Under consideration for publication in Math. Struct. in Comp. Science

Abstract Machines, Optimal Reduction, and Streams

Anna Chiara Lai†, Marco Pedicini†, Mario Piazza‡

†*Department of Mathematics and Physics, Roma Tre University,*

Via della Vasca Navale 84, 00146 Roma, Italy,

‡*Department of Philosophical, Pedagogical and Economic-Quantitative Sciences, University of Chieti-Pescara,*

Via dei Vestini 31, 66013 Chieti, Italy

Received 10 March 2017

In this paper, we explore a new approach to abstract machines and optimal reduction via streams, infinite sequences of elements. 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 the λ-calculus interpreter called *Parallel Environment for Lambda Calculus Reduction* (PELCR), a software for λ-calculus reduction 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

In the 1960s Peter Landin introduced the *Stack, Environment, Core and Dump* machine (SECD), the first abstract machine for the λ-calculus (Landin, 1964). Since then, abstract machines describing the implementations of functional languages have been conceived of as bridges between a high-level language and a low-level architecture (Hindley and Seldin, 1986; Curien, 1990; Fairbairn and Wray, 1987; Cousineau and Mauny, 1998; Accattoli et al., 2014). From the vantage point of logic, it is well-know that the Curry-Howard isomorphism guarantees a direct correspondence between typed λ-calculus and constructive logic, so that concepts like λ-terms and formal proofs turn out to be different representations of the same mathematical objects. Namely, cut-elimination on proofs may be regarded as identical to β-reduction on λ-expressions, allowing for the mathematical description of abstract machines as executions of programs. In particular, some abstract machines (Asperti et al., 1996) have been proposed as a tool for studying the theory and implementation of optimal reduction of the λ-calculus (Lévy, 1978; Lévy, 1980; Lamping, 1989) (see also (Asperti and Guerrini, 1998) for an overview of the topic). Other abstract machines (Pedicini, 1998; Mackie, 1995; Pinto, 2001) are based on the Geometry of Interaction (GoI), a mathematical framework developed by J.Y. Girard to provide

a semantical view of linear logic as well as to model the dynamics of cut-elimination (Girard, 1989; Gonthier et al., 1992).

In this paper, we explore a new approach to abstract machines and optimal reduction through *streams* – infinite sequences of elements – which are ubiquitous in mathematics and computer science (see for instance (Rutten, 2005a) and (Rutten, 2005b)). The main goal is to introduce a mathematical model of computation oriented to the quantitative analysis and the optimization of machines performing optimal reduction on parallel architectures. To this end, we begin by designing sequential abstract machine whose dynamic of execution relies on the algebraic properties of *dynamic monoids*; the sequential execution is then extended to some degree of parallelisation. Finally, we restrict our investigation to computations based on GoI to prove the soundness of parallel execution with respect to the sequential case.

More precisely, we recall that a *virtual reduction* (Danos and Regnier, 1993) is a fine-grained way to do optimal reduction based on Girard's dynamic algebra Λ^*. Virtual Reduction (VR) hinges on 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. By keeping "algebraic trace" of the performed compositions to be stored on the current graph, VR allows one to compute without useless (re)compositions. On the other hand, a *Directed Virtual Reduction* (DVR) (Danos et al., 1997) is a variant of VR which exploits the original algebraic machinery of GoI, by removing the added part of the algebra introduced in (Danos and Regnier, 1993), while managing to avoid re-compositions. The proposed sequential abstract machine is a generalization to arbitrary dynamic monoids of DVR.

When one considers the abstract machine performing DVR, the extension to the parallel implementation yields a formal definition of the PELCR (Parallel Environment for optimal Lambda-Calculus Reduction) engine, that is a parallel implementation of DVR described in (Pedicini and Quaglia, 2007) and available on the Web at `https://github.com/pis147879/PELCR`.

The style of parallelism of PELCR is similar to the Bulk Synchronous Parallelism (BSP) originally introduced in Valiant's paper (Valiant, 1990). In BSP, the computational load is divided on many processing elements alternating working on separate data sets with communicating and synchronising of results; actually, in PELCR this communication phase is not strictly synchronised, and there are no synchronisation barriers like in BSP approach. The notion of BSP has been applied in the parallel programming model known as Partitioned Global Address Space (PGAS) and included in the language X10 specified by IBM (Charles et al., 2005). As a remedy for the difficulties in automatic parallelisation, PGAS allows people to choose one proper parallel programming model (or a form of mixture of models) to develop their parallel applications on a particular platform. The programming style PGAS realises is a particular parallelism, oriented to the partitioning of data sets. On the other hand, PELCR can be regarded as an *ante litteram* implementation of a distributed memory model with a global addressing memory space for storing the current state of the computation. This state is here represented by a partially evaluated graph of the GoI representation of the initial λ-term. Global addressing of the memory

is obtained by using message passing of virtual addresses via the libraries for Message Passing Interface (MPI) available on many computer architectures.

To conclude, let us mention some of the advantages of our formal representation of PELCR's parallelism. First, it allows us to compare various models of parallel execution in a uniform setting. By providing an in-deep analysis of parallel execution on different models (Valiant, 2011), it permits us to assess quantitatively their differences, and could impact PELCR itself. Moreover, it provides a grammar to describe extensions of λ-calculus oriented toward parallel execution, and to quantify the efficiency of their evaluation strategies. It is also worth stressing that GoI is flexible enough to deal with resource sensitive calculi (Solieri, 2016) as well as with implicit computational complexity logical systems (Baillot and Pedicini, 2001). In (Canavese et al., 2015) the implementation of a software library of algebraic type in terms of implicit computational complexity combines a formal approach to complexity with a view of PELCR as the physical device for distributed execution of arithmetic functions.

This paper is organised as follows. In Section 2 we present the Sequential Abstract Machine, 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 in the case of machines performing DVR. Finally, Section 5 presents our conclusions.

Acknowledgments: A preliminary version of this paper was presented at the Symposium on Trends in Functional Programming 2014 as (Pedicini et al., 2014). The new presentation of similar contents is here improved and corrected by filling deficiencies in their presentation as well as by considering new cases in examples.

2. Sequential Abstract Machine

The processing unit of the abstract machine, which we are about to introduce, is a universal device which consumes a pipeline of elementary instructions, while producing further instructions to be processed. The memory of the machine is represented by a *dynamic graph*, i.e., a graph characterised by some algebraic properties of its labels, which are assumed to be taken in a *dynamic monoid* – see Definition 1 below. Moreover, also instructions to be executed are edges, and their execution consists of the transformation of the graph in memory and the production of a sequence of new instructions to be executed.

We report the pseudocode of the implementation PELCR, which is a parallel interpreter for λ-terms based on the GoI. In PELCR, many processing elements P_i cooperate to the evaluation of a single λ-term viewed as a dynamic graph. The single processing unit execution flow is sketched in Figure 1. Main evaluation loop consists in buffering instructions coming as messages from other processing units, then processing these instructions one by one by composing the edge carried by the message with any edge already hosted by the same target node, deciding where the new node originated by residuals of the composition must be allocated and sending residual edges to the processing units hosting their target nodes.

```
program P_i;
1  initialize();
2  while not end_computation do
3     ⟨collect all incoming messages and store them in incoming_i⟩
4     while not empty(incoming_i) do
5        ⟨extract a message m from incoming_i⟩;
6        if m.target ∈ nodes_i     'node already in the local list'
7        then
8           for each edge e ∈ nodes_i(m.target).combusted do
9              ⟨compose the edge carried by m with e⟩;
10             ⟨select the destination process P_j for hosting
                 the node originated by the composition⟩;
11             ⟨send the edges produced by the composition to
                 P_k and P_h hosting m.source and e.source
                 respectively⟩
12          endfor
13       else ⟨add m.target to nodes_i⟩;   'delayed creation'
14       ⟨add the edge carried by m to
         nodes_i(m.target).combusted⟩
15    endwhile;
16    ⟨end_computation = check_termination()⟩;
17 endwhile
```

Fig. 1. Pseudocode for a process P_i in PELCR as given in (Pedicini and Quaglia, 2007)

What we achieve in the first part of the paper is the description of a family of abstract machines whose functioning relies on some requirements satisfied by the PELCR evaluation machine. Moreover, these requirements can be used to design various evaluation models which in principle could be executed through our abstract machines.

On the one side, the fact that the defined machines can be used to perform DVR ensures their completeness with respect to computability; on the other side, these very machines can in turn be employed to define other computational models (see Examples 4 and 5). The formal definition of these abstract machines will be given in sections 2.4, 3.1 and 3.2. Before defining formally the algebraic version of the PELCR implementation from which to extract the basic notions for the machines (sections 2.1 and 2.2), it is useful to sketch hereunder an informal presentation.

Roughly speaking, instructions are (polarized labelled) edges called *actions*, that are added to the flow of control. An evaluation sequence as performed in the PELCR machine is represented by a flow of actions, one by one operating on the current state which is a graph. Thus, at any evaluation step we have a machine configuration (A, G) which is a pair, as reported in Figure 2:

— a list A of *pending actions* (on the left of the picture) and

— a (dynamic) graph G (as represented on the right part of the picture).

The abstract machine transition is obtained by applying an action α, taken from the sequence A, to the current graph G. In fact, α carries information required to transform the graph and this information coincides with an edge description: an elementary computational step is then performed by accessing the target node of the edge in G and by performing the computational payload expressed by the interaction of α with any edge in G having the same target node.

The result of this elementary computational step is composed of two parts

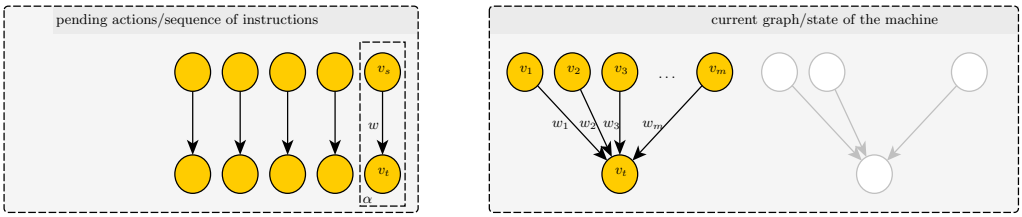


Fig. 2. Evaluation scheme

— a new set Δ_α of instructions to be executed called *residual actions* of α, which are to be added to pending actions;
— the updated graph $G \cup \{\alpha\}$ which now keeps track of the edge carried by α.

We build from the result $(\Delta_\alpha, G \cup \{\alpha\})$ of the elementary step of computation the corresponding transition

$$(A, G) \xrightarrow{\tau_\alpha} (A \setminus \{\alpha\} \cup \Delta_\alpha, G \cup \{\alpha\})$$

which transforms the configuration (A, G) to a new configuration where residual actions Δ_α are added to the list of pending actions A and α is removed from A and a corresponding edge is added to G.

The above transition is a rough description of what is defined as a step of half-combustion strategy of DVR (see (Pedicini and Quaglia, 2007)): it includes a symbolic computation in the algebraic structure associated to the graph (the dynamic monoid) and it is based on computations introduced in the GoI.

The memory of the machine is initialised with an empty graph, so that the execution of a terminating program on the abstract machine can be summarised by a finite sequence of transitions

$$(A^0, \emptyset) \xrightarrow{\tau_{\alpha_0}} (A^1, G^1) \xrightarrow{\tau_{\alpha_1}} \cdots \xrightarrow{\tau_{\alpha_{n-1}}} (A^n, G^n) \xrightarrow{\tau_{\alpha_n}} (\emptyset, G^{n+1}) \tag{1}$$

where $\alpha_i \in A^i$ for all $i = 0, \ldots, n$. Note that the initial action set A^0 is the interpretation of the program, and the final graph G^{n+1} represents the result of the evaluation. Following the notation for GoI, we say that the final graph G^{n+1} is the *Execution Formula* of the initial sequence A^0.

Now we focus on the essential traits of the computation as described above. To this aim, we rearrange concepts from GoI and DVR to shape an original presentation of an abstract algebraic setting for the machine: Section 2.1 deals with the formal definition of the algebraic structure concerning memory and instructions, Section 2.2 gives the elementary computational step associated to an instruction (half-combustion step), and Section 2.3 illustrates the algebraic version of the communication layers between components (the notion of stream of actions).

2.1. *The state of the machine: polarised dynamic graphs*

First of all we recall the definition of monoid and free monoid. A *monoid* is a set M closed under an associative binary operation $\cdot$, called the *product*, and containing an identity

element 1 (*i.e.*, $1 \cdot m = m \cdot 1 = m$, for any $m \in M$). The *free monoid* generated by a set A is the set A^* whose elements are all the finite sequences of zero or more elements from A, with sequence concatenation as the monoid operation.

Definition 1. A *dynamic* monoid on the alphabet A is the free monoid M generated by A such that

— there exists $0 \in M$ such that 0 is an absorbing element for product (*i.e.*, $0 \cdot m = m \cdot 0 = 0$, for any $m \in M$);
— 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 (Stable Form Condition).
Let M be a dynamic monoid. A non-zero element a of M is *positive* if it does not contain any inversion $*$. Let a, b be positive elements of M: we say that b^*a has a (or can be rewritten in) *stable form* if there exist non-zero $a', b' \in M$ (uniquely determined by a, b) such that $b^*a = a'b'^*$. We say that the dynamic monoid satisfies the *stable form condition* (SFC) if

$$\text{for any } a, b \in M \text{ either } b^*a = 0 \text{ or it has a stable form.} \qquad \text{(SFC)}$$

If SFC holds and it is computable (in linear time), it permits to perform computation by means of DVR. However, while a dynamic monoid satisfying SFC is enough to execute computations in the machines we present here, a special kind of dynamic monoid is required if we wish to have invariant properties (e.g., the invariance of normal form) given by the logical interpretation of programs. To this aim, we introduce the dynamic monoid employed by Girard in defining GoI for linear logic, a monoid which can be also applied to the interpretation of λ-calculus.

Definition 3 (Girard dynamic algebra Λ^*). The so-called *Girard dynamic algebra* Λ^* is the dynamic monoid generated by the constants p, q, and a family $W = \{w_i\}_i$ of exponential generators, with a morphism $!(.)$, such that for any $u \in \Lambda^*$:

$$x^*y = \delta_{xy} \qquad \text{for } x, y \in \{p, q, w_i\}, \qquad \text{(ANNIHILATION)}$$
$$!(u)w_i = w_i!^{e_i}(u), \qquad \text{(COMMUTATION)}$$

where δ_{xy} is the Kronecker operator, e_i is an integer associated with w_i called the *lift* of w_i, i is called the *name* of w_i and we often write w_{i,e_i} to explicitly note the lift of the generator.

Remark 1. The reader is referred to (Danos and Regnier, 1995). Note well that the morphism ! is indeed an endomorphism for the dynamic monoid, thus

$$!(uv) = (!u)(!v) \quad !(u^*) = (!u)^* \quad !1 = 1 \text{ and } !0 = 0 \qquad \text{for any } u \text{ and } v.$$

Notice that annhilation and commutation rules imply that for every $a, b \in \Lambda^*$ either $b^*a = 0$ or it has a stable form, that is Λ^* satisfies SFC. For instance, setting $a = w_{1,2}$ and $b = !^2q$, by applying the annihilation rule we get:

$$b^*a = (!^2q)^*w_{1,2} = !(!q^*)w_{1,2} = w_{1,2}!^2(!q^*) = w_{1,2}(!^3q)^* = a'b'^* \qquad (2)$$

with $a' = a$ and $b' = !b$.

Definition 4 (Dynamic graph, polarity). Given a dynamic monoid M satisfying SFC, a *dynamic graph* G on M is a graph $G = (V, E \subset V \times V \times M)$ with edges labelled on M.

A *polarised dynamic graph* on M is a dynamic graph G whose edges e are endowed with a *source polarity* $\varepsilon_s \in \{+, -\}$ and a *target polarity* $\varepsilon_t \in \{+, -\}$. More precisely, the edge set E is a subset of $\{+, -\}^2 \times V^2 \times M$ and every edge e is represented by a triplet $\langle(\varepsilon_t, \varepsilon_s), (v_t, v_s), w\rangle$, where: $\varepsilon_t, \varepsilon_s \in \{+, -\}$ are the target and source polarities of e, respectively; $v_t, v_s \in V$ are the target and source nodes of e, respectively; and $w \in M$ is the label of e.

Definition 5. For any dynamic monoid M we denote by $\mathbb{G}_M$ (resp. $\mathbb{G}^+_M$) the set of all (resp. polarised) dynamic graphs on M.

To point out the peculiar role of the edges with respect to the execution of the abstract machines, we define the *actions* as graph transformation instructions. Any action has as a payload information concerning an edge to be added to the current dynamic graph. We then compute composition with other edges via interaction rules which produce new instructions from residual edges.

Now we have to introduce the notion of reference to identify those nodes which have multiple occurrences in a sequence of graphs.

Definition 6. Given a sequence of graphs $G_i = (V_i, E_i)$, we call *reference* (to a node) an injective map ρ from the set of all nodes $\bigcup_i V_i$ to integers.

The set A_M of all possible *polarised actions* on M is a set of a graph edge specifications, more precisely:

$$A_M = \{\langle(\varepsilon_t, \varepsilon_s), (\rho(v_t), \rho(v_s)), w\rangle, \text{ where } \rho(v_t) \text{ and } \rho(v_s) \text{ are references to nodes of some graph in } \mathbb{G}^+_M, \varepsilon_t, \varepsilon_s \text{ are polarities, and } w \in M\}$$

Pending actions are sequences of instructions for transforming graphs: the core transformation any instruction represents, is the addition of nodes and edges to the graphs, as we show in Section 2.4. Actions can make reference to nodes which possibly are not in the current graph (for instance, consider the initial (empty) graph and the initial stream of pending actions). Therefore, to be more precise we have to say that an action has information on an edge, while nodes are expressed as references. We can apply the same strategy in presenting parallel abstract machines: in this case, the current graph is decomposed by allocating nodes in different processing units, and it can happen that one edge has source node on a unit and the target node on another. In this case edge information is hosted in the dynamic graph to which the target node belongs, whilst source node information is given as a reference to a node of the part of the dynamic graph hosted by the other unit. For the sake of simplicity, however, we use always the node notation to avoid to graphically distinguish between a node v and references to that node $\rho(v)$.

Notation 1. Polarization induces a bi-partition of edges co-inciding on the same node

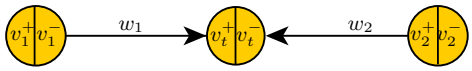


Fig. 3. A polarized graph.

v in two sets of edges with the same polarity. We denote the two sets by v^+ and v^- accordingly to their respective polarities.

Example 1. Consider the polarised dynamic graph $G \in \mathbb{G}_M^+$ depicted in Figure 3. We have $G = (V, E)$ where $V = \{v_1, v_2, v_t\}$. The edges of G are

$$\alpha_1 = \langle (+,-), (v_t, v_1), w_1 \rangle \quad \text{and} \quad \alpha_2 = \langle (-,+), (v_t, v_2), w_2 \rangle.$$

Finally we notice that the seminodes associated to v_t are the edge sets $v_t^+ = \{\alpha_1\}$ and $v_t^- = \{\alpha_2\}$.

Example 2 (Encoding λ-terms as inputs for the machine via GoI). We consider the pure λ-term representing the self application $\Delta = \lambda x.xx$ applied to the term $I = \lambda x.x$. In a quite standard way (Regnier, 1992) this term is translated in a linear logic proof net, represented in Figure 4.

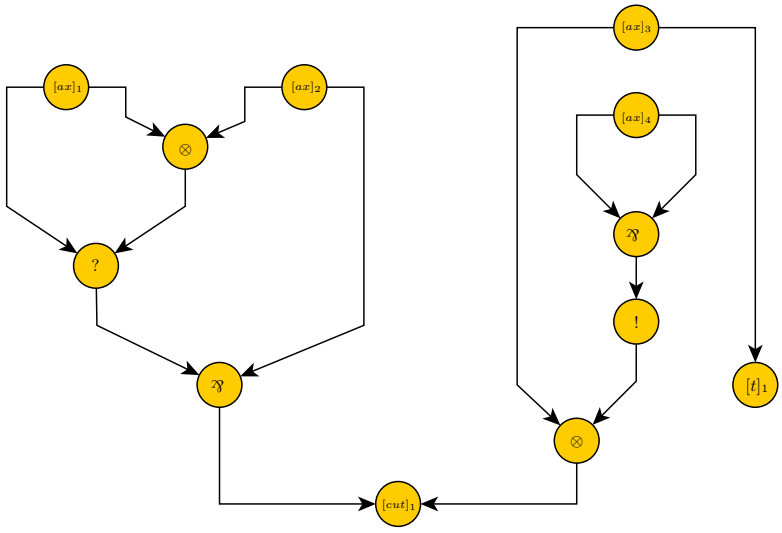


Fig. 4. Proof net of $(\Delta\, I)$

Indeed, the term $(\Delta\, I)$ can be typed in linear logic by extending the type system with a fix point equation on formulas (namely, the one used to define Scott domains: $D = D \to D$). By following its representation as a pure proof net, we get the corresponding GoI

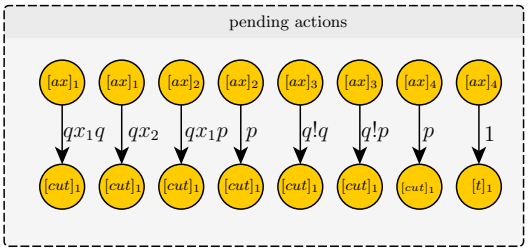


Fig. 5. Initial list of pending actions for the computation of the Execution Formula of $(\Delta\, I)$

interpretation, the following matrix with entries in the Girard dynamic algebra:

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

The matrix representation is redundant (the matrix is in some sense symmetric $a_{ij} = a_{ji}^*$, in some model for Λ^* we would say *hermitian*) and sparse. This drawback was one motivation in Danos and Regnier's paper (Danos and Regnier, 1993) 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):

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

and edges $((v_t, v_s), w)$ get a weight $w \in \Lambda^*$. Note that here 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:

$$\begin{aligned} E = \{&(([\mathrm{cut}]_1, [\mathrm{ax}]_1), qx_1q), (([\mathrm{cut}]_1, [\mathrm{ax}]_1), qx_2), (([\mathrm{cut}]_1, [\mathrm{ax}]_2), qx_1p), \\ &(([\mathrm{cut}]_1, [\mathrm{ax}]_2), p), (([\mathrm{cut}]_1, [\mathrm{ax}]_3), q!q), (([\mathrm{cut}]_1, [\mathrm{ax}]_3), q!p), \\ &(([\mathrm{cut}]_1, [\mathrm{ax}]_4), p), (([\mathrm{t}]_1, [\mathrm{ax}]_4), 1)\}. \end{aligned} \tag{3}$$

To compute the Execution Formula associated to the term, we initialise the list of pending actions with such edge set, see Figure 5, and we proceed to the evaluation of these actions on an initial empty graph as described in Equation (1) (for simplicity, at this stage the polarities are omitted in the notation):

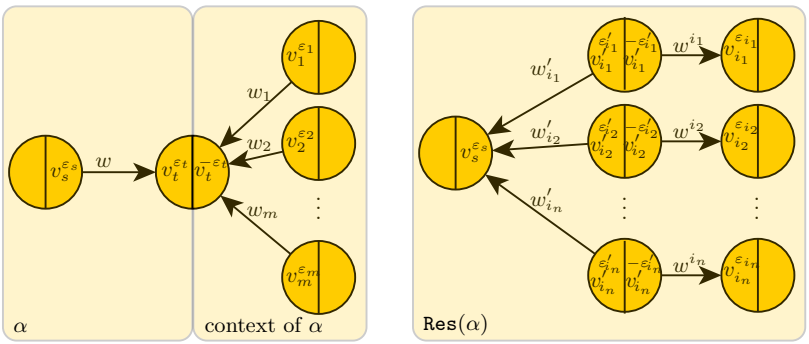


Fig. 6. Elementary computational step (as half-combustion). Note that the computation of m interactions originates $2n$ residual edges, with $n \leq m$

2.2. *The elementary computational step: generalized half-combustion*

Let $G \in \mathbb{G}_M^+$ be a polarised dynamic graph and let $\alpha = \langle(\varepsilon_t, \varepsilon_s), (v_t, v_s), w\rangle$ be an action on G. We define the *context* of α to be the set of all edges $\{\beta_1, \ldots, \beta_m\}$ of G such that

$$\beta_i = \langle(-\varepsilon_t, \varepsilon_i), (v_t, v_i), w_i\rangle \qquad \text{for some } \varepsilon_i,\ v_i, \text{ and } w_i.$$

In other words, the context of α is the set of edges belonging to the seminode $v_t^{-\varepsilon_t}$, that is, edges of G insisting on the same target node as α but with opposite target polarity.

Now, consider an action α and an element of its context, β_i. By the definition of dynamic graph, the weights of every edge are elements of a dynamic monoid M, that is $w = a$ and $w_i = b_i$ for some $a, b_i \in M$. If $b_i^* a$ is different from 0, let $a_i' b_i'^*$ be its normal form. Then the interaction between α and β_i generates a new node v_i' and new actions

$$\alpha_i := \langle(\varepsilon_i, \varepsilon), (v_i, v_i'), a_i'\rangle \quad \text{and} \quad \beta_i' := \langle(\varepsilon_s, -\varepsilon), (v_s, v_i'), b_i'\rangle, \tag{4}$$

where ε is arbitrarily chosen in $\{+, -\}$. The node v_i' is added, together with its outgoing edges (v_i, v_i') and (v_s, v_i'), to G. The set of all residual actions originated by α after performing the *elementary computational step* (that is the computation of all interactions for $i = 1, \ldots, m$, as depicted in Figure 6), is therefore the set

$$\texttt{Res}(\alpha) := \{\alpha_{i_1}, \beta_{i_1}', \ldots, \alpha_{i_n}, \beta_{i_n}'\} \subseteq \{\alpha_1, \beta_1', \ldots, \alpha_m, \beta_m'\} \tag{5}$$

where indices i_j are those for which $a_{i_j}' b_{i_j}'^* \neq 0$.

Remark 2. The number of residual pairs is possibly less than the number of actions β_i

(a) the action α and its context node v_t (b) the set Res(α) of residuals of α

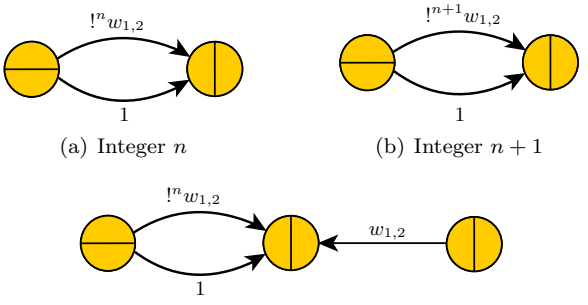
Fig. 7. Elementary computational step of an action α acting on a context with one edge

in the context, therefore we write $n \leq m$ residuals. The reason is that for some of them there may not be a stable form of the product ${b_i}^* a$ (*i.e.*, the result is null).

Example 3. As in Definition 3, assume $M = \Lambda^*$ and $a = !^2q$ and $b = w_{1,2}$. Let $\alpha = \langle(+,-),(v_t,v_s),a\rangle$ and assume that its context is the edge $\beta = \langle(-,-),(v_t,v_1),b\rangle$. Recall from Equation 2 that $b^*a = a'b'^*$ where $a' = !^3q$ and $b' = w_{1,2}$. We have in Figure 7 a step of computation.

Example 4 (Encoding natural numbers via Girard dynamic algebra). As a first step towards an embedding of recursive functions into dynamic graphs, we represent the natural number $n \in \mathbb{N}$ in terms of the polarised dynamic graph $G_{\Lambda^*}(n)$ in Figure 8 part (a), whereas in part (b) we show the representation of the successor of n. We illustrate how to obtain, through the same GoI reduction mechanism, a representation of the successor function on integers. For the sake of simplicity, the labels of nodes are here omitted, while keeping the fact that polarities bipartite nodes in two sets.

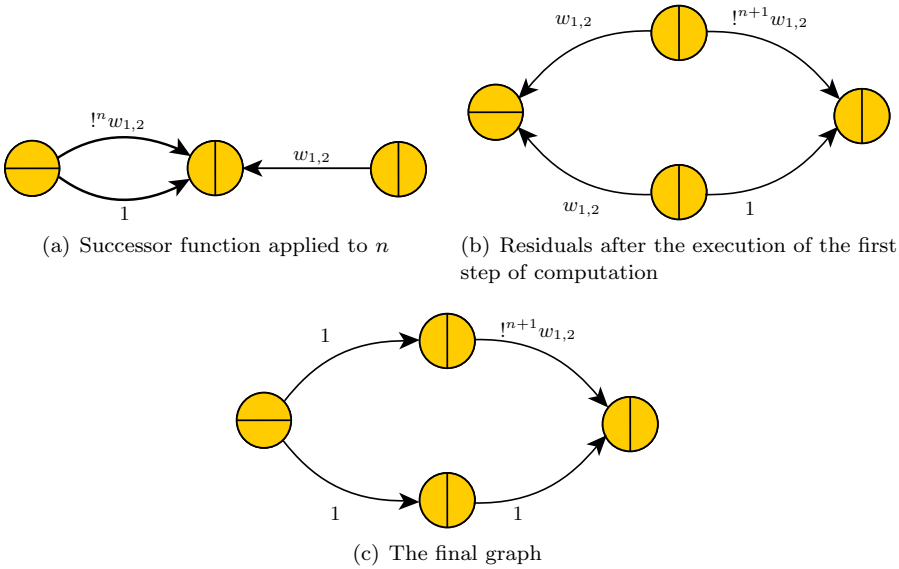
The application of the successor function to n is represented by the interaction between $G_{\Lambda^*}(n)$ and the action labelled with $w_{1,2}$ (notice that in GoI this interaction represents the reduction of a commutative cut). Figure 8 (c) displays the application of the successor function to the integer n, whilst Figure 9 shows the reduction steps. The reader may notice that the result (Figure 9 (c)) is not the representation of $n+1$ as given in Figure 8 (b)) as we would expect from Figure 8. To overcome this sort of problem, however, we exploit a modified interaction considered in (Pedicini and Quaglia, 2007) and called



(a) Integer n (b) Integer $n+1$

(c) Successor function applied to n

Fig. 8. Polarised dynamic graphs representing the two integer numbers n and $n+1$, and the successor function applied to n



(a) Successor function applied to n

(b) Residuals after the execution of the first step of computation

(c) The final graph

Fig. 9. Polarised dynamic graphs representing intermediate states during the reduction of the successor function applied to an integer number n

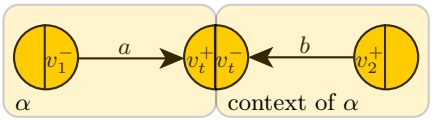
optimisation of one (see Figure 10 (a)). In case of residuals labelled by identity, such an interaction works by creating only one residual in place of the two prescribed by the usual interaction rule (see Equation 4). This example also illustrates how a computation runs during execution by means of elementary steps; and, on the other hand, the encoding of such a basic function (in the class of recursive functions) is an indication that other basic recursive functions can be realised by this computational device. Anyway, the problem of completeness of dynamic graphs with respect to the class of recursive functions is not considered with this encoding of integers, being beyond the scope of the present paper.

Example 5 (Computing languages of automata). In this example we show a coding into dynamic graphs for deterministic finite state automata. This approach is informally justified by the discussion of the particular case where paths are preserved by reduction. The aim of this example is to give some hints of non-standard applications of the GoI machinery, however the rigorous discussion of the general case is postponed to a future work.

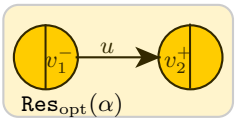
Let $\mathcal{A} = (T, q_0, F)$ be a deterministic finite state automaton with alphabet $A = \{a_1, \ldots, a_n\}$ and finite set of states $Q = \{q_0, q_1, \ldots q_m\}$.

The transition function $T : Q \times A \to Q$ for any pair (q, a) associates a new state $T(q, a)$. The state q_0 is called the initial state and $F \subset Q$ is the set of final (accepting) states.

We provide here a correspondence between automata and dynamic graphs on the Girard monoid Λ^*, such that the computation of the execution of the graph corresponds to the computation of the regular language accepted by the automaton.

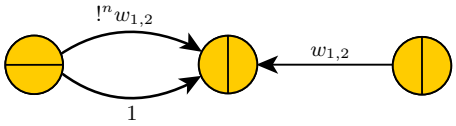


(a) the action α and its context node v_t

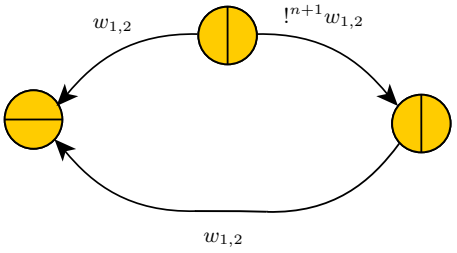


(b) the optimised set $\mathtt{Res}_{\mathrm{opt}}(\alpha)$ of residuals of α

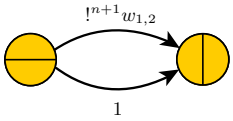
(a) Optimised interaction rule: the elementary step of computation (for the dynamic graph on the left) in general (if the product of the labels involved in the interaction is different from 0) includes the creation of a new source node v_s and of a couple of edges targeting the nodes v_1 and v_2 (with labels, say, u_1 and u). Nevertheless, in the particular case where $u_1 = 1$, an optimised rule can be applied avoiding the creation of the node v_s and only a residual is created (corresponding to an edge with source v_1 and target v_2 and labeled with u as depicted on the right).



(b) Successor function applied to n (as in Figure 9)



(c) Residuals after the execution of the first steps of computation (using the optimised interaction rules)



(d) The final optimised graph

Fig. 10. Optimisation of one

Definition 7. For any automaton $\mathcal{A}$ let us define its dynamic graph

$[\mathcal{A}] := (G, v_0, V_F)$ where $G = (V_q \cup V_t, E) \in \mathbb{G}^+_{\Lambda^*}$, $v_0 \in V_q$, and $V_F \subseteq V_q \cup V_t$ is a vertex set.

— every state $q \in Q$ is associated with a vertex $v_q \in V_q$;

— every element $a_i \in A$ is coded as the element $!w^*_{i,1}$ of Λ^*, then the mapping is extended to A^* by monoid homomorphism; namely, a word $x = x_1 x_2 \dots x_n \in A^*$ is represented by $!w^*_{i_n} \dots !w^*_{i_1} = !(w_{i_n} \dots w_{i_1})^*$, if $x_k = a_{i_k} \in A$. We notice that the empty word $\varepsilon \in A^*$ is then mapped onto $1 \in \Lambda^*$;

— transitions are represented via auxiliary vertices (composing the vertex subset V_t) and edges. In particular, every $(q_1, q_2) \in Q \times Q$ and $a \in A$ such that $\delta(q_1, a) = q_2$ is associated with a triplet $(e_1, e_2, v_{12}) \in E \times E \times V_t$ where

$$e_1 = ((+,-),(v_{q_1}, v_{12}), 1) \quad e_2 = ((-,+),(v_{q_2}, v_{12}), !w^*_{1,1});$$

— initial node q_0 is coded into the vertex v_0;

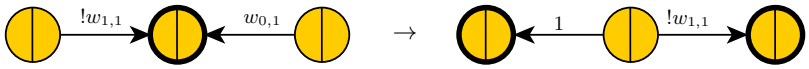


Fig. 11. The reduction rule for final nodes of the automaton.

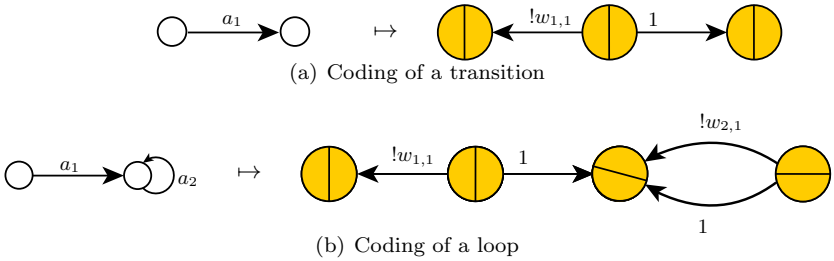


Fig. 12. Graph representations of some automata and their coding as dynamic graph

— the final nodes $q_f \in F$ are coded as elements of the vertex subset V_F. The property of being a "final" node is propagated during the computation by the rule depicted in Figure 11.

See Figure 12 for some examples of application of the above rules. We notice that the elementary step of computation preserves paths. Consider for instance the path along an automaton and its coding as a dynamic graph depticted in Figure 13(a). After a step of reduction we get the configuration depicted in Figure 13(b) and, subsequently, the configuration in Figure 13(c). (recall indeed that the involution * represents an arc reversal on dynamic graphs). Hence the path from the leftmost node to the rightmost node is $!w^*_{1,1}!w^*_{2,1}$, which corresponds to the element ab in A^*.

Consider the automaton $\mathcal{A}$ and its coding according to the above rules, depicted in Figure 14

We remark that $\mathcal{A}$ recognizes the regular language $L = \{a_1 u \mid u \in A^*\}$, where $A =$

.

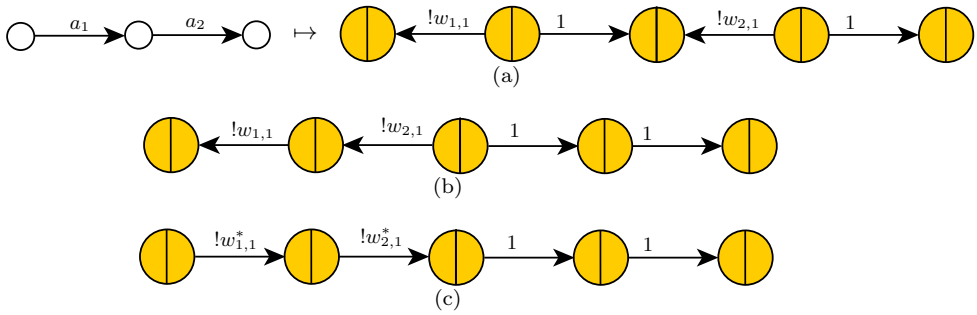


Fig. 13. Coding of an automaton and some steps of reduction

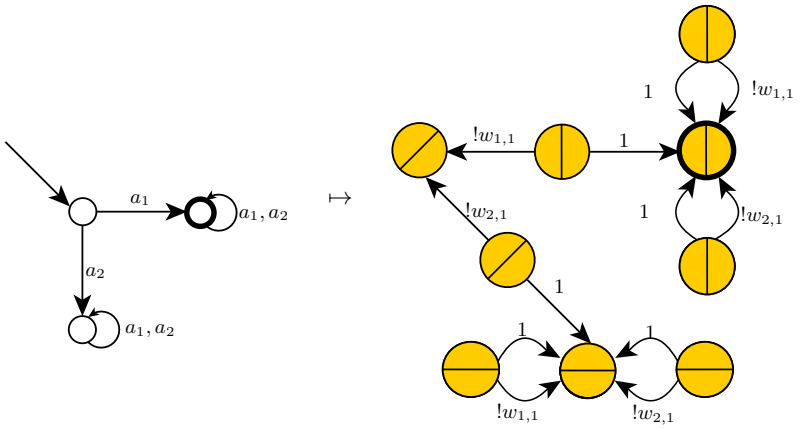


Fig. 14. Coding of the automaton $\mathcal{A}$

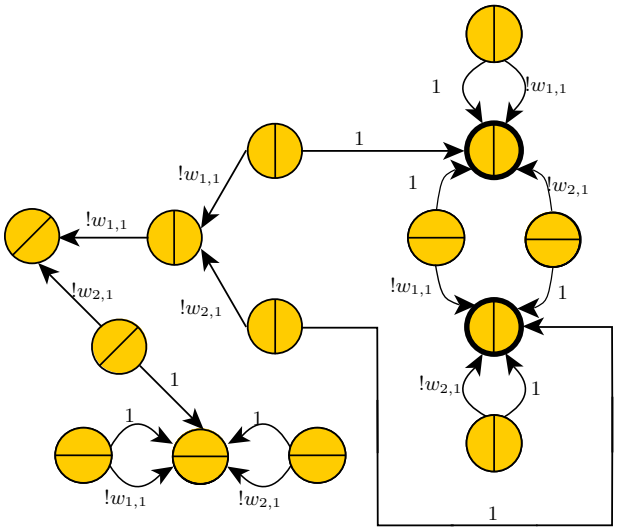


Fig. 15. Dynamic graph representing the automaton $\mathcal{A}$ after a step of reduction

$\{a_1, a_2\}$. After one reduction step (of the edges insisting on the target node) we get the polarised dynamic graph depicted in Figure 15.

We notice that the paths from the initial state to final nodes are $!w^*_{1,1}!^*w_{1,1}1$ and $!w^*_{1,1}!w^*_{2,1}1$ and they correspond to a_1a_1 and a_1a_2 in A^*, i.e. the words of length 2 accepted by the automaton.

Remark 3 (The role of polarity in DVR). In the DVR procedure, a strong assumption on the input is made: that is, the input graph is the interpretation of a proof-net into a virtual net (the proof net, at its turn, could be the interpretation of a λ-term). Consequently, any edge in the virtual net represents a half of a straight path. We recall that a straight path in a proof net is any path which is neither bouncing nor twisting. We

say that a path is *non-bouncing* if it does not contain any edge a followed by the same edge taken in the reversed direction a^*, whereas the path is *non-twisting* if it does not contain any edge a_j follow by a distinct premise of the same link a_j^*. Note that although neither non-bouncing nor non-twisting properties are preserved by path composition, we know that all the weight of straight paths incident on the same node form two orthogonal sets, such that residuals of orthogonal paths are still orthogonal (Pedicini and Quaglia, 2007, §3).

The original purpose of introducing polarity in DVR procedure was then to distinguish the elements belonging to each of those orthogonal sets, to reduce useless computation. Indeed if two labels belong to orthogonal sets, then their normal form is 0: if this information is properly stored in the polarity sign (and properly propagated to the residuals) then one can avoid to compute their normal form. This mechanism is automatised by restricting the computation to those edges with opposite target polarities. In the more general setting presented here, this orthogonality property of incident edges does not hold: assigning a polarity to the edges is a procedure more oriented to flow control than to the optimisation of the computation.

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 $\mathbf{0} + \alpha = \alpha$. Following Rutten (2005) we give the following

Definition 8. A *stream* S on A is a sequence $S : \mathbb{N} \to A$ of elements of A. We define the set A^ω of all streams as $A^\omega = \{S \mid S : \mathbb{N} \to \mathbf{A}\}$.

For a stream S, we call $S(0)$ the *initial value* of S and we adopt the following notations:

$$S = (S(0), S(1), S(2), \dots)$$

and

$$\alpha :: S = (\alpha, S(0), S(1), S(2), \dots)$$

and nil as the stream defined by the equation

$$\mathsf{nil} = \mathbf{0} :: \mathsf{nil}.$$

We consider the following operations on shifts.

Definition 9 (Shift and zip). The *shift* operation (also called *derivative*, or *tail*) is defined by the equation

$$S = S(0) :: \mathtt{shift}(S). \tag{6}$$

The *zip* of a couple of streams S and T is given by the system of equations

$$\begin{cases} \texttt{shift}(\texttt{zip}(S,T)) = \texttt{zip}(T, \texttt{shift}(S)) \\ \texttt{zip}(S,T)(0) = S(0) \end{cases}$$

We also adopt the following notation $S \ltimes T := \texttt{zip}(S,T)$.

Remark 4. Notice that by definition

$$\begin{aligned} \texttt{zip}(S,T)(2i) &= S(i), \\ \texttt{zip}(S,T)(2i+1) &= T(i) \end{aligned}$$

for all $i \in \mathbb{N}$. Also note that `zip` is not commutative.

Definition 10 (Strip). The *strip* of a proper sub-stream S_2 from a stream S_1 is

$$\texttt{strip}(S_1, S_2) := \begin{cases} \texttt{strip}(\texttt{shift}(S_1), \texttt{shift}(S_2)) & \text{if } S_1(0) = S_2(0) \\ S_1(0) :: \texttt{strip}(\texttt{shift}(S_1), S_2) & \text{if } S_1(0) \neq S_2(0) \end{cases}$$

Definition 11 (Weak-bisimulation and weakly-bisimilar streams). 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:

$$S(0) = T(0) \text{ and } (\texttt{shift}(S), \texttt{shift}(T)) \in \rho; \tag{7a}$$

$$S(0) = \mathbf{0} \text{ and } (\texttt{shift}(S), T) \in \rho; \tag{7b}$$

$$T(0) = \mathbf{0} \text{ and } (S, \texttt{shift}(T)) \in \rho. \tag{7c}$$

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

Note that $S \approx T$ if and only if $\texttt{strip}(S, \mathsf{nil}) = \texttt{strip}(T, \mathsf{nil})$. Needless to say, it is not possible to have $\texttt{strip}(\mathsf{nil}, \mathsf{nil})$ since nil is not a subsequence of itself.

We finally introduce the notion of *node view* (or *view of base* v) of a stream S of actions (of a polarised dynamic graph):

Definition 12. Given a stream of actions S and a node v, the polarised view of base v^ε is defined by selecting actions with target node v and opposite target polarity with respect to the polarity of the base. Namely:

$$(S)_{v^\varepsilon} := \begin{cases} \mathsf{nil} & \text{if } S = \mathsf{nil} \\ S(0) :: (\texttt{shift}(S))_{v^\epsilon} & \text{if } S(0) = \langle (-\varepsilon, \varepsilon_s), (\rho(v), \rho(v_s)), w \rangle \\ (\texttt{shift}(S))_{v^\epsilon} & \text{otherwise} \end{cases}$$

We define also the view of base v as $(S)_v = (S)_{v^+} \ltimes (S)_{v^-}$.

In a similar way, it is possible to define the *graph view* of a stream of actions S:

Definition 13. Given a stream of actions S and a graph $G = (V,E) \in \mathbb{G}_M^+$, we define a

sub-stream of actions by selecting actions with target node in V:

$$(S)_G := \begin{cases} \mathsf{nil} & \text{if } S = \mathsf{nil} \\ S(0) :: (\mathtt{shift}(S))_G & \text{if } S(0) = \langle(\varepsilon, \varepsilon_s), (\rho(v), \rho(v_s)), w\rangle \text{ and } v \in V \\ (\mathtt{shift}(S))_G & \text{otherwise.} \end{cases}$$

Definition 14. Given a set $X \subset A$ we define

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

Note that the above definition gives a stream which is defined up to a permutation.

Given an action α, and the corresponding set of residuals $\mathtt{Res}(\alpha)$ (as in Equation (5)) we define the finite stream obtained rearranging actions in $\mathtt{Res}(\alpha)$ as

$$\mathsf{execute}(\alpha) := \mathtt{streamset}(\mathtt{Res}(\alpha)).$$

Note that $\mathtt{Res}(\alpha)$ can be an empty set, in this case

$$\mathsf{execute}(\alpha) = \mathtt{streamset}(\emptyset) = \mathsf{nil}.$$

Remark 5. This non deterministic definition stems from one of the main features of local and asynchronous execution displayed in virtual reductions: parallel implementation can get rid of the typical confluence and the synchronisation difficulties in distributed systems, inasmuch as the algebraic machinery ensures the correctness of the computation.

2.4. *Sequential Abstract Machine*

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

— the *stack* S, which is used to store the current action;
— the *environment* E, is a node of the graph and it provides the local environment where the current action has to be performed, or it not determined (`NULL`);
— the *control* C is a stream of 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;
— the *dump* D is the current dynamic graph and contains the environment for next actions.

For a given dynamic monoid M, at any step of computation, the machine has a configuration taken in the set:

$$\mathcal{C}_M := \left\{ (S, E, C, D) \text{ such that } S \in \bigcup_{n \geq 0} A_M^n, D = (V_D, E_D) \in \mathbb{G}_M^+, E \in V_D, C \in A_M^\omega \right\},$$

where S is a finite set of actions of A_M (see Definition 6), D is a dynamic graph, E is a vertex of D and C is a sequence, possibly infinite of actions from A_M.

For the sake of simplicity, we adopt the following notation in the case where we have

to add nodes to the dump: $D \cup \{v_t, v_s\} = (V_D \cup \{v_t, v_s\}, E_D)$, or where we have to add an edge $D \cup \{\alpha\} = (V_D, E_D \cup \{\alpha\})$. Moreover, in the latter case we denote the edge to be added in the same way as the action, namely if the action uses references to nodes $\alpha = \langle(\varepsilon_t, \varepsilon_s), (\rho(v_t), \rho(v_s)), w\rangle$, then the corresponding edge is $\langle(\varepsilon_t, \varepsilon_s), (v_t, v_s), w\rangle$.

We denote the empty dump (resp. a new/uninitialized environment) with $\emptyset$ (resp. $\mathtt{NULL}$). We also introduce the notation

$$\mathsf{target}(\alpha) = \begin{cases} v_t & \text{if } \alpha = \langle \varepsilon, (v_t, v_s), w\rangle \\ \mathtt{NULL} & \text{if } \alpha = \mathbf{0}. \end{cases}$$

This function takes as a value a node of the graph, considered as the environment where the action represented by α has to be performed (if $\alpha = \mathbf{0}$ we get the $\mathtt{NULL}$ environment).

Then $\mathsf{target}(\alpha)$ is added to the graph as a node. Again if $\alpha = \mathbf{0}$ we put

$$D \cup \{\mathsf{target}(\mathbf{0})\} = D.$$

We define the basic operations of this SECD machine in terms of a series of transitions from one configuration to another:

$$\mathsf{name}\frac{\text{configuration before}}{\text{configuration after}}$$

$$\mathsf{INIT}\frac{(\langle\rangle, \mathtt{NULL}, \mathsf{nil}, \emptyset)}{(\langle\rangle, \mathtt{NULL}, \mathtt{read}(), \emptyset)} \qquad \mathsf{ENV}\frac{(\langle\alpha\rangle, \mathtt{NULL}, C, D) \qquad \alpha \neq \mathbf{0}}{(\langle\alpha\rangle, \mathsf{target}(\alpha), C, D \cup \{\mathsf{target}(\alpha)\})}$$

$$\mathsf{POP}\frac{(\langle\rangle, \mathtt{NULL}, \alpha :: C, D)}{(\langle\alpha\rangle, \mathtt{NULL}, C, D)} \qquad \mathsf{NENV}\frac{(\langle\alpha\rangle, \mathtt{NULL}, C, D) \qquad \alpha = \mathbf{0}}{(\langle\rangle, \mathtt{NULL}, C, D)}$$

$$\mathsf{HC}\frac{(\langle\alpha\rangle, \mathsf{target}(\alpha), C, D)}{(\langle\rangle, \mathtt{NULL}, C \ltimes \mathsf{execute}(\alpha), D \cup \{\alpha\})}$$

Fig. 16. Sequential Abstract machine

We denote by $\mathsf{R}_1;\mathsf{R}_2$ the composition of the application of the transition rule R_1 followed by the application of R_2. So, let us assume that the machine has configuration (S, E, C, D). Then, we obtain the new configuration by applying the transition R_1 and we write $\mathsf{R}_1(S, E, C, D) = (S', E', C', D')$.

In Figure 16, we give the five types of transition which fully describe our first machine, the sequential abstract machine; the infinite execution loop for this machine is given by

$$(\mathsf{INIT};((\mathsf{POP};\mathsf{NENV})^*;\mathsf{POP};\mathsf{ENV};\mathsf{HC})^*)^* \tag{8}$$

to be applied to the initial configuration $\mathsf{S}_0 := (\langle\rangle, \mathtt{NULL}, \mathsf{nil}, \emptyset)$.

Note that the infinite loop in Equation 8 is the only way to concatenate the transitions; in fact, if we consider hypothesis conditions on the rules, we have the following mutually

exclusive conditions on the configuration:

$$(\mathsf{configuration} = \mathsf{S}_0) \tag{T_0}$$

$$(\mathsf{configuration} = (\langle\alpha\rangle, \mathtt{NULL}, C, D)) \text{ and } \alpha = \mathbf{0} \tag{T_1}$$

$$(\mathsf{configuration} = (\langle\alpha\rangle, \mathtt{NULL}, C, D)) \text{ and } \alpha \neq \mathbf{0} \tag{T_2}$$

$$(\mathsf{configuration} = (\langle\alpha\rangle, \mathsf{target}(\alpha), C, D)) \tag{T_3}$$

We can restate the machine in a procedural style as reported in Algorithm 1.

Algorithm 1 Restatement of the execution cycle (Eq. 8) of the sequential machine in Fig. 16

```
configuration ← S0
while true do
    configuration ← INIT(configuration)
    while ¬T0 do
        configuration ← POP(configuration)
        if T1 then
            configuration ← NENV(configuration)
        else
            configuration ← ENV; HC(configuration)
        end if
    end while
end while
```

Remark 6. With the settings of the Example 2, the stream returned by the function `read`() is the GoI interpretation of the λ-term (ΔI) concatenated with the stream nil.

Remark 7. By construction, at every step of computation the stack S is either the *empty stack*, $\langle\rangle$, or it contains the next action acting on the graph $\langle\alpha\rangle$. However, by definition S may contain any finite number of actions: this more general setting is actually used to prove the correctness of the machine, see the definition in Figure 17 below.

Remark 8 (On the effectiveness of stream-based computation). The fact that the computation is stream oriented — i.e., the input is a finite or infinite stream — implies that the machine never stops: if the input is infinite, then it has no last non-null action; if the input is finite, then it is a stream which eventually coincides with nil. Both cases display the problem of getting the result of the computation, for the result has to be *extracted* from the dump D. Thus, nodes in the dump graph must be partitioned in terminal nodes and non-terminal ones and the sub-graph of terminal nodes is the result of the computation. To manage the terminal nodes, during initialisation we need an explicit information tagging terminal nodes in the read stream. Then, in the course of a computation, this tag is broadcasted to nodes which are source node of actions pointing to terminal nodes.

As an alternative, actions pointing to terminal nodes can be emitted through a devoted

output stream. Note that in the special case in which the read stream has a finite number of actions pointing to terminal nodes, the termination can be checked dynamically during the computation even if the read stream is infinite: all we need to keep updated, at each step of computation, is the number of actions with a terminal node as source. This number can be computed by counting the residual actions with terminal source node and by adding this value to the previously computed sum (decreased by one, if the current action is terminal). When the total number of such actions in the stream equals zero, then any further step of computation will never produce new terminal actions and the machine can be stopped.

2.5. *Generalised full combustion abstract machine*

Now we introduce a variant of the SECD machine defined in Section 2.4. This variant processes all the actions in a view of base v before focusing on another node: this execution strategy generalizes to arbitrary dynamic monoids the *full combustion strategy* for DVR.

$$\mathsf{INIT}\ \frac{(\langle\rangle, \mathtt{NULL}, \mathsf{nil}, \emptyset)}{(\langle\rangle, \mathtt{NULL}, \mathtt{read}(), \emptyset)} \qquad \mathsf{NENV}\ \frac{(\langle\rangle, \mathtt{NULL}, C, D) \qquad (C)_v = \mathsf{nil}}{(\langle\rangle, \mathtt{NULL}, C, D)}$$

$$\mathsf{POP}^\dagger\ \frac{(\langle\rangle, \mathtt{NULL}, C, D) \qquad X = \{\alpha_1, \ldots, \alpha_n\} \qquad (C)_v = \mathtt{streamset}(X) \qquad \mathsf{ZOV}(v)}{(\langle\alpha_1, \ldots, \alpha_n\rangle, v, \mathtt{strip}(C, (C)_v), D \cup \{v\})}$$

$$\mathsf{HC}\ \frac{(\alpha :: S, v, C, D)}{(S, v, C \ltimes \mathsf{execute}(\alpha), D \cup \{\alpha\})}$$

Fig. 17. Sequential Full Combustion Abstract machine

Operations of the *full combustion* machine are given in Figure 17; they differ from the ones given in the sequential case since the POP and ENV rules have been combined in a unique $\mathsf{POP}^\dagger$ rule. This rule depends on the choice of a node v, and it can be applied whenever the stream C contains a finite set of actions whose target refers to v. Moreover, if v does not appear as source node of any action in the stream we have that (as a consequence of the half combustion rule) that no further action with v as target node can be produced as residual of a reduction step. Any node which satisfies the condition (of not appearing as) source node of any action in the core stream is called a *zero-valence node*, see (Danos et al., 1997). So in Figure 17, we denote the property of a node to be of zero-out-valence by $\mathsf{ZOV}(v)$, the choice of v must fall on zero-out-valence nodes moreover if v is such that no action in C has that node as the target, then we apply the NENV rule. Processing actions in the stack by the elementary computational step HC is then carried one action at time step as in the sequential machine until the empty stack is reached.

The behaviour of this machine is described by the following loop:

$$(\mathsf{INIT}; (\mathsf{NENV}^*; \mathsf{POP}^\dagger; \mathsf{HC}^*)^*)^*. \tag{9}$$

2.6. *Strong local confluence in the case $M = \Lambda^*$*

In the introduction to Section 2, we provided the general description of what is the model of calculus based on a computing device with a state represented by a dynamic graph G and the task to be executed represented by a sequence of actions A, see Equation (1). The machine consumes the first action of the sequence, modifies the state (*i.e.*, the dynamic graph) and finally the residual actions Δ_α produced by the execution, are added to the sequence of actions.

This step of computation has been then formalised as the elementary step of computation, Section 2.2 and included in the formal description of the SECD machine in Section 2.4.

When $M = \Lambda^*$ and the input is the coding of a λ-term into a virtual net (see Example 2) then an elementary step of computation coincides with an half-combustion step of DVR. In (Danos and Regnier, 1993), 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. Moreover that calculus was in line with the theory of interaction nets and a strong confluence property was showed: in fact, the algebraic modification was sufficient to keep coherent the computation. As a consequence, 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 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 as well (Danos et al., 1997; Pedicini and Quaglia, 2007).

Remark 9. Inasmuch as a term in untyped λ-calculus may not have a finite execution (even a normalizing one), we need to be able to cope with an infinite output to design a machine that can evaluate terms in parallel (with two or more computational units exchanging data). This is a further motivation for a stream-based approach, which allows the infinite sequence of actions to be used as input in a different 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.

Let us introduce the following notation on tensors which represent components of the multiple units of the parallel machine. First of all, we can use a compact notation for k stacks as follows:

$$\bigotimes_{i=1}^{k} S_i = S_1 \otimes S_2 \otimes \cdots \otimes S_k$$

$$\textsf{INIT}\ \frac{(\langle\rangle^k, \texttt{NULL}^k, \textsf{nil}^k, \emptyset^k)}{(\langle\rangle^k, \texttt{NULL}^k, \texttt{read}() \otimes \textsf{nil}^{k-1}, \emptyset^k)}$$

$$\textsf{ENV}\ \frac{(\bigotimes_{i=1}^k \langle\alpha_i\rangle, \texttt{NULL}^k, \bigotimes_{i=1}^k C_i, \bigotimes_{i=1}^k D_i) \qquad \alpha_i \neq \mathbf{0} \text{ for some } i \in \{1,\ldots,k\}}{(\bigotimes_{i=1}^k \langle\alpha_i\rangle, \bigotimes_{i=1}^k \textsf{target}(\alpha_i), \bigotimes_{i=1}^k C_i, \bigotimes_{i=1}^k D_i \cup \{\textsf{target}(\alpha_i)\})}$$

$$\textsf{POP}\ \frac{(\langle\rangle^k, \texttt{NULL}^k, \bigotimes_{i=1}^k \alpha_i :: C_i, \bigotimes_{i=1}^k D_i)}{(\bigotimes_{i=1}^k \langle\alpha_i\rangle, \texttt{NULL}^k, \bigotimes_{i=1}^k C_i, \bigotimes_{i=1}^k D_i)}$$

$$\textsf{NENV}\ \frac{(\bigotimes_{i=1}^k \langle\alpha_i\rangle, \texttt{NULL}^k, \bigotimes_{i=1}^k C_i, \bigotimes_{i=1}^k D_i) \qquad \alpha_i = \mathbf{0} \text{ for all } i \in \{1,\ldots,k\}}{(\langle\rangle^k, \texttt{NULL}^k, \bigotimes_{i=1}^k C_i, \bigotimes_{i=1}^k D_i)}$$

$$\textsf{HC}\ \frac{(\bigotimes_{i=1}^k \langle\alpha_i\rangle, \bigotimes_{i=1}^k v_i, \bigotimes_{i=1}^k C_i, \bigotimes_{i=1}^k D_i) \qquad \alpha_{i_1} \neq \mathbf{0} \ldots \alpha_{i_d} \neq \mathbf{0} \quad \alpha_i = \mathbf{0} \text{ if } i \neq i_j}{(\langle\rangle^k, \texttt{NULL}^k, \bigotimes_{i=1}^k C_i \ltimes (\textsf{execute}(\alpha_{i_1}))_{D_i} \ltimes \ldots \ltimes (\textsf{execute}(\alpha_{i_d}))_{D_i}, \bigotimes_{i=1}^k D_i \cup \{\alpha_i\})}$$

Fig. 18. Parallel k-Units Synchronous Abstract Machine

Then, in the special case in which all the S_i's are the empty stack, we introduce such a notation:

$$\langle\rangle^k = \underbrace{\langle\rangle \otimes \cdots \otimes \langle\rangle}_{k-\text{times}} = \bigotimes_{i=1}^k \langle\rangle.$$

Note that the dumped graph $D = D_1 \otimes D_2 \otimes \cdots \otimes D_k$ can be defined as the union of edge sets in the individual D_i that is

$$D = \bigcup_{i=1}^k D_i.$$

3.1. *Synchronous Case*

The synchronous model of execution makes the machine step into the computational cycle in a synchronous way. This means that the same step of computation is performed (synchronously) on any computational unit. This forces to mix together the rules ENV and NENV: in Figure 18 such a mixed rule is referred to as ENV, while NENV is used to denote the special situation in which all the current actions in the stack of every unit are **0**.

Note also that at any step of computation the dumped graphs provide a partition of the global current graph $D = D_1 \cup \cdots \cup D_k$; one problem arises with respect to the edges with a source node on a graph and target node on the other. To avoid duplication of nodes, we adopt the following strategy: edges are stored on a graph by accessing the source node with the reference and the target node explicitly. Thus in the HC step of Figure 18 we write $D_i \cup \{\alpha_i\}$ meaning that from an action α_i we dump an edge specified by the target node and the reference to the source node:

$$\alpha_i = \langle \varepsilon_i, (v_t^i, \rho(v_s^i)), w_i \rangle, \tag{10}$$

which uniformly treats any edge of the graph and solves the question of the special edges which crosses the partition of the graph.

Remark 10.

1. The computation of the stream $(\mathsf{execute}(\alpha_j))_{D_i}$ is performed by the j-th computing unit, while the sub-stream relative to the nodes in the i-th dumped graph D_i 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. Actions in the set of residuals $\mathtt{Res}(\alpha_i)$ of the action α_i, possibly have target nodes in the dumped graph distributed on the units. In fact, the target node of these newly created actions , coincides with the α_i source node v_s^i. The unit hosting the node v_s^i is selected by considering the view with respect to the dumped graph of the unit, namely when from the stream $\mathsf{execute}(\alpha_j)$ we extract the sub stream $(\mathsf{execute}(\alpha_j))_{D_i}$ by selection of actions with target node in D_i.
   Notice that the source node of pairs of residuals coming from the same action-action interaction is a newly created node $v = \mathtt{new}()$. 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 in previous steps of computation.

3.2. *Asynchronous Case*

In the asynchronous case one deals with modelling 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.

Let us consider a parallel machine with k computing units whose state is therefore represented by

$$(p, S, E, C, D) = (p, S_1 \otimes \cdots \otimes S_k, E_1 \otimes \cdots \otimes E_k, C_1 \otimes \cdots \otimes C_k, D_1 \otimes \cdots \otimes D_k)$$

where $p \in \{1, \ldots, k\}$ is the control of the unit to be activated at the next transition step. The asynchronous model of execution makes the machine units to step in the computational cycle in an independent way. While each unit has to follow the execution cycle as specified in Equation 8, irrespective of the application on different units. In a way, this amounts to performing in one step any combination of k rules on the k components.

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 update rules in Figure 19, where the next scheduled unit p' is randomly chosen with uniform probability in $\{1, \ldots, k\}$. In fact, this asynchronous machine is very similar to the sequential one (and the two definitions coincide for $k = 1$). We recognise conditions for the application which are the very same $\mathsf{T_0}$, $\mathsf{T_1}$, $\mathsf{T_2}$ and $\mathsf{T_3}$ of the sequential machine albeit parametrised by the chosen component p.

The sequence of controls p is by itself a stream (of integers in the set of unit identifiers:

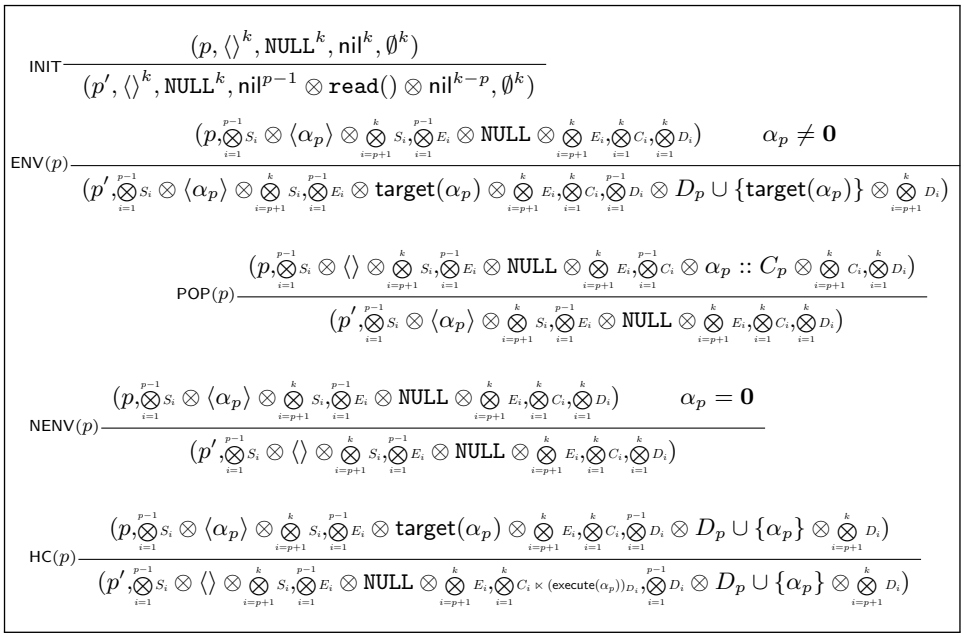


Fig. 19. Parallel k-units Asynchronous Abstract Machine

$\{1, \ldots, k\}$). In place of a random sequence, we may force a particular scheduling by explicitly specifying the sequence.

Remark 11. By choosing the scheduling constant to 1 and by allocating newly created nodes (see Remark 10.2) to the first computing unit, we recover the sequential machine. Moreover, if we fix the round-robin scheduling $1, 2, \ldots, k, 1, 2, \ldots k, \ldots$ which scans the computing units sequentially one after the other, we have a k-steps correspondence with the parallel synchronous model.

In the sequel, we will use full combustion strategy as a way to show invariance of execution with respect to the parallel version. To this aim, we have to introduce full combustion also in the case of k-units machines. This strategy is straight-forwardly implemented in the asynchronous case by adding a $\mathsf{POP}^\dagger$ rule which performs on the unit hosting the chosen zero-out-valence node v. Therefore, the scheduling sequence in the full combustion strategy for the parallel asynchronous machine consists in a sequence of nodes v from which depends the sequence of hosting units. For a given node v we denote the hosting unit by $p(v)$.

The POP-rule for the k-units machine performing full combustion is therefore given as

$$\mathsf{POP}^\dagger(v)\ \frac{p = p(v) \quad (p, \langle\rangle^k, \mathtt{NULL}^k, \bigotimes_{i=1}^{k} C_i, \bigotimes_{i=1}^{k} D_i) \qquad X = \{\alpha_1, \ldots, \alpha_n\} \qquad (C_{p(v)})_v = \mathtt{streamset}(X) \qquad \mathsf{ZOV}(v)}{(p, \langle\rangle^{p-1} \otimes \langle\alpha_1, \ldots, \alpha_n\rangle \otimes \langle\rangle^{k-p}, \mathtt{NULL}^{p-1} \otimes v \otimes \mathtt{NULL}^{k-p}, \bigotimes_{i=1}^{p-1} C_i \otimes \mathtt{strip}(C_p, (C_p)_v) \otimes \bigotimes_{i=p}^{k} C_i, \bigotimes_{i=1}^{p-1} D_i \otimes D_p \cup \{v\} \otimes \bigotimes_{i=p+1}^{k} D_i)}$$

thus the full combustion consists in applying rules in a loop similar to Equation 9 by

following scheduling driven by the choice of a node v and acting on the hosting unit $p(v)$:

$$(\mathsf{INIT};(\mathsf{NENV}^*;\mathsf{POP}^\dagger(v);\mathsf{HC}^*)^*)^*. \tag{11}$$

Note that by "HC", we mean a k-ary version of the half combustion rule depicted in Figure 17 since in this mode of execution at most one stack is non empty at any given step

$$\mathsf{HC}\ \frac{(p, \langle\rangle^{p-1} \otimes \alpha :: S \otimes \langle\rangle^{k-p}, \mathtt{NULL}^{p-1} \otimes v \otimes \mathtt{NULL}^{k-p}, \bigotimes_{i=1}^{k} C_i, \bigotimes_{i=1}^{k} D_i)}{(p, \langle\rangle^{p-1} \otimes S \otimes \langle\rangle^{k-p}, \mathtt{NULL}^{p-1} \otimes v \otimes \mathtt{NULL}^{k-p}, \bigotimes_{i=1}^{k} C_i \ltimes (\mathsf{execute}(\alpha))_{D_i}, \bigotimes_{i=1}^{p-1} D_i \otimes D_p \cup \{\alpha\} \otimes \bigotimes_{i=p+1}^{k} D_i)}$$

Note that in this version of the machine the schedule is determined by the choice of the node selected in the POP rule, therefore it can be omitted when stating the POP and HC.

4. Execution equivalence

In this section we assume $M = \Lambda^*$ and the input is the coding of a λ-term into a virtual net. As mentioned in Section 2.6, under these assumptions, the sequential machine realises the graph reduction introduced in (Danos et al., 1997), namely DVR. 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 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).

We need to introduce the notion of equivalence of the states of two machines.

Definition 15 (Dynamic graph isomorphism). A graph isomorphism $\phi : D_1 \to D_2$ is a bijection between graphs preserving adjacency, i.e., v_1 and v_2 are adjacent if and only if $\phi(v_1)$ and $\phi(v_2)$ are adjacent. We extend this definition to polarised dynamic graphs by assuming that ϕ preserves the labels of edges and the product of target polarities of pairs of edges inciding on the same node. In particular, for each edge $e_1 = ((\varepsilon_t, \varepsilon_s), (v_t, v_s), w)$ of D_1 one has $\phi(e_1) = ((\varepsilon'_t, \varepsilon'_s), (\phi(v_t), \phi(v_s)), w)$ and for all pairs of edges inciding on the same node, the corresponding target polarities, say ε_{t_1} and ε_{t_2}, satisfy $\varepsilon_{t_1}\varepsilon_{t_2} = \varepsilon'_{t_1}\varepsilon'_{t_2}$ [†].

Definition 16. Let $A = (S_1, E_1, C_1, D_1)$ and $B = (S_2, E_2, C_2, D_2)$ configurations of the machines M_1 and M_2, respectively. We say that A is equivalent to B ($A \simeq B$) whenever

1. D_1 and D_2 are isomorphic as polarised dynamic graphs, namely there exists an isomorphism ϕ of dynamic polarised graphs, such that:

[†] Notice that by construction, when considering couples of dynamic graphs generated by the sequential abstract machine, the product of source polarities of edges outgoing from the same node is automatically preserved.

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 $\phi(v)$, $(C_1)_v \approx \sigma((C_2)_{\phi(v)})$ for some permutation of actions σ, and
3. the two stacks contain isomorphic actions: $S_1 = \langle\alpha\rangle$ is isomorphic with $S_2 = \langle\beta\rangle$ (*i.e.*, $\beta = \phi(\alpha)$).

Lemma 1. The state equivalence $\simeq$ is an equivalence relation.

Proof. Trivial: $\simeq$ is the intersection of two equivalence relations. □

Another important fact is that:

Lemma 2. For any fixed polarised node v^ϵ, actions in $\alpha \in (S)_{v^\epsilon}$ are originated by actions acting on the same node, i.e., there exists v_0 such that for any $\alpha \in (S)_{v^\epsilon}$ there exists β with $\mathsf{target}(\beta) = v_0$ and $\alpha \in \mathtt{Res}(\beta)$.

In some sense this lemma guarantees that all the elementary steps of reduction performed on actions with the same target node are done on the same computational unit. This lemma follows from the Definition 12 and it provides evidence that execution done on parallel machines performs in a way isomorphic to sequential ones. Nevertheless, this property is not sufficient to guarantee the state equivalence of the resulting graphs after the execution of a single action.

A node in the dumped graph without incident actions is called a *ghost* node (Pedicini and Quaglia, 2007, §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 17. In DVR the combustion strategy (Danos et al., 1997, §4.1) chooses a node of valence 0 and performs all the possible actions on that node.

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 and performing partial execution of incident actions.

We provide here the analogue proof of correctness in the setting of abstract machines. First, we consider the particular synchronisation of the sequential machine given in Figure 17: such a synchronisation gives us a machine implementing full combustion . Then, we show that the machine implements an asynchronous version of full combustion. In fact, in the full combustion we need to focus on nodes which appear as target node in actions, while not appearing in the dumped graph; the 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). The full combustion (see Figure 17) processes all the actions in a view of base a zero out valence node v before focusing on another node.

Remark 12. We notice that sequential abstract machines are initialised with empty configurations. Also, after a read operation, the view of base v even if considered on multiple streams (like in the case of parallel machines) corresponds to the view with respect to just one stream because the base v possibly belongs to at most one of the dumped graphs (as a consequence of the redistribution of residual actions with respect to the dumped graph where a node is dumped, see Equation (10)). Thus, the notion of full action $v.A$ is the configuration obtained starting with configuration A after the execution of one step of full-combustion with base v that is:

$$v.A := \mathsf{POP}_v^{\dagger}; \mathsf{HC}^*(A)$$

When applied to parallel machines, it results in performing the pop of the set of actions with target node v and then by iterating the elementary computational step on such a set of actions on the unit containing the node v. Notice also that the dumped graphs of the other units are left unchanged.

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 the configuration A of the machine M_1 is equivalent to the configuration B of the machine M_2, (i.e., there exists an isomorphism ϕ such that $\phi(A) = B$), then we have that $v.A \simeq \phi(v).B$.

Proof. If we consider a zero-out-valence s-node v, and we compute the residuals of the view of base v, i.e. residuals obtained by processing actions in $(C)_v$, then we may consider the complete set of residuals relative to the node v: let us indicate this set by $X_v = \bigcup_{\alpha \in (C)_v} \mathsf{execute}(\alpha)$; note that this cumulative set of residual actions is redistributed in any order whatsoever on the various unit core-streams, according to the respective target nodes[‡].

After this step of computation v becomes a ghost node, and since its out-valence is zero we know that no action with target node v will appear in any stream. The set X_v contains residuals for any pair of actions with opposite polarities which appears in $(C)_v$: 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 the base in the source nodes appearing in the performed actions and this is in accord with definition of configuration isomorphism, see Definition 16, where sequences of actions in the isomorphic core streams are considered up to reordering as stated in case 2 of the definition. □

[‡] See Remark 10 where for any unit i the sub-stream of residuals selected in accord to the graph D_i is zipped to the core stream of the unit i: $C_i \ltimes (X_v)_{D_i}$.

5. Conclusions

In this paper, we introduced a stream-based class of abstract machines whose elementary step generalizes the half-combustion strategy for DVR. When considering the classical GoI setting, the proposed approach provides 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 intractable 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 (Asperti and Chroboczek, 1997) (cf. §2), based on safe operators, which employs a *sequential* safe-tagging algorithm. PELCR's ability to distribute *dynamically* the workload among the available processors displays *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. We finally remark that the paper also contains a first exploration of possible applications of PELCR to inputs structures different from virtual nets. Indeed an example of computation of the language recognized by an automaton and a coding of natural numbers are proposed. In future works, we plan to investigate possible further relations between PELCR and other models of computations.

References

Accattoli, B., Barenbaum, P., and Mazza, D. (2014). Distilling abstract machines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 363–376.

Asperti, A. and Chroboczek, J. (1997). Safe Operators: Brackets Closed Forever Optimizing Optimal lambda-Calculus Implementations. *Applicable Algebra in Engineering, Communication and Computing*, 8(6):437–468.

Asperti, A., Giovanetti, C., and Naletto, A. (1996). The Bologna Optimal Higher-order Machine. *Journal of Functional Programming*, 6(6):763–810.

Asperti, A. and Guerrini, S. (1998). *The optimal implementation of functional programming languages*, volume 45. Cambridge University Press.

Baillot, P. and Pedicini, M. (2001). Elementary complexity and geometry of interaction. *Fund. Inform.*, 45(1-2):1–31. Typed lambda calculi and applications (L'Aquila, 1999).

Canavese, D., Cesena, E., Ouchary, R., Pedicini, M., and Roversi, L. (2015). Light combinators for finite fields arithmetic. *Science of Computer Programming*, 111, Part 3:365 – 394. Special Issue on Foundational and Practical Aspects of Resource Analysis (FOPARA) 2009 – 2011.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., and Sarkar, V. (2005). X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM.

Cousineau, G. and Mauny, M. (1998). *The Functional Approach to Programming*. Cambridge University Press.

Curien, P.-L. (1990). The $\lambda\rho$-calculus: An Abstract Framework for Environment Machines. Technical report, LIENS–CNRS.

Danos, V., Pedicini, M., and Regnier, L. (1997). Directed virtual reductions. In *Computer science logic (Utrecht, 1996)*, volume 1258 of *Lecture Notes in Comput. Sci.*, pages 76–88. Springer, Berlin.

Danos, V. and Regnier, L. (1993). 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.

Danos, V. and Regnier, L. (1995). Proof-nets and the Hilbert space. In *Advances in Linear Logic*, pages 307–328. Cambridge University Press.

Fairbairn, J. and Wray, S. (1987). TIM : A Simple Lazy Abstract Machine to Execute Supercombinators. In Kahn, G., editor, *Proceedings of Conference on Functional Programming and Computer Architecture*, volume 274 of *Lecture Notes in Compututer Science*, pages 34–45. Springer-Verlag.

Girard, J.-Y. (1989). Geometry of Interaction I. Interpretation of system **F**. In *Logic Colloquium '88 (Padova, 1988)*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. North-Holland, Amsterdam.

Girard, J.-Y. (1990). Geometry of Interaction II. Deadlock-free algorithms. In *COLOG-88 (Tallinn, 1988)*, volume 417 of *Lecture Notes in Compututer Science*, pages 76–93. Springer, Berlin.

Girard, J.-Y. (1995). Geometry of Interaction III: Accommodating The Additives. In *Advances in Linear Logic*, pages 329–389. Cambridge University Press.

Gonthier, G., Abadi, M., and Lévy, J.-J. (1992). 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.

Hindley, J. R. and Seldin, J. P. (1986). *Introduction to combinators and λ-calculus*, volume 1 of *London Mathematical Society Student Texts*. Cambridge University Press, Cambridge.

Lamping, J. (1989). An algorithm for optimal lambda calculus reduction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–30. ACM.

Landin, P. J. (1964). The Mechanical Evaluation of Expressions. *Computer Journal*, 6(4):308–320.

Lévy, J.-J. (1978). *Réductions correctes et optimales dans le lambda-calcul.* PhD thesis, Université Paris VII.

Lévy, J.-J. (1980). Optimal reductions in the lambda-calculus. *To HB Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 159–191.

Mackie, I. (1995). The geometry of interaction machine. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 198–208. ACM.

Pedicini, M. (1998). *Execution et Programmes.* PhD thesis, Université Paris VII.

Pedicini, M., Pellitta, G., and Piazza, M. (2014). Sequential and Parallel Abstract Machines for Optimal Reduction. In Hage, J., editor, *Preproceedings of the 15th Symposium on Trends in. Functional Programming.*

Pedicini, M. and Quaglia, F. (2007). PELCR: parallel environment for optimal lambda-calculus reduction. *ACM Transactions on Computational Logic (TOCL)*, 8(3).

Pinto, J. S. (2001). Parallel Implementation Models for the Lambda-Calculus Using the Geometry of Interaction. In *Typed Lambda Calculi and Applications (Abramsky, S., editor)*, volume 2044 of *Lecture Notes in Compututer Science*, page 385399. Springer, Berlin.

Regnier, L. (1992). *Lambda-calcul et réseaux*. PhD thesis, Université Paris VII.

Rutten, J. J. M. M. (2005a). A coinductive calculus of streams. *Mathematical Structures in Computer Science*, 15(01):93–147.

Rutten, J. J. M. M. (2005b). A tutorial on coinductive stream calculus and signal flow graphs. *Theoretical Computer Science*, 343(3):443–481.

Solieri, M. (2016). Geometry of resource interaction and Taylor-Ehrhard-Regnier expansion: A minimalist approach. *Mathematical Structures in Computer Science*, pages 1–43.

Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the Association of Computing Machinery*, 33(8):103–111.

Valiant, L. G. (2011). A bridging model for multi-core computing. *Journal of Computer and System Sciences*, 77(1):154–166.