et al.1996] P. Hartel, M.Alt, R.D.Lins et al. Benchmarking Implementations of Functional Languages with Pseudoknot, a Float-Intensive Benchmark, *J. Functional Programming*, 6(4):621-655, 1996.

[Scott 1980] D.Scott. Relating Theories of the lambda-calculus. In J.P.Seldin and J.R.Hindley, editors, *To H.B.Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism*. Academic Press, 1980.

[Turner 1979] D.A. Turner. A new implementation technique for applicative languages. *SP&E*: 9, 1979.

A Appendix: Compiling Partial Categorical Combinators

Language compilation is a far more efficient implementation technique than interpretation. ΓCMC [Lins and Lira 1993] was the first abstract machine for the implementation of lazy functional languages to transfer the execution flow control to C, as much as possible. ΓCMC glues together procedure calls, unevaluated expressions and functions, data-structures, etc. Categorical Multi-Combinators [Lins 1987b] are the basis for the evaluation model of the ΓCMC abstract machine.

This appendix briefly addresses the main issues on how Partial Categorical Multi-Combinators were compiled into ΓCMC.

A.1 The Original ΓCMC Machine

The original ΓCMC abstract machine is presented in [Lins and Lira 1993]. A brief introduction to it is presented herein.

A.1.1 Compiling into ΓCMC

A program in ΓCMC is formed by a set of function definitions plus an expression to be evaluated. The expression to be evaluated is compiled by scheme $\mathcal{E}$ and each function in the script is compiled depending on its nature. Strict functions on all arguments which produce results of ground type are called *special* and are compiled directly as procedures in C. All arithmetic expressions are also translated directly into C code. This is the key to the efficiency of ΓCMC, because in doing so it takes advantage of the very fast implementation of procedure calls in RISC architectures. The compilation schemes for the kernel of ΓCMC are presented in [Lins and Lira 1993], and their behaviour can be summarised as follows:

Scheme $\mathcal{E}$ is responsible for the printing routine and driving the evaluation mechanism.

Scheme $\mathcal{S}$ is responsible for starting-up the compilation of special functions generating procedures in C.

Scheme $\mathcal{S}'$ is ancillary to $\mathcal{S}$ and is responsible for the compilation of inner parts of the body of a special function generating parts of procedure code in C.

Scheme $\mathcal{T}$ is responsible for the compilation of ordinary functions and generates code which is handled by the abstract machine.

Scheme $\mathcal{G}$ generates code which whenever executed fills the fields of a cell in an evaluation environment.

Scheme $\mathcal{T}'$ produces code which whenever executed generates cells on the top of the T-stack. It also makes parameters ready for special functions or arithmetic expressions whenever called inside an ordinary function.

Scheme $\mathcal{Z}'$ makes parameters ready for special functions or arithmetic expressions whenever called inside a cell generating scheme.

Scheme $\mathcal{L}$ is responsible for the compilation of lists and functions over lists.

A.1.2 Example of Compilation

Let us show an example of ΓCMC compiled code. The script:

```
fib n = if n<2 then 1 else fib(n-1) + fib(n-2)
twi f x = f (f x)
twi fib 5?
```

by using the schemes above whose rules are presented in [Lins and Lira 1993], is compiled as,

$$\begin{aligned}
\mathsf{twi\ fib\ 5} &\rightarrow \mathsf{MKEcell(2); MKEpc(A,1); MKEcte(5,0);}\\
&\quad \mathsf{Pushfun(twi); Popenv; print();}\\
\mathsf{A} &\rightarrow \mathsf{eval'(0); MKTk(0, fib((*(topT)) \Rightarrow rem.val));}\\
\mathsf{twi} &\rightarrow \mathsf{MKTcomp(A'); eval_env(1);}\\
\mathsf{A'} &\rightarrow \mathsf{MKTvar(0); eval_env(1);}\\
\mathsf{fib} &\rightarrow \mathsf{if\{n < 2\}return(1)}\\
&\quad \mathsf{else\{return(fib(n-1) + fib(n-2))\}}
\end{aligned}$$

As one can see, the result of compilation of the special function fib is a procedure in C, which needs only a heading with type declarations to be compiled and executed by the C compiler.

A.1.3 State Transition Laws

ΓCMC is as a state transition machine. A state of ΓCMC is a 5-uple

$$\langle C, T, H, O, E\rangle$$

in which each component is interpreted in the following way:

C: The code to be executed. This code is generated by the translation rules presented by the compilation schemes above.

T: The reduction stack. The top of **T** points to the part of the graph to be evaluated.

H: The heap where graphs are stored. The notation $H[d = e_1 \ldots e_n]$ means that there is in H a n-component cell named d. The fields of d are filled with $e_1 \ldots e_n$, in this order. Cells are fully-boxed.

O: The output.

E: The environment stack. Its top contains a reference to the current environment.

ΓCMC is defined as a set of transition rules. The transition

$$\langle C, T, H, O, E \rangle \Rightarrow \langle C', T', H', O', E' \rangle$$

must be interpreted as: "whenever the machine arrives at state $\langle C, T, H, O, E \rangle$, it can get to state $\langle C', T', H', O', E' \rangle$".

A.1.4 Sharing

Sharing of computation can bring substantial improvement to the performance of the machine. A number of ways introduced sharing to ΓCMC . The result of evaluation of special functions and arithmetic expressions is assigned to temporary C variables, being automatically shared by whoever points at it. Sharing of ordinary functions was implemented in ΓCMC by a mechanism similar to the one in CMCM [Thompson and Lins 1992], which is inspired in the frame update mechanism of TIM [Fairbairn and Wray 1987]. Compile-time analysis generate annotations (U combinator) to specify variables to be shared.
The U combinator performs the following state transition:
48. $\langle$ $\mathsf{U(i)}$.c,d.T, H[$e_0 = \ldots a_i \ldots$], O, e_0.E $\rangle \Rightarrow \langle$ c, d.T, H[e_0=$\ldots d \ldots$], O, e_0.E $\rangle$

The only exception not covered by this mechanism in ΓCMC is the sharing of partial applications. The CM-CM machine [Thompson and Lins 1992] obtained sharing at the cost of having dynamic code generation, similarly to TIM [Fairbairn and Wray 1987]. This low level mechanism claims for machine-level programming and ruins code portability. A much neater solution is offered by Partial Categorical Multi-Combinators, as explained below.

A.2 Partial Categorical Multi-Combinators in ΓCMC

The adoption of Partial Categorical Multi-Combinators impose two conceptual changes to ΓCMC . The first one is variable representation by the co-DeBruijn number. The second one deals with the possibility of evaluating partial applications.

A.2.1 Variables as co-DeBruijn numbers

Variables in Partial Categorical Multi-Combinators are represented by the co-DeBruijn number, and fetch their argument from the corresponding environment (the one on the top of E-stack), counting its depth in reverse order (left-to-right). This imposes minor changes to compilation schemes. For instance, in scheme $\mathcal{T}$ [Lins and Lira 1993], the original rules (7) and (8) are replaced by:

7. $\mathcal{T}^{y_0 \cdots y_j}[y_i] = \mathsf{MKTvar(i-j)};$ if y_i is evaluated
8. $\mathcal{T}^{y_0 \cdots y_j}[y_i] = \mathsf{eval\_env(i-j)};$ if y_i is not evaluated

All other schemes which compile variables are modified in a similar fashion.

A.2.2 Compiling Partial Applications

The CMC based ΓCMC implementation assumes that the compilation of an ordinary function forming an incomplete application yields code which assumes that needed parameters will appear dynamically, as the result of the evaluation of expressions is of ground type.

The original ΓCMC machine assumes that whenever the code of such a function appears, the function code is entered and its parameters are on the top of the T-stack. In the case of a shared partial application, in the original ΓCMC , the first call to the function will find an Update (shared) annotation amongst the parameters, such as,

$$f_n\ x_0\ \ldots\ x_i\ U(i)\ x_j\ \ldots\ x_n$$

The compilation of such an ordinary function in ΓCMC is performed by:

5. $\mathcal{T}^{y_0 \cdots y_j}[f_n] = \mathsf{MKenv(n+1); Pushfun(f_n); Popenv;}$

The original ΓCMC machine ignores the Update annotation, jumping over it to fetch parameters, making the application complete.

The correct evaluation mechanism to allow the partial application $f_n\ x_0\ \ldots\ x_i$ to be shared has to be able to process parameters x_0 to x_i in f_n and generate another function, whose code is shared. The implementation of Partial Categorical Multi-Combinators in ΓCMC allows MKenv to check for sharing annotations amongst the parameters on T-stack. In the original ΓCMC , whenever MKenv(n) reaches the leftmost position of the code the following state transition takes place:

10. $\langle$ MKenv(n).c, $d_1 \ldots . d_n \ldots d_m$.T, H,O, E $\rangle \Rightarrow$
$\langle$ c,$d_{n+1} \ldots d_m$.T, H [e=$d_1 \ldots d_n$], O, e.E $\rangle$

The state transition law above transfers the n-top pointers on T-stack to an environment cell in the heap H, referenced by the top of the environment stack E. By construction c is $\mathsf{Pushfun(f_n); Popenv;}$, which will enter the code for function $\mathsf{f_n}$ and dispose the corresponding environment cell (e), eventually.

In the case of the implementation of PCMC in ΓCMC, the rule for MKenv is replaced by:

10. $\langle$ MKenv(n).c, $d_1 \ldots . d_i\ U\ d_{i+1} \ldots d_n \ldots d_m$.T, H,O, E $\rangle \Rightarrow$
$\langle\ c^i$,$d_{i+1} \ldots d_m$.T, H [e=$d_1 \ldots d_i$], O, e.E $\rangle$

where, c^i is the version of function f_n for which i parameters are shared and $n-i$ are not. This compilation strategy imposes that every ordinary function of arity n is compiled into $(n-1)$ functions, allowing combinator definitions to be curried.

The compilation of PCMC into ΓCMC allows for an elegant way of obtaining sharing of partial applications without any need for dynamic code generation. The PCMC machine, which together with some other code optimisations, became $\mu\Gamma CMC$, was used to implement the lazy functional language Haskell$_\#$ [Carvalho Jr. et al. 2002, Lima et al. 2004] and proved efficient. Sharing of partial applications is intensively used in Pseudoknot [Hartel et al.1996]. In this benchmark $\mu\Gamma CMC$ performed about 20% faster than ΓCMC, which can already be considered amongst the best lazy functional compilers available.