Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science 135 (2006) 107–117

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Supporting Function Calls within PELCR

Antonio Cosentino[1]

*Dipartimento di Informatica, Sistemi e Produzione,
Università degli Studi di Roma "Tor Vergata", Viale del Politecnico 1, Rome, Italy*

Marco Pedicini[2]

*Istituto per le Applicazioni del Calcolo "M. Picone",
CNR, Viale del Policlinico 137, Rome, Italy.*

Francesco Quaglia[3]

*Dipartimento di Informatica e Sistemistica,
Università degli Studi di Roma "La Sapienza", Via Salaria 113, Rome, Italy.*

Abstract

In [10,11], PELCR has been introduced as an implementation derived from the Geometry of Interaction in order to perform virtual reduction on parallel/distributed computing systems.
In this paper we provide an extension of PELCR with computational effects based on directed virtual reduction [2], namely a restriction of virtual reduction [3], which is a particular way to compute the Geometry of Interaction [5] in analogy with Lamping's optimal reduction [6]. Moreover, the proposed solution preserves scalability of the parallelism arising from local and asynchronous reduction as studied in [11].

Keywords: Functional Programming, Optimal Reduction, Linear Logic, Geometry of Interaction, Virtual Reduction, Parallel Implementation.

[1] e-mail: antocos@tiscali.it

[2] e-mail: marco@iac.cnr.it

[3] e-mail: quaglia@dis.uniroma1.it

[4] Partially supported by CNR/CNRS cooperation project n. 16251 "Interaction and Complexity", 2004-2005.

1 Introduction

PELCR (Parallel Environment for optimal Lambda Calculus Reduction) is a software package supporting optimal lambda calculus reduction on parallel/distributed computing systems. It is devised as an interpreter for pure lambda calculus (complete) reduction, whose development relies on the Geometry of Interaction [5] and successive results in the field of functional programming and linear logic [3], which have shown that the reduction of lambda terms can be mapped onto a graph rewriting technique known as Directed Virtual Reduction (DVR), see [2]. Specifically, PELCR implements a particular strategy for DVR, referred to as Half Combustion (HC), see [10,11], which permits great exploitation of parallelism by allowing the composition between two edges coincident on a same node of the graph as soon as these edges become available to the processor hosting that node. A set of optimisations are also implemented within PELCR allowing a reduction of the communication overhead, and a fair policy for distributing dynamically originated load (i.e. new nodes and edges generated during the reduction) among processors.

Although pure lambda calculus has a variety of applications, many functional programming languages tend to deviate from it in order to become more attractive and effective for programmers, and to enrol the use of non-functional constructs. With respect to this point, let us cite the most diffused examples of functional languages, namely ocaml and haskell, both having additional base types and facilities for explicit interactions with the underlying operating system.

In this paper we show how it is possible to support similar extensions in PELCR, without preventing the possibility to exploit parallelism (and hence to achieve run-time effectiveness) arising from Geometry of Interaction. Our starting point is Mackie's work on the implementation of the Geometry of Interaction where the author extends Girard's algebra with extra generators for natural numbers and for the successor function. The new generators form an equational theory which defines a particular abstract data type.

By generalising Mackie's approach, we extend the applicability of PELCR as an environment for the execution of functional and imperative languages through automatic and adaptive distribution, where the functional parts take into account functional dependencies and where external functions, let say *x-functions* for short, make calls to imperative code implementing parts less prone to be specified in a functional language. Also, compared to Mackie's original proposal, which deals with functions with a single parameter, we address the case of functions with multiple parameters.

The rest of this paper is structured as follows. In Section 2, we provide the theoretical framework for the previously mentioned extension. In Section 3, we present the interpretation of extended lambda calculus in dynamic graphs with x-functions to be executed on top of directed virtual reduction. How to support the extension within PELCR is described in Section 4. In Section 5, we give an example of code using x-functions and report experimental data related to the run-time behaviour while varying the number of used processors.

2 Geometry of Interaction and Extended $\mathsf{L}^\star$

The extension we provide is obtained following the approach introduced by Mackie in [7] and then expanded by Pinto in [13]. This technique can be summarised with the addition of generators in $\mathsf{L}^\star$ with computational effects consisting of data to be manipulated and executables to be evaluated when interactions occur. This work explores this direction, and in fact we map those generators onto data structures and functions defined in external libraries implemented in C language.

Let us recall Pinto's example, we consider a new generator S representing the successor function $S : \mathbb{N} \to \mathbb{N}$ and a generator n for every integer $n \in \mathbb{N}$. Then a pair of specific interaction equations are given

$$\mathsf{S}^\star \mathsf{n} = (\mathsf{n}+1)\mathsf{S}^\star, \tag{1}$$
$$\mathsf{S}^\star \mathsf{S} = 1, \tag{2}$$

and added to the algebra $\mathsf{L}^\star$. Note that, while the evaluation of Equation (1) is presented as a rewriting, it has attached a computational task to compute the result $S(n)$. In fact, these rules are better understood in the following terms: we add to $\mathsf{L}^\star$ a generator S for the function and a generator N which stands for a ground type object, a natural number in this case. Any generator added to the algebra has a corresponding allocated memory space:

- to store its state n in the case of an object of type N, denoted by $\mathsf{N} : n$,
- to store a program address p in the case of a function, denoted by $\mathsf{S} : p$.

So now Equations (1) and (2) can be rephrased as

$$(\mathsf{S} : p)^\star(\mathsf{N} : n) = (\mathsf{N} : p(n))(\mathsf{S} : p)^\star, \tag{3}$$
$$(\mathsf{S} : p)^\star(\mathsf{S} : p) = 1. \tag{4}$$

Note that $p(n)$ is obtained by calling the function with address p with argument stored in N, and by storing the result in the space allocated for N.

We have two kinds of problems with this approach. First, we need to consider how to extend such an approach to functions with more than one argument: $f : A_1 \times A_2 \times \ldots A_k \to B$. Second, we have to consider generators for partially evaluated functions since, as in currification, the generator $\mathsf{F} \in \mathsf{L}^\star$

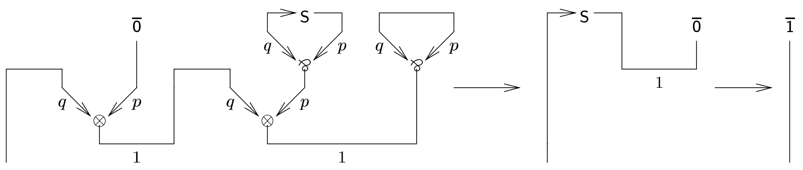

Fig. 1. Pinto's reduction of a term involving successor $((\lambda x.x)\lambda a.\mathtt{S}(a)\,\mathtt{0})$

associated with f, interacts with its arguments one by one.

In order to give the general form of Equations (1) and (2), we introduce a new family of generators, let say x-generators, with identifier i and constant lift 1, denoted by x_i; from the algebraic point of view x_i behaves like exponential generators of lift 1, we then specify the computational task associated with any generator, i.e. its computational effect.

For any x_i we have a type $\tau(i)$ and a $state(i)$, the state stores information on the type of the computational task, and on its evaluation status; essentially we have two classes of evaluation states:

- in case of *data*, denoted by $x_i : a$, x_i type is a ground type and its evaluation state is the stored value a;
- in case of *functions*, denoted by $x_i : (p, v)$, x_i type is a functional type and its evaluation state is given by a function pointer p and by an ordered list of values v of length strictly less than the arity of the function, note that the vector may possibly be the empty one.

For the sake of simplicity we suppose a unique ground type σ and the arrow type constructor, so the set S of types is $S := \sigma | \sigma \rightarrow S$. We denote $\sigma \rightarrow (\sigma \rightarrow \ldots (\sigma \rightarrow \sigma) \ldots)$ by $\sigma^n \rightarrow \sigma$.

Definition 2.1 For any x_i we have that $state(i) = \langle \tau, p, v \rangle$ where its type $\tau = \sigma^n \rightarrow \sigma \in S$, p is a reference to a function code, and $v = (a_1, a_2, \ldots, a_m)$ with $0 \leq m < n$. The case of data is treated as a particular case where $state(i) = \langle \sigma, \mathtt{null}, (a_1) \rangle$.

We now introduce the definition of the Geometry of Interaction algebra extended with x-generators. Interaction rules for x-generators are defined only for well typed data/function; all the other types of interaction are undefined. In Definition 2.2, and in particular in Equation (6) in Definition 2.3 below, we suppose that x_i has a functional type and x_j a ground data type, moreover we denote by s_i (respectively by s_j) its state $state(x_i) = \langle \sigma^n \rightarrow \sigma, p, (a_1, \ldots, a_m) \rangle$ (resp. $state(x_j) = \langle \sigma, \mathtt{null}, b \rangle$):

Definition 2.2 For any pair of x-generators x_i and x_j the `eval` function acts on the respective states s_i and s_j as follows:

$$\texttt{eval}(s_i, s_j) = \begin{cases} \langle \tau(i), p, () \rangle & m = n-1, \\ \langle \tau(i), p, (a_1, \dots, a_m, b) \rangle & m < n-1, \end{cases}$$

and

$$\texttt{eval}(s_j, s_i) = \langle \gamma, \texttt{null}, p(a_1, \dots a_{n-1}, b) \rangle,$$

if $\tau(i) = \sigma_1, \dots, \sigma_n \to \gamma$.

The next definition extends the usual presentation of Girard's dynamic algebra with interactions corresponding to the evaluation of the computational effects associated with x-generators:

Definition 2.3 The extended monoid $\mathsf{L}^\star$ of the Geometry of Interaction is the free monoid with a morphism $!(.)$, an involution $(.)^\star$ and a zero, generated by p, q, a family $W = (w_i)_i$ of exponential generators, and a family $X = (x_i)_i$ of x-generators such that for any $u \in \mathsf{L}^\star$:

$$a^\star b = \delta_{ab} \qquad \text{for } a, b = p, q, w_i, \tag{5}$$

$$(x_i : s_i)^\star (x_j : s_j) = \begin{cases} (x_i : \texttt{eval}(s_i, s_j))^\star & \text{if } i \neq j \text{ and } m = n-1, \\ (x_j : \texttt{eval}(s_j, s_i))(x_i : \texttt{eval}(s_i, s_j))^\star & \text{if } i \neq j \text{ and } m < n-1, \\ 1 & \text{if } i = j, \end{cases} \tag{6}$$

$$!(u)a = a!^{e(a)}(u), \qquad \text{where either } a = w_i \text{ either } a = x_i, \tag{7}$$

where δ_{ab} is the Kronecker operator, $e(a)$ is an integer associated with a called the *lift* of a; note that $e(x_i) = 1$ for all i, i is called the *name* of w_i or x_i and we will often write $w_{i,e(i)}$ to explicitly denote the lift of the exponential generator.

Orienting Equations (5-7) from left to right, one gets a rewriting system which is terminating and confluent, provided that x-function calls eventually return. The non-zero normal forms, known as *stable forms*, are the terms $ab^\star$ where a and b are *positive* (i.e., written without *s).

Example 2.4 Let us consider the following interaction:

$$(x_1 : (\sigma^2 \to \sigma, \&\texttt{ADD}(), ()))^\star (x_2 : (\sigma, \texttt{null}, 1))(x_2 : (\sigma, \texttt{null}, 3)),$$

it implicates the functional generator x_1 associated with function $ADD()$ of arity 2 from integers to integers, and reference $\&\texttt{ADD}()$, and the generator x_2

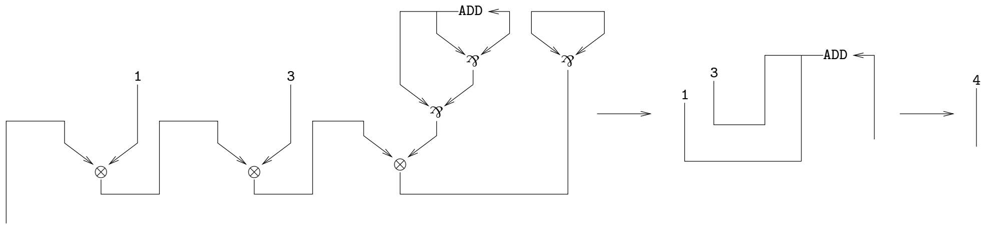

Fig. 2. Reduction of the term corresponding to $((\lambda x.x)\lambda a.\lambda b.\mathtt{ADD}(a,b)\,\mathtt{1}\,\mathtt{3})$

of a ground type for integer; it is reduced in the following way:

$$(x_1 : (\sigma^2 \to \sigma, \&\mathtt{ADD}(), ()))^\star(x_2 : (\sigma, \mathtt{null}, 1))(x_2 : (\sigma, \mathtt{null}, 3)) \to$$
$$\to (x_1 : (\sigma^2 \to \sigma, \&\mathtt{ADD}(), (1)))^\star(x_2 : (\sigma, \mathtt{null}, 3)) \to$$
$$\to (x_2 : (\sigma, \mathtt{null}, 4))(x_1 : (\sigma^2 \to \sigma, \&\mathtt{ADD}(), ()))^\star.$$

3 Encoding Extended Lambda Calculus in PELCR

In the previous section, we have introduced the extension of the dynamic algebra, and illustrated how to use it while considering the evaluation of arbitrary arity functions. In this section we sketch out how to fill the gap between the natural extension of term interpretation in the Geometry of Interaction by Mackie and Pinto, in Figure 1, and the analogous interpretation of an arity 2 function, see Figure 2.

We have to trade-off between apparently clashing requirements:

- to have a single execution path weighted with x-generators (see Figure 3.a);
- to map multiple arity functions to nodes with multiple links (see Figure 3.b);
- to execute interactions by using the parallel directed virtual reduction provided by PELCR (see Figure 3.d).

In fact, we have included in PELCR features allowing us to obtain all the above mentioned requirements. The first step is obtained by using the internalisation in DVR of a synchronisation scheme which reduces x-function and arguments interaction to a linear path with the correct configuration, i.e., like in Example 2.4, $F^\star A_1 \dots A_n$. This approach appeared in a simplified form as a construction to accommodate the conditional term of PCF, in Mackie's work on interaction nets [8].

Once we have obtained a single execution path, it must be proved not to alter global properties of directed virtual reduction, namely splitness and square-freeness [3], which are the basic properties to prove confluence and termination of VR.

As we show in Section 5, devoted to execution examples, although we

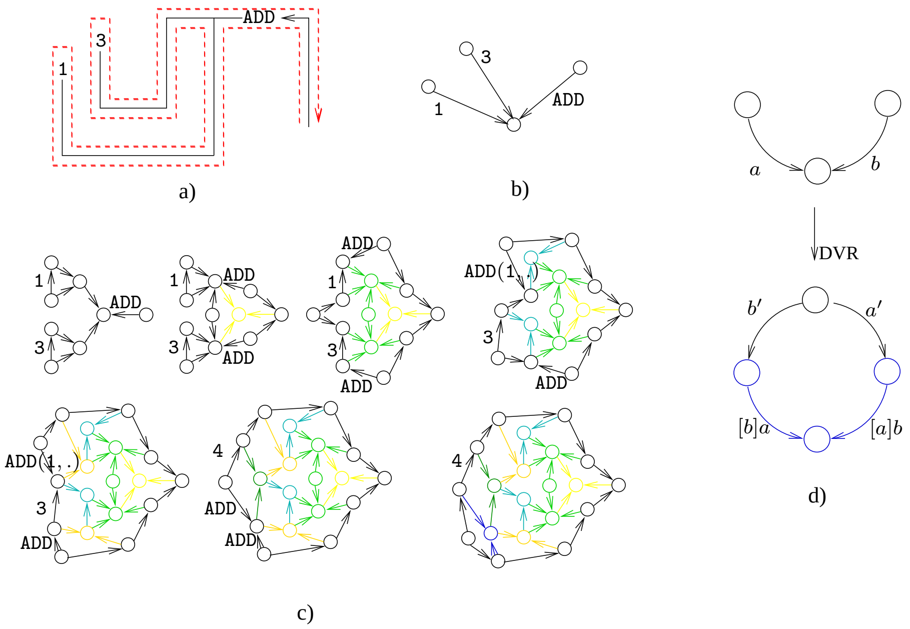

Fig. 3. The reduction of (ADD 1 3)

are forced to introduce a particular sequential evaluation pattern, the good properties on scalability and speedup of DVR are preserved, and so DVR is capable to exploit the available parallelism coming from functional specification of the program. Let us note that with our technique, we can exploit parallelism emerging from the functional part of the code, but we do not enter in x-functions which are treated as black boxes.

4 Supporting the $\mathsf{L}^\star$ Extension within PELCR

In the original version of PELCR, only dealing with pure lambda calculus, the basic operation executed while performing the reduction is the composition of pair of edges coincident on a same node (see Figure 3.d). This is supported through adequate data structures maintained by PELCR to represent the weights of the edges. These data structures are mostly based on string representation of the weights, and those strings constitute the essential part of the payload of each application message exchanged between distinct processes to notify each other of the existence of new edges in the graph originated by DVR steps.

To support the $\mathsf{L}^\star$ extension with no substantial modification of the basic mechanisms employed by PELCR for the support of parallelism (i.e., message aggregation and load balancing mechanisms whose benefits on the run-time behaviour have already been extensively tested in the context of pure lambda calculus), and with no variation in the HC strategy, which performs DVR in a parallel effective manner, we have extended the data structures maintained by PELCR in a way allowing a compact representation of x-functions to be eventually evaluated. The representation is based on a structured data type, namely `function_descriptor`, which implements the state associated with x-generators by maintaining: (i) a function pointer, allowing the retrieve of the code associated with the function when the evaluation needs to be performed, and (ii) a vector of structured entries, namely `parameters`, that, for each parameter to be passed to the function, indicates whether the parameter has already been stored and, in the positive case, stores the corresponding value.

The structured type `function_descriptor` has two additional fields storing the number of arguments for the function, and the number of arguments which have already been stored within the `parameters` vector. Hence, the evaluation of the function takes place as soon as the vector records the whole set of parameters required by the function itself. In accord with Equation (6) in Definition 2.3, this occurs when an edge carrying a `function_descriptor` for a function with n parameters, which already stores $n-1$ of these parameters, is composed through DVR with an edge carrying an additional parameter to be passed to the function (hence the whole set of parameters for the function gets completed).

The type of the return value for the function and the type of the parameters can be also selected by the user through a proper macro, namely `USERTYPE`. However, the current implementation of PELCR cannot cope with parameters of pointer type. This is because, while performing DVR steps, the `function_descriptor` and the function parameters might migrate across the different processes, so that the function might eventually be evaluated by a process that does not really host the buffer pointed by the parameter passed to the function. This is the same problem appearing in standard Remote Procedure Calls (RPCs), which require proper solutions we plan to introduce within next releases of PELCR.

The fact that the `function_descriptor` carries a function pointer, instead of the whole function code, allows a reduced size for the application messages, especially when dealing with functions having large size of the corresponding modules. On the other hand, this approach imposes the constraint that the function code needs to be available at all the processes so that the function can be evaluated by any of them while performing the reduction. However, in our implementation we allow a process to dynamically load the function code whenever required in the form of a Dynamic Linking Library (DLL). Beyond a better usage of memory, this approach also provides the advantage of permitting the expansion/modification of the set of supported functions

```
#uselib "./shared.so"

#def double = \l.\k.\x.((l)k)((l)k)x
#def map = \f.\l.\k.\x.((l)\y.(k)(f)y)x
#def F1 =\x.xfunction(foo)(x)
#def FF = \l.((map)F1)(double)l
#def AA = \k.\x.((k)123)((k)10)((k)3)x

#def five = \f.\x.(f)(f)(f)(f)(f)x
((five)FF)AA
#quit
```

Fig. 4. Benchmark code

without the need for recompiling the whole PELCR package.

5 Experimental Results

The experiments have been performed by using the code reported in Figure 4, which also gives an idea of the language that can be used within PELCR. One of the main motivations for the use of PELCR is that parallelism comes transparently for the end-user, which writes programs in a mixed functional/imperative language, taking advantage of the parallelism emerging from the functional structure and without having to explicitly use message passing primitives. For this application, we have used an SMP computing system, namely an IBM machine with 16 SP Power3 CPUs. The application works on lists of integers. The list structure is encoded in lambda calculus, whilst integers are 32bit unsigned integers, natively supported by the C compiler. The library (`shared.so`) provides the x-function and contains the definition of the numerical function `foo` programmed in C language. The functional program takes a list as input, and iterates, by means of an opportune Church numeral, the application of the function `FF` to the initial list `AA`. The lambda term `FF` gets a list l, builds a list obtained by concatenating l with itself and maps the x-function `foo` to the list.

Note that at each iteration the list doubles in size and so the application of the function may rise a good degree of parallelism as proved by the scalability shown in Figure 5. The reported data are very encouraging and essentially confirm good scalability for the execution parallelism achievable by PELCR even when using the computational effects hereby presented.

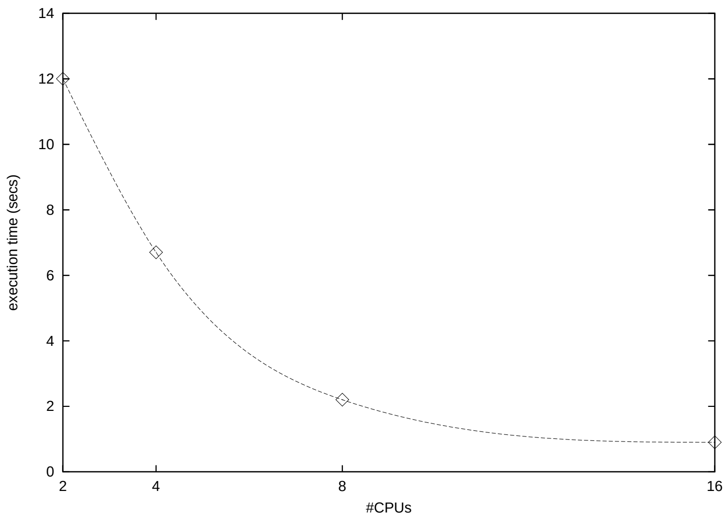

Fig. 5. Execution Time Results.

References

[1] A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in TCS*. Cambridge University Press, 1998.

[2] V. Danos, M. Pedicini, and L. Regnier. Directed virtual reductions. In M. Bezem D. van Dalen, editor, LNCS 1258, pages 76–88. EACSL, Springer Verlag, 1997.

[3] V. Danos and L. Regnier. Local and asynchronous beta-reduction (an analysis of Girard's EX-formula). LICS, pages 296–306. IEEE Computer Society Press, 1993.

[4] V. Danos and L. Regnier. Proof nets and the Hilbert space. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*. Cambridge University Press, 1995.

[5] J.-Y. Girard. Geometry of interaction 1: Interpretation of system F. In R. Ferro, et al. editors *Logic Colloquium '88*, pages 221–260. North-Holland, 1989.

[6] J. Lamping. An algorithm for optimal lambda calculus reduction. In *Proc. of 17th Annual ACM Symposium on Principles of Programming Languages*. ACM, San Francisco, California, pages 16–30, 1990.

[7] I. Mackie. The geometry of interaction machine. In *POPL*, pages 198–208, 1995.

[8] I. Mackie. YALE: yet another lambda evaluator based on interaction nets In ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming, pages 117–128, ACM, 1998.

[9] M. Pedicini. *Exécution et Programmes.* PhD thesis, Équipe de Logique Mathématiques, Université de Paris 7, 1999.

[10] M. Pedicini and F. Quaglia. A parallel implementation for optimal lambda-calculus reduction PPDP '00: Proceedings of the 2nd ACM SIGPLAN international conference on Principles and practice of declarative programming, pages 3–14, ACM, 2000.

[11] M. Pedicini and F. Quaglia. PELCR: Parallel environment for optimal lambda-calculus reduction. *CoRR*, cs.LO/0407055, accepted for publication on TOCL, ACM, 2005.

[12] J. S. Pinto. Parallel implementation models for the lambda-calculus using the Geometry of Interaction. In *TLCA*, pages 385–399, 2001.

[13] J. S. Pinto. *Parallel Implementation with Linear Logic (Applications of Interaction Nets and of the Geometry of Interaction)*. PhD thesis, École Polytechnique, 2001.

[14] L. Regnier. *Lambda-Calcul et réseaux*. PhD thesis, Université Paris VII, 1992.