# Sequential and Parallel Abstract Machines for Optimal Reduction

Marco Pedicini[1], Giulio Pellitta[2], and Mario Piazza[3]

[1] Department of Mathematics and Physics, University of Roma Tre
Largo San Leonardo Murialdo 1, 00146 Rome, Italy
marco.pedicini@uniroma3.it
[2] Department of Computer Science and Engineering, University of Bologna
Mura Anteo Zamboni 7, 40126 Bologna, Italy
pellitta@cs.unibo.it
[3] Department of Philosophy, University of Chieti-Pescara
Via dei Vestini 31, 66013 Chieti, Italy
m.piazza@unich.it

**Abstract.** In this paper, we explore a new approach to abstract machines and optimal reduction through streams, ubiquitary objects in mathematics and computer science. We first define a sequential abstract machine capable of performing directed virtual reduction (DVR) and then we extend it to its parallel version, whose equivalence is explained through the properties of DVR itself. The result is a formal definition of PELCR, a software for $\lambda$-calculus reductions based on the Geometry of Interaction. In particular, we describe PELCR as a stream-processing abstract machine, which in principle can also be applied to infinite streams.

## 1 Introduction

Starting from the pioneering work of Peter Landin [11] abstract machines describing the implementations of functional languages have been conceived of as bridges between a high-level language and a low-level architecture [10,1]. On the side of logic, the Curry-Howard isomorphism guarantees a direct correspondence between typed $\lambda$-calculus and constructive logic, so that concepts like $\lambda$-terms and formal proofs turn out to be different representations of the same mathematical objects: cut-elimination on proofs, indeed, may be regarded as identical to $\beta$-reduction on $\lambda$-expressions. This means that abstract machines can be described in mathematical terms as executions of programs. In particular, some abstract machines [12,3] have been proposed as a tool for studying the theory and implementation of optimal reduction of $\lambda$-calculus: these are the machines based on Geometry of Interaction (GoI), a mathematical framework developed by J.-Y. Girard in order to provide a semantics of linear logic and to model the dynamics of cut-elimination [7,9].

In this paper, we explore a new approach to abstract machines and optimal reduction through streams, ubiquitary objects in mathematics and computer science. We begin by designing a sequential abstract machine which performs

*directed* virtual reduction (DVR) [5], that is a variant of virtual reduction (VR) [6], which realises optimal reduction in a fine-grained way. More precisely, VR is a local and confluent reduction on graphs whose elementary computational step consists of adding to the graph (representing the state of the computation) new edges representing composed paths. In this way, an "algebraic trace" of the performed compositions is stored on the current graph without useless (re)compositions. On the other hand, DVR is a variant of VR which exploits the original algebraic machinery of GoI, by removing the added part of the algebra introduced in [6], and still manages to avoid recompositions.

Then, we proceed to extend the machine for DVR to its parallel version, so as to obtain the formal definition of the PELCR (Parallel Environment for optimal Lambda-Calculus Reduction) engine [13]. Because in all the considered cases the computation is based on GoI, within which computation is naturally parallel and confluent, the normal form resulting from the execution turns out to be the *same* across abstract machines characterized by *different* degrees of parallelism: *sequential*, *parallel synchronous*, *parallel asynchronous*. Whereas the sequential machine is very close to the definition of DVR (half-combustion strategy for DVR, indeed [13, §4]), its parallel version may correspond to the computation of different patterns of execution. In this case, we prove the soundness of parallel execution with respect to the sequential case. What we obtain in this paper is mainly an abstract setting for formally expressing the parallelism introduced in PELCR; the principal advantage resides in the possibility of giving a formal definition of different choices of the parallel execution model, so as to compare these models in a uniform setting. This kind of analysis will impact PELCR itself, leading also to an in-deep analysis of the parallel execution also from a quantitative point of view on many different models of machines [16].

This paper is organized as follows. In Section 2 we present PELCR as a device for optimal reduction, providing the basics notions and examples that will be useful in the sequel. In Section 3, we discuss Parallel Abstract Machine distinguishing between synchronous and asynchronous parallel machines. In Section 4, we sketch the soundness of the parallel computation with respect to the sequential one.

## 2 PELCR as a device for optimal reduction

We begin by defining the virtual machine employed in the implementation of the parallel evaluator. We describe the *half-combustion virtual machine* (`hc-vm`, for short) as an abstract computational device evaluating the execution formula in the sense of the GoI. This sequential machine realises the graph reduction introduced in [5], namely DVR. It is worth noting that with respect to other graph reduction techniques in optimal reduction, like in Asperti's BOHM [3], DVR yields a less synchronised computation, becoming a good candidate for parallel implementation. Moreover, with respect to VR, DVR permits to compute the same result by extending the algebraic structure of the GoI. In particular, the half-combustion strategy in DVR allows us to tackle the problem of recompo-

sition by means of Girard's *dynamic algebra* [7] (at the price of (re)introducing a bit of synchronisation in execution). As for the soundness of this system, the reader is referred to [5,13]. Being the basic one, the abstract machine without parallelism is taken as the starting point for presenting parallel versions.

An essential part of the algebraic computation peculiar to DVR lies on the notion of free inverse semigroup generated by a set. We introduce the basic algebraic structure with a property of normal form at the heart of DVR:

**Definition 1.** *Let $M$ be a monoidal algebra with identity element* 1 *(*i.e.*, $1 \cdot a = a \cdot 1 = a$, for any $a \in M$). $M$ is a* dynamic *monoid if $M$ is such that*

- *there exists $0 \in M$ s.t. $0$ is an absorbing element for product (*i.e.*, $0 \cdot a = a \cdot 0 = 0$, for any $a \in M$);*
- *$M$ is endowed with an inversion operator $(\cdot)^*$ (an involutive antimorphism for $0, 1$ and product, that is $0^* = 0$, $1^* = 1$ and $(a^*)^* = a$ and $(ab)^* = b^* a^*$, for any $a, b \in M$).*

**Definition 2.** *Let $M$ be a dynamic monoid. Let $a, b$ non-zero elements of $M$: we say that $b^* a$ has a (or can be rewritten in)* stable form *if there exist $a', b' \in M$ (uniquely determined by $a, b$) s.t. $b^* a = a' b'^*$.*

In fact, Girard's dynamic algebra $\Lambda^*$ is a dynamic monoid generated by *multiplicative* ($p$ and $q$) and *exponential* constants ($x_i$ for some $i$), endowed with a morphism denoted by !. We skip the details on the complete definition of dynamic algebra which is indeed specified by an additional set of equations to be satisfied by elements of $\Lambda^*$. For any exponential constant $x_i$, a non-negative integer $lift(x_i)$ which represents the difference between the box-depth of a *dereliction* link and the *pax* link corresponding to a given exponential variable, [4].

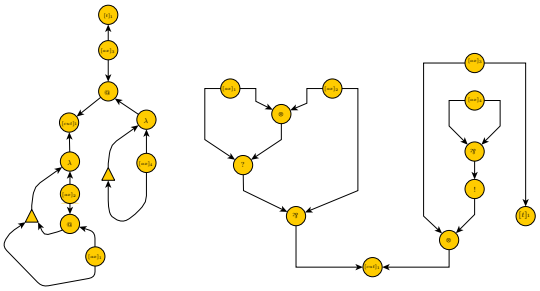
Let us introduce a basic example based on (untyped) $\lambda$-calculus:

*Example 1.* In Fig. 1, we give an example built from the pure $\lambda$-term representing the self application $\Delta = \lambda x.xx$ applied to the term $I = \lambda x.x$. Although this term cannot be directly typed, its representation in pure proof nets conforms to a representation in the GoI which leads to the following interpretation of the term $(\Delta I)$ given as a matrix in the GoI, the matrix can be also considered as an incidence matrix of a graph (the so called *virtual-net* [6, §6]):

$$
\begin{array}{c}
 \\
{\scriptstyle [ax]_1} \\
{\scriptstyle [ax]_2} \\
{\scriptstyle [ax]_3} \\
{\scriptstyle [ax]_4} \\
{\scriptstyle [cut]_1} \\
{\scriptstyle [t]_1}
\end{array}
\begin{array}{c}
\begin{array}{cccccc}
{\scriptstyle [ax]_1} & {\scriptstyle [ax]_2} & {\scriptstyle [ax]_3} & {\scriptstyle [ax]_4} & {\scriptstyle [cut]_1} & {\scriptstyle [t]_1}
\end{array} \\
\left(\begin{array}{cccccc}
0 & 0 & 0 & 0 & qx_2 + qx_1q & 0 \\
0 & 0 & 0 & 0 & qx_1p + p & 0 \\
0 & 0 & 0 & 0 & q!q + q!p & 0 \\
0 & 0 & 0 & 0 & p & 1 \\
x_2^* q^* + p^* x_1^* q^* & p^* x_1^* q^* + p^* & (!q^*)q^* + (!p^*)q^* & p^* & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0
\end{array}\right)
\end{array}
$$

The matrix representation is redundant (the matrix is in some sense symmetric $a_{ij} = a_{ji}^*$, in some model for $\Lambda^*$ we would say *hermitian*) and sparse. This drawback was one motivation in Danos and Regnier's paper [6] for considering *virtual reduction* using a graph as a notation for the sparse matrix. Thus our example becomes a graph where nodes are axioms, cuts and conclusions (terminal nodes):

$$V = \{[ax]_1, [ax]_2, [ax]_3, [ax]_4, [cut]_1, [t]_1\}$$



(a) sharing graph (b) proof net

**Fig. 1.** translations of the lambda term $(\Delta I)$

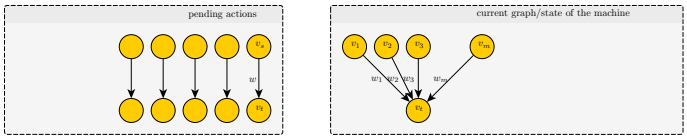
and edges $((v_t, v_s), w)$ get a weight $w \in \Lambda^*$ where $v_t$ is the target node and $v_s$ is the source node. In this example, the "sparse" representation, consisting of the list of edges with a non-null weight, is more compact:

$$E = \{(([cut]_1, [ax]_1), qx_1q), (([cut]_1, [ax]_1), qx_2), \\ (([cut]_1, [ax]_2), qx_1p), (([cut]_1, [ax]_2), p), (([cut]_1, [ax]_3), q!q), \\ (([cut]_1, [ax]_3), q!p), (([cut]_1, [ax]_4), p), (([t]_1, [ax]_4), 1)\}.$$

In what follows we denote the GoI interpretation of a lambda term $M$ with $[M]$, thus in the example above $[(\Delta I)] = E$. For the sake of conciseness, we gave an example of GoI interpretation in the case of a very simple lambda term. To have a complete account of interpretation of lambda terms as proof nets and of proof nets as GoI graphs, we refer the reader to [14,6].

### 2.1 Half-combustion machine

Now, we present the machine which is crucial for the later introduction of parallelism, by giving a new formal setting to express DVR as a computing device. In a way already in the spirit of "virtual machines" we reformulate the processing unit from the point of view of a universal device which consumes a pipeline of elementary instructions, while producing further instructions to be processed.




The elementary step of computation consisting in performing a step of half-combustion by following the definition in terms of DVR is described in [13]: and it can be thought as the effect of an action performed by the machine when a pending edge arrives and has to be processed: the edge $\alpha = ((v_t, v_s), w)$ acts

on its context which is given by edges $\beta_1, \ldots, \beta_m$ insisting on the node $v_t$, with weights $w_1, \ldots, w_m$, as in Fig. 2a.

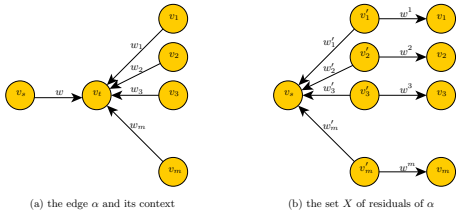
For any $\beta_i$ such that we get a stable form $a'b'^*$ of $b^*a$ different from 0, where $a'$ is the residual of $a = w$ and $b'$ is the residual of $b = w_i$, we obtain one corresponding pair of residual edges

$$\alpha_i = ((v_i, v_i'), a') \text{ and } \beta_i' = ((v_s, v_i'), b') \tag{1}$$

where $v_i'$ are new nodes to be added to $G$, as in Fig. 2b. The set of all residual edges originated by $\alpha$ is therefore the set

$$X \subset \{\alpha_1, \beta_1', \ldots, \alpha_m, \beta_m'\}. \tag{2}$$

Note that the number of residual pairs is possibly lower than the number of edges in the context. Indeed, for some of the $\beta_i$ there is no stable form.




(a) the edge $\alpha$ and its context (b) the set $X$ of residuals of $\alpha$

**Fig. 2.** Half-combustion: the action of the edge $\alpha$ on its context.

We have that the abstract machine performing the basic computational task can be described as the edge $\alpha$ acting on the graph $G$ which produces a modified graph $G'$ and a new computational task represented by the residuals $X$:

$$\alpha.G \to (X, G'). \tag{3}$$

In the next example, we show a few steps of DVR on the Example 1.

*Example 2.* We start with the empty graph $G_0 = (\varnothing, \varnothing)$ and the sequence of actions given by the GoI interpretation of the lambda term:

$$\begin{aligned}[(\Delta I)].G_0 = ((&[cut]_1, [ax]_1), qx_1q).((([cut]_1, [ax]_1), qx_2).\\ &.((([cut]_1, [ax]_2), qx_1p).((([cut]_1, [ax]_2), p).((([cut]_1, [ax]_3), q!q).\\ &.((([cut]_1, [ax]_3), q!p).((([cut]_1, [ax]_4), p).(([[t]_1, [ax]_4), 1).G_0\end{aligned}$$

After the first actions are consumed without giving rise to residuals, we get a reduced sequence of actions and the expanded graph:

$$\begin{aligned}&((([cut]_1, [ax]_1), qx_1q).((([cut]_1, [ax]_1), qx_2).\\ &\qquad((([cut]_1, [ax]_2), qx_1p).((([cut]_1, [ax]_2), p).G_4\end{aligned}$$

where $G_4 = (V_4, E_4)$ with $V_4 = \{[t]_1, [ax]_4, [cut]_1, [ax]_2\}$ and

$$E_4 = \{(([t]_1, [ax]_4), 1), (([cut]_1, [ax]_4), p), \\ (([cut]_1, [ax]_3), q!p), (([cut]_1, [ax]_3), q!q)\}.$$

Then, we execute $(([cut]_1, [ax]_2), p).G_4$, that is the product of its weight $p$ against $q!p$, $q!q$ and $p$: in the first two cases (by following the rules in $\Lambda^*$) we have no residuals, but, in the third case, since $p^*p = 1$, we have two residuals with weight 1 for both of them, common source node, a new one $v_0$ and pointing to $[ax]_2$ and to $[ax]_4$. Therefore, $X = \{\alpha', \beta'\}$ where $\beta' = (([ax]_4, v_0), 1)$ and $\alpha' = (([ax]_2, v_0), 1)$.

The sequence of actions $X$ is concatenated to the actions that have to be applied to the graph: so that the updated situation is

$$(([ax]_4, v_0), 1).(([ax]_2, v_0), 1). \\ (([cut]_1, [ax]_1), qx_1q).(([cut]_1, [ax]_1), qx_2).(([cut]_1, [ax]_2), qx_1p).G_5$$

where $G_5 = (V_5, E_5)$ with $V_5 = V_4 \cup \{[ax]_2\}$ and $E_5 = E_4 \cup \{(([cut]_1, [ax]_2), p)\}$.

This shows that in general we have a sequence $S$ of pending actions insisting on a graph $G$ and we get the first action $\alpha$ of $S$ and if $\alpha.G = (X, G')$ then we perform the transition $S.G \overset{HC}{\rightarrow} X.(S'.G')$ where $S = \alpha :: S'$.

In order to process in parallel we need a further setting for performing infinite computations on infinite sequences. We will show that the data structure representing streams provide effective means to express the GoI machine implemented in [13] in terms of an abstract machine on streams of elementary actions [15]. This machine will be exploited in next sections to formally express and study the parallel algorithm behind PELCR. An action can be viewed as the atomic amount of information needed to perform a step of half-combustion, namely:

- the *polarity* of the edge: any edge in the graph represents a half of a straight path. Therefore, we know from [13, §3] that all the paths incident on the same node form two orthogonal sets, such that residuals of orthogonal paths are still orthogonal. We use this property to reduce the number of null products by assigning an initial positive or negative polarity to edges and propagating this information to residuals during execution;
- its *topological* description (as a pair of nodes): any edge is specified by a pair of nodes whose addresses are given in a global virtual format. The two nodes give the routing information towards the processing units which hosts the physical memory of the target node containing the context information for the action. The source node of the edge is also an information on the processing unit hosting the node which is used as target node of the residual edges produced by a DVR step;
- its *algebraic* description (as a weight in the dynamic algebra, $\Lambda^*$): the weight is taken in Girard's dynamic algebra.

Polarization induces a bi-partition of edges co-inciding on the same node $v$ in two sets of edges with the same polarity, we denote the two sets by $v^+$ and $v^-$ accordingly to their respective polarities.

Note that polarities are assigned to be opposite in corresponding pair of residual edges choosing any possible orientation for the new source node $v_i'$ in Equation 1:

$$\alpha_i = ((v_i, (v_i')^{\epsilon}), a') \text{ and } \beta_i' = ((v_s, (v_i')^{-\epsilon}), b').$$

*Example 3.* The edges of Example 1 can be decorated with polarities:

$$E = \{(([cut]_1, [ax]_1{}^-)^-, qx_1q), (([cut]_1, [ax]_1{}^+)^-, qx_2),$$
$$(([cut]_1, [ax]_2{}^-)^-, qx_1p), (([cut]_1, [ax]_2{}^+)^-, p), (([cut]_1, [ax]_3{}^-)^+, q!q),$$
$$(([cut]_1, [ax]_3{}^+)^+, q!p), (([cut]_1, [ax]_4{}^-)^+, p), (([t]_1, [ax]_4{}^+), 1)\}.$$

### 2.2 Actions

Now let us define the set of actions employed in the stream oriented virtual machine corresponding to PELCR implementation. The idea is that a partial graph can be described as follows:

**Definition 3 (PELCR Actions).** *Given a dynamic graph $G$, which is a graph $G = (V, E \subset V \times V)$ with edges labeled on Girard's dynamic algebra $\Lambda^*$, we define an action $\alpha$ on $G$ as $\langle \epsilon, e, w \rangle$ where $\epsilon \in \{+, -\}$, $e = (v_t, v_s)$ is a pair of nodes in $G$ and $w \in \Lambda^*$.*

We define the abstract machine performing DVR, starting from the basic computational task $\alpha.G \to (X, G')$, as in Equation 3. Our description is reminiscent of the usual definition of SECD machines employed to give the operational semantics of $\lambda$-calculus. The result $\alpha.G$ of the execution of the action $\alpha$ on the dynamic graph $G$ is a set $X$ and an updated dynamic graph $G'$. Residual actions are then applied to the graph. In [6], the theory of virtual reductions was introduced to optimize the execution order: the resulting calculus was a local and asynchronous way to compute the GoI execution formula. That calculus was in line with the theory of interaction nets and a strong confluence property was showed: the algebraic modification was sufficient to keep coherent the computation. Since DVR is obtained as a special case of VR, we get the same strong local confluence and after one step of computation, the generated residuals $X$ can be applied in any order to the graph, without affecting the result.

This fact is at the origin of the idea that the computational device can be easily parallelised, and therefore it can be viewed at the origin of the implementation of PELCR itself [5,13]. In other words, we have that for any permutation $\sigma$ of $m$ indices, if

$$\alpha_m.\alpha_{m-1}.\ldots.\alpha_1.G_0 \to \beta_1.\ldots.\beta_n.G_m$$

then there exists a permutation $\tau$ such that

$$\alpha_{\sigma_m}.\alpha_{\sigma_{m-1}}.\ldots.\alpha_{\sigma_1}.G_0 \to \beta_{\tau_1}.\ldots.\beta_{\tau_n}.G_m.$$

*Remark 1.* Since in untyped $\lambda$-calculus a (normalizing) term may not have a finite execution, in order to design a machine that can evaluate terms in parallel (with two or more computational units exchanging data) we need to be able to cope with an infinite output.

We therefore introduce the streams to model possibly infinite inputs.

### 2.3 Streams

Let $A$ be any set. We avoid any assumption on $A$ in this section. Yet in what follows we use streams to distribute the computational load on many devices and this means that we need to define streams of *actions*. So, we have to look at $A$ as the set of all possible actions the computational device can perform. For the ease of exposition of the execution equivalence results given in Section 4, we consider $\mathbf{A}$ as the set of formal sums of elements of $A$, in particular a null element (the empty sum) $\mathbf{0}$, such that $\mathbf{0} + \alpha = \alpha$.

**Definition 4.** *A* stream *$S$ on $A$ is a sequence of elements of $A$. The set of all streams is denoted by $A^\omega$. The element-wise sum $S+T$ of two streams $S$ and $T$ is given by the equation:*

$$(S+T)(i) = S(i) + T(i), \qquad i = 0, 1, \ldots \tag{4}$$

For any stream $S$ we consider also the *shift* operation (also called *derivative*, or *tail*) where we have

$$\mathtt{shift}(S)(i) = S(i+1) \qquad i = 0, 1, \ldots \tag{5}$$

Thus, with no further ado, it follows that the stream $\mathsf{nil}$ is the neutral element of the sum of streams. Another useful operation on streams is the addition of an element as initial action of the stream:

**Definition 5 (Append).** *For any $\alpha \in \mathbf{A}$ and any stream $S$ on $A$, we define the stream $\alpha :: S$ by the following equation:*

$$(\alpha :: S)(0) = \alpha, \qquad (\alpha :: S)(i) = S(i-1) \qquad i = 1, 2, \ldots \tag{6}$$

**Definition 6 (Zip).** *For any pair of streams $S$ and $T$, we define the* zip $\ltimes$ *the stream obtained by intertwining of elements of the two streams:*

$$(S \ltimes T)(i) = \begin{cases} S(i/2) & \text{if } i \pmod 2 \equiv 0 \\ T((i-1)/2) & \text{if } i \pmod 2 \equiv 1 \end{cases}. \tag{7}$$

Note that zip is not commutative.

**Definition 7.** *A* weak-bisimulation *on $A$ is a relation $\rho \subset \mathbf{A}^\omega \times \mathbf{A}^\omega$ such that, for all streams $S$ and $T$ on $A$, if $(S,T) \in \rho$ then one of the following holds:*

$$S(0) = T(0) \text{ and } (\mathtt{shift}(S), \mathtt{shift}(T)) \in \rho; \tag{8a}$$

$$S(0) = \mathbf{0} \text{ and } (\mathtt{shift}(S), T) \in \rho; \tag{8b}$$

$$T(0) = \mathbf{0} \text{ and } (S, \mathtt{shift}(T)) \in \rho. \tag{8c}$$

**Definition 8.** *Two streams $S$ and $T$ defined on $A$ are* weakly-bisimilar*, denoted $S \approx T$ if there exists a weak-bisimulation $\rho$ such that $S \rho T$.*

### 2.4 Sequential Abstract Machine

In a way similar to that of classical SECD machines we define the state of the machine in terms of four components:

- a *stack* $S$, which is used to store the current action. In fact it can be the *empty stack*, $\langle\rangle$, or it can contain the next action acting on the graph $\langle\alpha\rangle$;
- an *environment* $E$, is a node of the graph and it provides the local environment where the current action has to be performed;
- a *control* $C$ is the stream of all actions either provided as initial input or created during the execution of other actions, it has to be executed in the context of the graph stored in the memory of the machine;
- a *dump* $D$ corresponds to the current graph and represents the global environment for all future actions.

The operations of this SECD machine may be defined by a state transition function $\tau$. We denote the empty dump (resp. a new/uninitialized node) with $\varnothing$ (resp. `NULL`).So, let us assume the machine in the state $(S, E, C, D)$. Then, we obtain the new configuration by applying transition $(S, E, C, D) \overset{\tau}{\mapsto} (S', E', C', D')$ where the possible cases are:

0. The `read()` operation returns a stream of PELCR actions; typically `read()` returns the GoI interpretation $[M]$ of a $\lambda$-term $M$; the initialisation is then

$$(\langle\rangle, \texttt{NULL}, \mathsf{nil}, \varnothing) \overset{\tau}{\mapsto} (\langle\rangle, \texttt{NULL}, \texttt{read}(), \varnothing).$$

   Nevertheless, we can initialise even with an "incorrect" stream provided by the `read()` and processed by the machine, with maybe unpredictable results.
1. If the stack is empty:

$$(\langle\rangle, \texttt{NULL}, \alpha :: C', D) \overset{\tau}{\mapsto} \begin{cases} (\langle\alpha\rangle, \texttt{NULL}, C', D) & \text{if } \alpha \neq \mathbf{0}, \\ (\langle\rangle, \texttt{NULL}, C', D) & \text{if } \alpha = \mathbf{0}. \end{cases}$$

2. If the stack is $\langle\alpha\rangle$, where $\alpha = \langle\epsilon, (v_t, v_s), w\rangle$ and the environment $v_t$ is missing, then

$$(\langle\alpha\rangle, \texttt{NULL}, C, D) \overset{\tau}{\mapsto} (\langle\alpha\rangle, v_t, C, D')$$

   where

$$D' = \begin{cases} D & \text{if } v_t \text{ already is a node of } D, \\ D \cup \{v_t\} & \text{if } v_t \text{ is a new node to be added to } D. \end{cases}$$

3. If we have an action in the stack and the corresponding node in the environment, we have

$$(\langle\alpha\rangle, v_t, C, D) \overset{\tau}{\mapsto} (\langle\rangle, \texttt{NULL}, C', D')$$

   where the dump $D' = D \cup ((v_t, v_s)^{\epsilon}, w)$ and the stream $C'$ is obtained combining $C$ with $X$, the set of residuals of the action $\alpha$ on its context $v_t^{-\epsilon}$:

$$C' = \begin{cases} C & \text{if } X = \varnothing \\ C \ltimes \mathsf{execute}(\alpha) & \text{if } X \neq \varnothing \end{cases}$$

where $\mathsf{execute}(\alpha)$ is defined as a finite stream obtained by concatenating in some order the residuals in $X$.

Note that this definition implies that the computation never terminates. We have a stream as an input to which we hook by means of an appropriate `read()` operation. The processed stream produces a new stream. The behaviour of the machine is the same whether the two streams are finite or not.

**Definition 9.** *Given an action $\alpha$, and the corresponding set of residuals $X$ (as in Equation (2)) we define the finite stream obtained rearranging actions in $X$ as* $\mathsf{execute}(\alpha) = \mathtt{streamset}(X)$*, where*

$$\mathtt{streamset}(X) = \begin{cases} \sigma :: \mathtt{streamset}(X \backslash \sigma) & \text{for some } \sigma \in X, \\ \mathsf{nil} & \text{if } X = \varnothing. \end{cases}$$

*Remark 2.* This non deterministic definition stems from one of the main features of local and asynchronous execution introduced in virtual reductions: parallel implementation can get rid of the typical confluence and synchronisation difficulties in distributed systems since it is the algebraic machinery that ensures the correctness of the computation.

## 3 Parallel Abstract Machine

We draw a distinction between synchronous and asynchronous parallel machines. In the first case the computing units perform a step of computation at the same time, the machines are clock synchronised and the computation proceeds on all machines. In the asynchronous computation one machine can perform many steps of computation while other machines perform one single step. In this second case we consider a scheduler which decides which is the current unit.

### 3.1 Synchronous Case

A parallel machine is given by $k$ computing units structured as a vector of $k$ distinct SECD structures. For the sake of simplicity, but without loss of generality, we define the basic case with $k = 2$. The state of the machine is therefore represented by $(S, E, C, D) = (S_1 \otimes S_2, E_1 \otimes E_2, C_1 \otimes C_2, D_1 \otimes D_2)$. The synchronous model of execution makes the machine step into the computational cycle in a synchronous way:

0. Initialization step reading the input stream is performed on the first unit:

$$(\langle\rangle \otimes \langle\rangle, \mathtt{NULL} \otimes \mathtt{NULL}, \mathsf{nil} \otimes \mathsf{nil}, \varnothing \otimes \varnothing) \overset{\tau}{\mapsto} (\langle\rangle \otimes \langle\rangle, \mathtt{NULL} \otimes \mathtt{NULL}, \mathtt{read}() \otimes \mathsf{nil}, \varnothing \otimes \varnothing).$$

1. Retrieve actions from the streams:

$$(\langle\rangle \otimes \langle\rangle, \mathtt{NULL} \otimes \mathtt{NULL}, \alpha_1 :: C_1' \otimes \alpha_2 :: C_2', D_1 \otimes D_2) \overset{\tau}{\mapsto} (\langle\alpha_1\rangle \otimes \langle\alpha_2\rangle, \mathtt{NULL} \otimes \mathtt{NULL}, C_1' \otimes C_2', D_1 \otimes D_2)$$

2. Prepare the two environments for the actions in the stacks:

$$(\langle\alpha_1\rangle \otimes \langle\alpha_2\rangle, \texttt{NULL} \otimes \texttt{NULL}, C_1 \otimes C_2, D_1 \otimes D_2) \overset{\tau}{\mapsto} \\ \overset{\tau}{\mapsto} (\langle\alpha_1\rangle \otimes \langle\alpha_2\rangle, v_1 \otimes v_2, C_1 \otimes C_2, D_1' \otimes D_2')$$

where

$$v_i = \begin{cases} v_t^i & \text{if } \alpha_i = \langle \epsilon_i, e_i, w_i \rangle \text{ and } e_i = (v_t^i, v_s^i) \\ \texttt{NULL} & \text{if } \alpha_i = \mathbf{0} \end{cases}$$

and

$$D_i' = \begin{cases} D_i & \text{if } v_t^i \text{ already is a node of } D^i, \\ D_i \cup \{v_t^i\} & \text{if } v_t^i \text{ is a new node to be added to } D^i. \end{cases}$$

3. Let actions act on the state:

$$(\langle\alpha_1\rangle \otimes \langle\alpha_2\rangle, v_t^1 \otimes v_t^2, C_1 \otimes C_2, D_1 \otimes D_2) \overset{\tau}{\mapsto} \\ (\langle\rangle \otimes \langle\rangle, \texttt{NULL} \otimes \texttt{NULL}, ((C_1 \ltimes \mathsf{execute}_1(\alpha_1)) \ltimes \mathsf{execute}_1(\alpha_2)) \otimes \\ \otimes ((C_2 \ltimes \mathsf{execute}_2(\alpha_1)) \ltimes \mathsf{execute}_2(\alpha_2)), D_1' \otimes D_2')$$

where $\mathsf{execute}_i(\alpha_j)$ is the stream obtained by selecting from $\mathsf{execute}(\alpha_j)$ the residual actions with the target node allocated on the computing unit $i$. If $\alpha_i = \mathbf{0}$, then the $\mathsf{execute}(\mathbf{0})$ is undefined and therefore it is not added to the corresponding $C_i$. When the set of residuals is empty $\mathsf{execute}(\alpha)$ is the stream $\mathsf{nil}$ and it is not added to the corresponding $C_i$. The graph $D_i$ is then transformed by adding the edge $((v_t^i, v_s^i)^{\epsilon_i}), w_i)$ to the graph $D_i$.

*Remark 3.* 1. The computation of the stream $\mathsf{execute}_i(\alpha_j)$ is performed by the $j$-th computing unit, but the stream is zipped to the stream $C_i$ on the $i$-th computing unit: this leads to the communication of residual actions towards their respective computing units.
2. Residuals are computed on the $i$-th computing unit performing the action $\alpha_i$ in the context $v_t^i \in D_i$. They are new actions containing edges with target node $v_s^i$ and with source node a newly created node $v = \texttt{new}()$. Note that both nodes can belong to either $D_1$ or $D_2$. The new node $v$ can be allocated to any computing unit depending on the chosen load balancing strategy, whereas $v_s^i$ is hosted by the unit decided once it was created as a source node of some residual action.

### 3.2 Asynchronous Case

In the asynchronous case one deals with modeling the behaviour of the parallel machine when the execution steps are not performed at the same time on all computing units. This model is realised through an asynchronous scheduling mode which rules the order of execution. Like in the synchronous case, it suffices to give the definition in the case of two computational units.

Let us consider a parallel machine with two computing units (the general case, with $k$ units, is a direct extension of this one), whose state is therefore represented by $(p, S, E, C, D) = (p, S_1 \otimes S_2, E_1 \otimes E_2, C_1 \otimes C_2, D_1 \otimes D_2)$ where $p \in \{1, 2\}$. The asynchronous model of execution makes the machine to step in the computational cycle in an independent way. We add a control $p$ which represents the computing unit which has to perform the next computational step. Then the machine by itself has the following update rules, where the next scheduled unit $p'$ is randomly chosen with uniform probability in $\{1, 2\}$:

0. Initialization step reading the input stream is performed on the first unit:

$$(1, \langle\rangle \otimes \langle\rangle, \texttt{NULL} \otimes \texttt{NULL}, \mathsf{nil} \otimes \mathsf{nil}, \varnothing \otimes \varnothing) \overset{\tau}{\mapsto} \\ \overset{\tau}{\mapsto} (p', \langle\rangle \otimes \langle\rangle, \texttt{NULL} \otimes \texttt{NULL}, \texttt{read}() \otimes \mathsf{nil}, \varnothing \otimes \varnothing)$$

1. If the current stack is empty ($S_p = \langle\rangle$), the current environment $E_p$ is $\texttt{NULL}$, and the current stream is $C_p = \alpha_p :: C'_p$, then

$$(p, S, E, C, D) \overset{\tau}{\mapsto} (p', S'_1 \otimes S'_2, E'_1 \otimes E'_2, C'_1 \otimes C'_2, D_1 \otimes D_2)$$

where $S'_i = \begin{cases} S_i & \text{if } i \neq p \\ \langle \alpha_i \rangle & \text{if } i = p \end{cases}$, $\quad E'_i = E_i$, $C'_i = C_i$ if $i \neq p$.

2. If an action is in the current stack ($S_p = \langle \alpha_p \rangle$) and the corresponding environment is not yet retrieved ($E_p = \texttt{NULL}$) then we distinguish two cases:
   (a) if we got a null action in the current stack $\alpha_p = \mathbf{0}$ then we

$$(p, S, E, C, D) \overset{\tau}{\mapsto} (p', S'_1 \otimes S'_2, E_1 \otimes E_2, C_1 \otimes C_2, D_1 \otimes D_2)$$

   where $S'_p = \langle\rangle$ and we leave the rest unchanged.
   (b) in the second case if $\alpha_p = \langle \epsilon_p, e_p, w_p \rangle$, then the transition is

$$(p, S, E, C, D) \overset{\tau}{\mapsto} (p', S'_1 \otimes S'_2, E'_1 \otimes E'_2, C'_1 \otimes C'_2, D'_1 \otimes D'_2)$$

where $E'_i = \begin{cases} E_i & \text{if } i \neq p \\ v_t^p & \text{if } i = p \end{cases}$ $S'_i = S_i$, $C'_i = C_i$ and

$$D'_i = \begin{cases} D_i & \text{if } i \neq p \text{ or } i = p \text{ and } v_t^p \text{ already is a node of } D_p, \\ D_i \cup \{v_t^p\} & \text{if } i = p \text{ and } v_t^p \text{ is a new node to be added to } D_p. \end{cases}$$

3. Finally, if the stack contains a non-null action $S_p = \langle \alpha \rangle_p$, $\alpha_p = \langle \epsilon_p, e_p, w_p \rangle$ and the environment is retrieved $E_p = v_t^p$ then

$$(p, S, E, C, D) \overset{\tau}{\mapsto} (p', S'_1 \otimes S'_2, E'_1 \otimes E'_2, C'_1 \otimes C'_2, D'_1 \otimes D'_2)$$

where

$$S'_i = \begin{cases} S_i & \text{if } i \neq p \\ \mathbf{0} & \text{if } i = p \end{cases} \qquad E'_i = \begin{cases} E_i & \text{if } i \neq p \\ \texttt{NULL} & \text{if } i = p \end{cases}$$

$$C'_i = C_i \ltimes \mathsf{execute}_i(\alpha_p)$$

and the graph $D'_i = D_i$ for all $i \neq p$ while $D'_p$ is obtained from $D_p$ by adding the edge $((v^p_t, v^p_s)^{\epsilon_p}, w_p)$. Note that like in previous cases if the set of residual actions to be addressed to the stream $C_i$ is empty the zip is not considered and we put $C'_i = C_i$.

The sequence of controls $p$ is by itself a stream (of integers $\{1, 2\}$). In place of a random sequence we may force a particular scheduling by fixing the sequence.

*Remark 4.* By choosing the scheduling constant to 1, we have the sequential machine. Moreover, if we fix the round-robin scheduling $1, 2, 1, 2, \ldots$ which alternates the computing unit 1 with the second one, we have the correspondence with the synchronous model.

## 4 Execution equivalence

In this section we sketch the soundness of the parallel computation with respect to the sequential one. We assume the soundness of the sequential machine by definition of DVR (`hc-vm` is sound with respect to computing of the execution formula of lambda terms) and we get the soundness of the parallel version by showing that for any input stream obtained by the `read()` operation at step 0 of the parallel and of the sequential machines, we have the same sequence of computational steps, executed by both the machines (up to zero steps or reordering of residuals of computational steps).

Let us introduce the notion of *node-view* (or *view of base* $v$) of a stream of actions $S$:

**Definition 10.** *Given a stream of actions $S$ and a node $v$, the polarised view of base $v^\epsilon$ is defined by selecting actions with target node $v$ and opposite polarity with respect to the polarity of the base. Namely:*

$$S_{v^\epsilon} = \begin{cases} \mathbf{0} & \text{if } S = \mathbf{0} \\ S(0) :: (\mathtt{shift}(S))_{v^\epsilon} & \text{if } S(0) = \langle -\epsilon, (v, v_s), w \rangle \\ (\mathtt{shift}(S))_{v^\epsilon} & \text{otherwise} \end{cases}$$

*We define also the view of base $v$ as $S_v = S_{v^+} \ltimes S_{v^-}$.*

We have a notion of equivalence of the states of two machines

**Definition 11.** *Let $A = (S_1, E_1, C_1, D_1)$ (resp. $B = (S_2, E_2, C_2, D_2)$) a state of a machine $M_1$ (resp. $M_2$), we say that the state $A$ is equivalent to the state $B$ (let us denote this* state equivalence *by $A \simeq_\sigma B$) whenever*

1. *$D_1$ and $D_2$ are isomorphic graphs, i.e., we have an isomorphism of graphs $\phi : D_1 \to D_2$, and*
2. *for any node $v \in D_1$ we have equivalent views on the controls (the two streams of actions) when taking $v$ and its corresponding node $\phi(v)$, $(C_1)_v \approx (C_2)_{\phi(v)}$, and*

*3. the current action $S_1$ is isomorphic with $S_2$ (*i.e.*, , $S_2 = \phi(S_1)$).*

**Lemma 1.** *$\simeq_\sigma$ is an equivalence relation.*

*Proof.* Trivial: $\simeq_\sigma$ is the intersection of two equivalence relations.

Note that the view of base $v$ of multiple streams (like in the case of parallel machines) is a stream since it results that the base $v$ possibly belongs to at most one of the dumped graphs. Another important fact is that:

**Lemma 2.** *Any action on the polarised view of base $v^\epsilon$, $S_{v^\epsilon}$ is originated by actions acting on the same node.*

The lemma is a consequence of the Definition 10. Nevertheless, this property is not sufficient to guarantee the state equivalence of the resulting graph after the execution of a single action.

A node in the dumped graph whitout incident actions is called a *ghost* node [13, §3]. Edges pointing to ghost nodes are ghost edges. Let the *valence* of a node be the number of non ghost edges exiting this node.

**Definition 12.** *The combustion strategy [5, §4.1] chooses a node of valence 0 and performs all the possible actions on that node.*

On the other hand the correctness of the execution is guaranteed by two facts:

1. full combustion of a node is correct because it is equivalent to a particular synchronisation strategy of directed virtual reduction;
2. half-combustion is a less synchronous version of full-combustion, therefore it permits to execute actions in a different order, by choosing (zero-valence) nodes in any order.

We provide here an analog proof of correctness in the setting of abstract machines. First, we define a particular synchronisation of the sequential machine: this synchronisation gives us a machine which implements full combustion. Then, we show that the machine implements an asynchronous version of full combustion.

To introduce full combustion we need to consider nodes which appear as target node in actions, while not appearing in the dumped graph: we have that residuals of any action have, by definition, fresh source nodes in the graph, and these nodes are added to the dumped graph only when a (first) action is executed on that node. We call such a node (existing as a target node of actions but not in the dumped graph) *s-node* (spiritual node); on the other hand, since no action can occur on a *ghost* node, it can be safely removed from the dumped graph $D$ (which is not used in this version of the machine). The full combustion (cf. Proposition 12) processes all the actions in a view of base $v$ before focusing on another node.

**Definition 13.** *Let us denote by $v.M$ the* full action *of $v$ on the machine $M$ as follows:*

1. *for some $v$ let us consider the view of base $v$, and let us suppose that $C_v \neq \mathbf{0}$, and reordering of actions $\sigma$*

$$(\mathbf{0}, \texttt{NULL}, \sigma(C' \ltimes C_v), D) \overset{\tau}{\mapsto} (C_v, (\{v\}, \varnothing), C', D),$$

2. *then repeat the following step until $S'$ is the empty stream:*

$$(\alpha :: S', E, C, D) \overset{\tau}{\mapsto} (S', (\{v\}, Y \cup \{((v, v_s)^{\epsilon}, w)\}), C \ltimes \mathsf{execute}(\alpha), D)$$

*where $E = (\{v\}, Y)$, $\alpha = \langle \epsilon, e, w \rangle$ and the edge is $e = (v, v_s)$.*

By means of full combustion we are in position to prove that full combustion implemented in the parallel case is equivalent to sequential full combustion:

**Theorem 1.** *Given a (sequential) machine $M_1$ and a (parallel) machine $M_2$ such that $M_1 \simeq_\sigma M_2$ by the isomorphism $\phi$, then we have that $v.M_1 \simeq_\sigma \phi(v).M_2$.*

*Proof.* If we consider an $s$-node $v$, and we compute the residuals of the view of base $v$, i.e. residuals obtained by processing actions in $S_v$, then we get the complete set of residuals relative to the node $v$, indicated with $X_v = \bigcup_{\alpha \in S_v} \mathsf{execute}(\alpha)$; it is worth noting that after this step of computation $v$ becomes a ghost node. The set $X_v$ contains residuals for any pair of actions with opposite polarities which appears in $S_v$: the order of execution is immaterial since any pair is used exactly once, and any single action is unaffected after the use. What remains modified by the ordering of the steps of computation is the way used to mix the streams obtained by the residuals of any action with the total stream of actions. These residuals, in fact, modify the views with base in the source nodes appearing in the performed actions.

## 5 Conclusions

We have outlined a stream-based description of PELCR, thus highlighting the message interchange mechanism at the base of the parallel executions of terms with PELCR. Although there exists implementations of functional languages which are generally more efficient in the sequential case, PELCR can execute those jobs whose huge size is made impossible to handle on sequential machines. Parallel implementations of optimal reductions are tricky, insofar as without optimization they are not particularly efficient. Moreover, most of the significant optimizations only work in the sequential case, like in Asperti's implementation [2] (cf. §2), based on safe operators, which employs a *sequential* safe-tagging algorithm. PELCR's ability to *dynamically* distribute the workload among the available processors exposes *intrinsic* parallelism of programs at hand (thus requiring no annotation from the programmer). Starting from this work, we plan to conduct a quantitative analysis of the behaviour of PELCR when executed on parallel and distributed architectures.

## References


1. Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In *Proceedings of The 19th ACM SIGPLAN International Conference on Functional Programming*, 2014.
2. Andrea Asperti and Juliusz Chroboczek. Safe operators: Brackets closed forever optimizing optimal lambda-calculus implementations. *Appl. Algebra Eng. Commun. Comput.*, 8(6):437–468, 1997.
3. Andrea Asperti, Cecilia Giovanetti, and Andrea Naletto. The Bologna Optimal Higher-order Machine. *Journal of Functional Programming*, 6(6):763–810, 1996.
4. V. Danos and L. Regnier. Proof-nets and the hilbert space. In *Advances in Linear Logic*, pages 307–328. Cambridge University Press, 1995.
5. Vincent Danos, Marco Pedicini, and Laurent Regnier. Directed virtual reductions. In *Computer science logic (Utrecht, 1996)*, volume 1258 of *Lecture Notes in Comput. Sci.*, pages 76–88. Springer, Berlin, 1997.
6. Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of Girard's execution formula). In *Proceedings of the Eighth Annual IEEE Symposium on Logic in Computer Science (LICS 1993)*, pages 296–306. IEEE Computer Society Press, June 1993.
7. Jean-Yves Girard. Geometry of interaction I. Interpretation of system **F**. In *Logic Colloquium '88 (Padova, 1988)*, volume 127 of *Stud. Logic Found. Math.*, pages 221–260. North-Holland, Amsterdam, 1989.
8. Jean-Yves Girard. Geometry of interaction II. Deadlock-free algorithms. In *COLOG-88 (Tallinn, 1988)*, volume 417 of *Lecture Notes in Comput. Sci.*, pages 76–93. Springer, Berlin, 1990.
9. Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 15–26, Albequerque, New Mexico, 1992.
10. J. Roger Hindley and Jonathan P. Seldin. *Introduction to combinators and $\lambda$-calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge, 1986.
11. Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, January 1964.
12. Ian Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–208. ACM, 1995.
13. Marco Pedicini and Francesco Quaglia. PELCR: parallel environment for optimal lambda-calculus reduction. *ACM Trans. Comput. Log.*, 8(3):Art. 14, 36, 2007.
14. Laurent Regnier. *Lambda-calcul et réseaux*. PhD thesis, Paris 7, 1992.
15. J.J.M.M. Rutten. A tutorial on coinductive stream calculus and signal flow graphs. *Theoretical Computer Science*, 343(3):443 – 481, 2005.
16. Leslie G. Valiant. A bridging model for multi-core computing. *J. Comput. System Sci.*, 77(1):154–166, 2011.