Mathematical Structures in Computer Science (2019), **29**, pp. 1379–1410
doi:10.1017/S096012951900001X

CAMBRIDGE UNIVERSITY PRESS

PAPER

Abstract machines, optimal reduction, and streams

Anna Chiara Lai[1], Marco Pedicini[2,*], and Mario Piazza[3]

[1]Department of Basic and Applied Sciences for Engineering, Sapienza University of Rome, Via Antonio Scarpa 16, 00181 Rome, Italy, [2]Department of Mathematics and Physics, Roma Tre University, Via della Vasca Navale 84, 00146 Rome, Italy, and [3]Classe di Lettere e Filosofia, Scuola Normale Superiore, Piazza dei Cavalieri 7, 56126 Pisa, Italy
*Corresponding author. Email: marco@iac.cnr.it

(Received 30 November 2015; revised 31 December 2018; accepted 31 December 2018; first published online 08 April 2019)

Abstract

In this paper, we propose and 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.

Keywords: Geometry of Interaction, Parallel Implementation, Functional Programming, Streams, Linear Logic, Curry-Howard Isomorphism

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 (Accattoli et al. 2014; Cousineau and Mauny 1998; Curien 1991; Fairbairn and Wray 1987; Hindley and Seldin 1986). From the vantage point of logic and proof theory in particular, it is well known 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 structurally analogous to β-reduction on λ-expressions, allowing for the mathematical description of abstract machines as executions of programs. Specifically, 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 (Lamping 1989; Lévy 1978, 1980) (see also Asperti and Guerrini 1998 for an overview of the topic). Other abstract machines (Mackie 1995; Pedicini 1998; 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, 2005b). Our main goal is to introduce a mathematical

model of computation oriented to the quantitative analysis and the optimisation of machines performing optimal reduction on parallel architectures. To this end, we begin by designing sequential abstract machine whose dynamics of execution relies on the algebraic properties of *dynamic monoids*, alias free inverse semigroups (see Lawson 1998). After which, we extend sequential execution to some degree of parallelisation and finally we restrict our investigation to computations based on GoI to prove the soundness of parallel execution with respect to the sequential case.

We recall that the virtual reduction (VR) (Danos and Regnier 1993) is a fine-grained way to achieve optimal reduction based on Girard's dynamic algebra Λ^*. 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. In particular, the *directed virtual reduction* (DVR) (Danos et al. 1997) is a variant of VR exploiting the original algebraic machinery of GoI by removing the added part of the algebra in Danos and Regnier (1993) while managing to avoid recompositions. The proposed sequential abstract machine is a generalisation to *arbitrary* dynamic monoids of DVR.

The extension of the abstract machine performing DVR to its parallel implementation leads to a formal definition of the PELCR (Parallel Environment for optimal Lambda-Calculus Reduction) engine, which is a parallel implementation of DVR described in Pedicini and Quaglia (2007) whose open source code is available online 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 such a communication phase is not strictly synchronised, and there are no synchronisation barriers like in the 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 mixture of models) to develop their parallel applications on particular platform. PGAS's programming style is a special parallelism, oriented to the partitioning of data sets. On the other hand, PELCR may 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 interpretation of the initial λ-term. Global addressing of the memory is obtained by using communication of virtual addresses via the libraries for Message Passing Interface (MPI) available on many computer architectures.

To conclude this introduction, 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), our representation may permit to assess quantitatively their differences and could impact PELCR itself. Moreover, it provides a grammar to describe extensions of λ-calculus oriented towards parallel execution, and to quantify the efficiency of their evaluation strategies (Allombert and Gava 2018). It is also worth stressing that the framework of GoI is flexible enough to deal with resource sensitive calculi (Solieri 2016) as well as with the implicit computational complexity of logical systems (Baillot and Pedicini 2001) or to model the complexity classes in pure abstract mathematical terms (Pedicini and Piazza 2018). In Canavese et al. (2014, 2015); (Cesena et al. 2012) 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 the distributed execution of arithmetic functions.

Contributions of the paper may be summarised as follows.

(a) A novel stream-based description of the implemented PELCR engine as a sequential abstract machine (Sections 2, 3, and 5).

(b) Definitions of the synchronous and the asynchronous parallel version of the sequential machine (Section 6).
(c) A proof of soundness of the parallelised versions with respect to the sequential one (Section 7).
(d) It is shown that the machines rely on a generalisation of DVR, from Girard's dynamic algebra to an arbitrary dynamic monoid.
(e) Novel encodings of two particular systems (other than untyped λ-calculus) into dynamic graphs, namely natural numbers and deterministic finite automata (Section 4).

The main advantage of reformulating the implemented parallel evaluation is that we can associate the very formal description with a mathematical setting. In this setting, it is then possible to study the qualitative properties of the parallel execution and to express them in quantitative terms. For instance, we have in mind non-determinism and probabilistic execution as well as distribution strategies based on quantitative measures, so as to furnish a conceptual tool for analysing performances with respect to different parallelisation strategies like in Pedicini and Quaglia (2002).

This paper is organised as follows. In Section 2, we first recall the behaviour of the PELCR implementation as originally introduced in Pedicini and Quaglia (2018, 2007) and then we reformulate its definition in a set-theoretic framework: this supplies a preliminary, informal description of our setting. In Section 3, we begin the formal definition of the abstract machine: we set the algebraic structure, i.e. the set of *polarised dynamic graphs*, representing the state of the device; then, we introduce the notion of *action*, which is a graph-oriented encoding of instructions. In general, a single action involves several elements (edges) of the state of the machine: such a set of edges is called *context* of an action. In Section 3.2, we furnish a formal definition of the context of an action and of the *elementary computational step*, which is the description of the interaction between an action and its context. Section 4 contains two extra logical examples: we provide an encoding of natural numbers and of finite automata in terms of dynamic graphs; we sketch how the elementary computational step may be a tool for computing the successor of a natural number; we prove how this tool can be used to compute the language of an automaton as well. Note that the presentation of these models is far from being a rigorous model of calculus: it simply aims to suggest potential non-standard applications of GoI machinery via our abstract machines. In Section 5, we use streams (and related operations on them) and SECD formalism to describe the (sequential) computational dynamics of the abstract machine. In Section 6, we extend such approach to both synchronous and asynchronous parallel architectures. In Section 7, we prove the soundness of the parallel computation with respect to the sequential one in the case of machines performing DVR. Finally, Section 8 presents our conclusions.

2. The Computational Behaviour of PELCR

Our goal here is to make the reader acquainted with the behaviour of the PELCR implementation that is reflected in the model of computation defined on streams (see Section 3.1). PELCR implementation relies on a particular strategy of DVR, called *half-combustion* in (Pedicini and Quaglia 2007), from which we extrapolate our description.

We begin by reporting the pseudocode of the PELCR implementation, which is a parallel interpreter for λ-terms based on GoI. In PELCR, many processing elements U_p cooperate to the evaluation of a single λ-term viewed as a dynamic graph. The single processing unit execution flow is sketched in Algorithm 1.

The main evaluation loop consists of the following phases:

1) buffering instructions transferred as messages from other processing units (line 4);
2) processing these instructions one by one (while-loop at lines 5–17);
   (a) extracting one instruction from the buffer of pending instructions (line 6);

Algorithm 1 Original pseudocode for a process U_p in PELCR as given in (Pedicini and Quaglia 2007).

```
1:  function U_p
2:      initialize()
3:      while not end_computation do
4:          ⟨collect all incoming messages and store them in incoming_p⟩
5:          while not empty(incoming_p) do
6:              ⟨extract a message m from incoming_p⟩;
7:              if m.target ∈ nodes_p then
8:                  for each edge e ∈ nodes_p(m.target).combusted do
9:                      ⟨compose the edge carried by m with e⟩;
10:                     ⟨select the destination process U_j for hosting the node originated by the composition⟩;
11:                     ⟨send the edges produced by the composition to U_k and U_h hosting m.source and e.source,
                          respectively⟩
12:                 end for
13:             else
14:                 ⟨add m.target to nodes_p⟩;
15:             end if
16:             ⟨add the edge carried by m to nodes_p(m.target).combusted⟩
17:         end while
18:     end while
19: end function
```

(b) for each edge already hosted by the same target node of the extracted instruction (for-loop at lines 8–12);
  –i– composing the edge with the extracted instruction (line 9)
  –ii– deciding where the new node originated by residuals of the composition must be allocated (line 10);
  –iii– sending residual edges as instructions to the processing units hosting their respective target nodes (line 11).

The description of this process amounts to an *evaluation protocol* activated during the computation. This protocol is designed to express the order of execution of operations like receiving/sending messages, loading/storing elements in the memory, scheduling of operations. We remark that the evaluation protocol is irrespective of the soundness with respect to the λ-calculus: rather, it is the theory on which PELCR is based on, namely (directed) VRs, which is on charge of the soundness of the computation. Indeed, the link with DVR (therefore with GoI and ultimately with the λ-calculus) is not manifest until the composition step (line 9), which involves an algebraic product in the specific structure introduced for the GoI.

To sum up, the configuration of the machine has two components: the *buffer* of pending messages and the *graph* of already used edges. The behaviour of the processing units (described in the algorithm) consists in receiving a message, computing residuals of one step of DVR between the edge carried by the message and all the edges already hosted on the same node in the graph, and finally buffering residuals to the buffer of pending messages.

2.1 PELCR execution: a set-theoretic view

An informal set-theoretic reformulation of the evaluation protocol of execution will be useful when it will come to extracting the basic notions for the machines presented formally in Sections 3.1 and 3.2. Section 3.1 deals with the formal definition of the algebraic structure concerning memory and instructions, Section 3.2 gives the elementary computational step associated with an instruction (half-combustion step), and Section 5 illustrates the algebraic version of the communication layers between components.

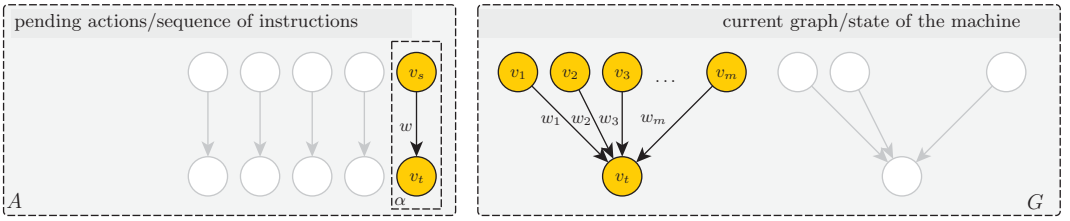

Figure 1. Graphical representation of the machine configuration *(A,G)* ready to apply the evaluation protocol after the choice of the action α.

Roughly speaking, instructions are specifications of (polarised labelled) edges called *actions*, concatenated to form the flow of control. A PELCR processing unit receives a flow of actions from other units (possibly the same unit) and processes them one by one, operating on the current state which is represented by a graph. Thus, at any evaluation step a machine configuration is a pair (A, G) (see Figure 1) such that:

— A is a set of *pending actions* (on the left of Figure 1),
— G is a (dynamic) graph (on the right part of Figure 1).

In Figure 1, the action α and the node v_t involved in the description of the elementary computational step below are highlighted, whilst the other actions of A and nodes of G are shadowed.

What is described in the evaluation protocol is the set-theoretic version of the abstract machine transition (formally defined in Section 3.2). Here, this transition consists in applying an action α, taken from A, to the current graph G.

In fact, the action α alone conveys the information required to transform the graph and coinciding with the edge description. To perform the transition associated with α, we have to compute two parts:

— by accessing the target node of the edge, the computational payload is expressed by the interaction of α with its *context* (any edge in G with the same target node). We call this *one-to-many* interaction the *elementary computational step*, whose outcome is a set Δ_α of new instructions to be executed, called *residual actions* of α;
— after the elementary computational step, the set A is updated by replacing α with the residual actions Δ_α, while the graph G is updated adding the edge carried by α.

In short, by the evaluation protocol, a transition $(A, G) \to (A', G')$ consists in:

1) choosing an element $\alpha \in A$ (line 6 in the Algorithm 1);
2) performing the elementary computational step by computing the set Δ_α (while-loop at lines 8–12);
3) defining A' and G' by adding Δ_α to A and α to G, respectively (lines 14 and 16), which is

$$A' = A \setminus \{\alpha\} \cup \Delta_\alpha \text{ and } G' = G \cup \{\alpha\}.$$

The above transition is an intuitive description of a step of the half-combustion strategy of DVR (see Pedicini and Quaglia 2007): it includes (in the elementary computational step) symbolic computations in the algebraic structure associated with the graph.

For the sake of perspicuity, in Section 3.1, we display in two steps the structure given in the original GoI definition: we first present the notion of *dynamic monoid* (a (sub)structure of Girard's *dynamic algebra*) through which we fix then the minimal set of operations and axioms required

for the definition of the evaluation protocol based on half combustion of DVR. Namely, such a dynamic monoid satisfies the *stable form condition* (Definitions 1 and 2). Note that by requiring the further properties which extend a dynamic monoid to the dynamic algebra, we recover as an *extra bonus* a setting for the evaluation of λ-calculus and GoI which is sound with respect to normalisation.

In such a set-theoretic setting, a computation starts with the empty graph initialising the machine's memory and the execution of a terminating program can be summarised by a finite sequence of transitions

$$\left(A^0, \emptyset\right) \longrightarrow \left(A^1, G^1\right) \longrightarrow \cdots \longrightarrow (A^n, G^n) \longrightarrow \left(\emptyset, G^{n+1}\right). \tag{1}$$

The *initial action set* A^0 plays the role of the program executed by the machine, whilst the final graph G^{n+1} represents the result of the evaluation. Following the GoI terminology, we say that the final graph G^{n+1} is the *execution formula* of the initial sequence A^0. *De facto*, in our formulation the final graph *contains* the execution formula as given in Girard (1989) together with any edge used during the computation, since no edge is removed from the graph of used edges at any time. Actually, PELCR implements a strategy for removing from the graph edges that are not necessary in successive steps of computation; this mechanism can be considered as a kind of garbage collection, but it is not analysed in the present paper, we just sketch the solution envisaged in PELCR, in Remark 9.

In some cases, the *termination condition* (given by the emptiness of the set A^i, for some i) could never occur during the computation, leading to an infinite sequence of transitions. Non-terminating cases justify the crucial role of the streams: if we conceive of PELCR as a device which performs a computation and produces a result, then this result should be usable as an input for a further computation. This general property is assured in GoI by the so-called *associativity of the execution formula*. If we combine associativity with the local and asynchronous version of the GoI represented by DVR (which enables the parallelism in PELCR), then we get the possibility of starting the computation of the execution from the result of another execution, even when the former is unfinished, that is, let us say, from a *finite approximation*. In those cases where the first computation does not terminate, the second computation gets an infinite input. Such an ability to process infinite inputs well motivates the introduction of streams of actions.

Let us conclude the informal presentation of PELCR's computational behaviour by rephrasing Algorithm 1 in terms of the set-theoretic notation just introduced (see Algorithm 2). This should put the reader in a better position to appreciate the connection between the PELCR evaluation protocol and the formal presentation of abstract machines in the next sections. Note that the pseudocode represents the evaluation protocol that the unit has to follow while concurrently evaluates a λ-term with other units.

3. Sequential Abstract Machines

We start now a formal description of a family of abstract machines whose functioning relies on algebraic requirements satisfied by the PELCR evaluation machine. Moreover, these requirements can be applied to the design of various evaluation models in principle executable through our abstract machines. We show in Section 4 an application to automata.

3.1 The state of the machine: polarised dynamic graphs

Our aim here is to rearrange notions from GoI and DVR to shape a new presentation of the algebraic setting for the abstract machine. Its processing unit is a universal device which consumes a pipeline of elementary instructions, while producing further instructions to be processed. The machine's memory is represented by a *dynamic graph*, i.e. a graph characterised by some algebraic

Algorithm 2 Original pseudocode for a process U_p rephrased in accord to the set-theoretic presentation of PELCR.

```
1: function U_p
2:     initialize()
3:     while not end_computation do
4:         ⟨ collect all incoming messages and store the corresponding actions in A_p⟩;
5:         while A_p ≠ ∅ do
6:             ⟨take an α from A_p ⟩;
7:             if target(α) ∈ G then
8:                 for each edge β_i ∈ target(α) in G do
9:                     ⟨perform the interaction between α and β_i⟩;
10:                    ⟨select the destination process U_j for hosting the node originated by the composition⟩;
11:                    ⟨send the edges produced by the composition to U_k and U_h hosting source(α) and source(β_i)
                         respectively⟩
12:                end for
13:            else
14:                ⟨update the part of G hosted on U_p by adding the node target(α)⟩;
15:            end if
16:            ⟨update the part of G hosted on U_p by adding α ⟩
17:        end while
18:    end while
19: end function
```

properties of its labels, which are assumed to be taken in a *dynamic monoid* (see Definition 1). Moreover, the instructions to be executed are edges and their execution is given by the transformation of the graph in memory and the production of a sequence of new instructions to be executed.

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 with 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 Σ is the set Σ^* whose elements are all the finite sequences of zero or more elements from Σ, with sequence concatenation as the monoid operation.

Definition 1. *A* dynamic monoid *over the alphabet Σ is the free monoid M generated by Σ 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, i.e. $0^* = 0$, $1^* = 1$ and $(a^*)^* = a$ and $(ab)^* = b^*a^*$, for any $a, b \in M$).*

Any free inverse semigroup is indeed a dynamic monoid as pointed out in Lawson (1998, chapter 6).

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 $(\cdot)^*$. 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*

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

If SFC holds and it is computable (in linear time), we may perform computation by means of DVR. However, while a dynamic monoid satisfying SFC suffices 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* Girard dynamic algebra Λ^* *is the dynamic monoid generated by the constants p, q, and a family* $W = \{w_i\}_{i\in\mathbb{N}}$ *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,

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

Note also that annihilation 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 commutation rule we get

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

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* $G = (V, E \subset V^2 \times M)$ *labelled with elements of 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 each edge e is represented by a triplet* $((\varepsilon_t, \varepsilon_s), (v_t, v_s), w)$*, 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 played by the edges with respect to the execution of the abstract machines, we define the *actions* in terms of graph transformation instructions. Any action has as a payload information concerning an edge to be added to the current dynamic graph. It is then possible to compute the composition with other edges via interaction rules producing new instructions from residual edges.

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

Definition 6 (Node reference). *Given a sequence of graphs* $(G_i)_{i\in I}$ *where* $G_i = (V_i, E_i)$*, by* reference *(to a node) we mean an injective map* ρ *from the set of all nodes* $\bigcup_{i\in I} V_i$ *to integers.*

Observe that a reference of a node is determined by a function ρ which depends on a sequence of graphs by definition. If we abstract from the sequence, we obtain a function defined on the set of the nodes of all possible graphs.

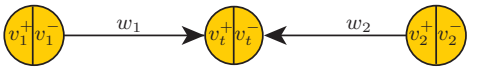

Figure 2. A polarised graph.

Definition 7 (Set of actions). *The set A_M of all possible* polarised actions *on M is a set of 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 fundamental transformation that any instruction represents is the addition of nodes and edges to the graphs, as we show in Section 5.1. It is possible for actions to make reference to nodes which 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 may happen that one edge has source node on a unit and the target node on another. In such a 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 distinguish graphically between a node v and references to that node $\rho(v)$.

Notation 1. *Polarisation induces a bipartition of edges coincident on the same node v in two sets of edges with the same polarity. We denote the two sets by v^+ and v^- according to their respective polarities and we call v^+ (resp. v^-) the* positive seminode *(resp.* negative seminode*) of v.*

Example 1. Consider the polarised dynamic graph $G \in \mathbb{G}_M^+$ in Figure 2. We have $G = (V, E)$ where $V = \{v_1, v_2, v_t\}$. The edges of G are

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

Finally we notice that the seminodes associated with 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 (Danos and Regnier 1995; Regnier 1992), this term is translated in a linear logic proof net, represented in Figure 3.

Indeed, the term $(\Delta\, I)$ may be typed in linear logic by extending the type system with a fixed 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 interpretation, that is this matrix with entries in the Girard dynamic algebra:

$$\begin{array}{c} \\ [\mathrm{ax}]_1 \\ [\mathrm{ax}]_2 \\ [\mathrm{ax}]_3 \\ [\mathrm{ax}]_4 \\ [\mathrm{cut}]_1 \\ [\mathrm{t}]_1 \end{array} \begin{array}{c} \begin{array}{cccccc} [\mathrm{ax}]_1 & [\mathrm{ax}]_2 & [\mathrm{ax}]_3 & [\mathrm{ax}]_4 & [\mathrm{cut}]_1 & [\mathrm{t}]_1 \end{array} \\ \left(\begin{array}{cccccc} 0 & 0 & 0 & 0 & qw_2 + qw_1q & 0 \\ 0 & 0 & 0 & 0 & qw_1p + p & 0 \\ 0 & 0 & 0 & 0 & q!q + q!p & 0 \\ 0 & 0 & 0 & 0 & p & 1 \\ w_2^*q^* + q^*w_1^*q^* & p^*w_1^*q^* + p^* & (!q^*)q^* + (!p^*)q^* & p^* & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{array}\right) \end{array}.$$

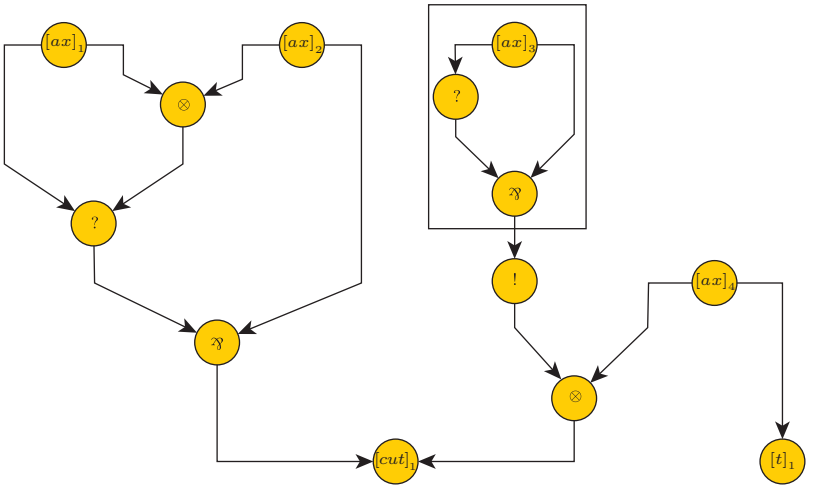

Figure 3. Proof net of $(\Delta\ I)$.

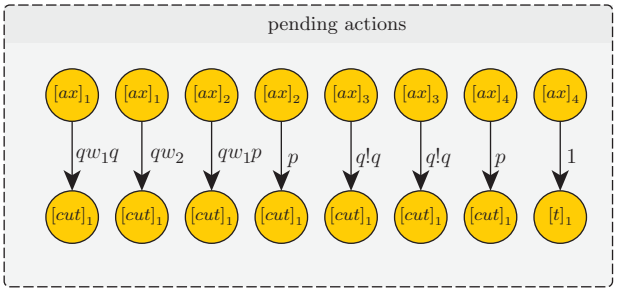

Figure 4. Initial list of pending actions for the computation of the Execution Formula of $(\Delta\ I)$.

The matrix representation is redundant (the matrix is in a way symmetric $a_{ij} = a^*_{ji}$, we would say *hermitian* in some model for Λ^*) and sparse. This drawback was one motivation in Danos and Regnier's (1993 paper) for considering *VR* using a graph as a notation for the sparse matrix. Hence, 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 $\langle (v_t, v_s), w\rangle$ 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:

$$\begin{aligned} E = \{&(([\mathrm{cut}]_1, [\mathrm{ax}]_1), qw_1q), (([\mathrm{cut}]_1, [\mathrm{ax}]_1), qw_2), (([\mathrm{cut}]_1, [\mathrm{ax}]_2), qw_1p),\\ &(([\mathrm{cut}]_1, [\mathrm{ax}]_2), p), (([\mathrm{cut}]_1, [\mathrm{ax}]_3), q!q), (([\mathrm{cut}]_1, [\mathrm{ax}]_3), q!p),\\ &(([\mathrm{cut}]_1, [\mathrm{ax}]_4), p), (([\mathrm{t}]_1, [\mathrm{ax}]_4), 1)\}. \end{aligned} \tag{3}$$

To compute the Execution Formula associated with the term, we initialise the list of pending actions with such an edge set (see Figure 4), and we proceed with 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).

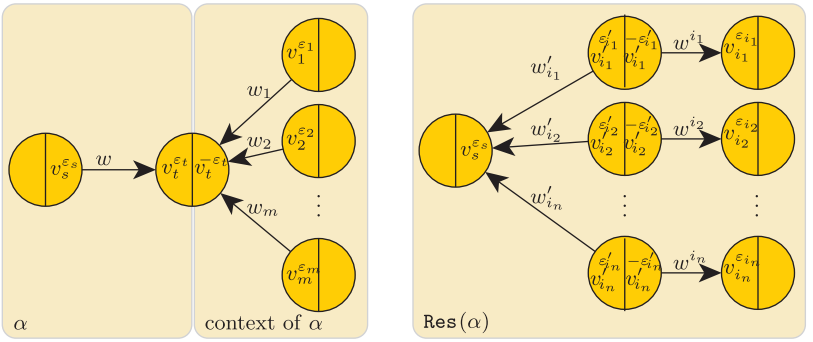

Figure 5. Elementary computational step (as half-combustion). Note that the computation of m interactions originates $2n$ residual edges, with $n \leq m$.

3.2 The elementary computational step: generalized half-combustion

Definition 8 (Context). *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. By* context *of α, we mean the set of all edges $\{\beta_1, \ldots, \beta_m\}$ of G such that*

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

In other words, the context of α is the set of edges belonging to the seminode $v_t^{-\varepsilon_t}$, namely the 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 each edge are elements of a dynamic monoid M, i.e. $w = a$ and $w_i = b_i$ for some $a, b_i \in M$.

Definition 9 (DVR step). *We define the result of the step of DVR between the action α and the edge β_i when $b_i^* a \neq 0$, and $a_i' b_i'^*$ is its stable form, as the new actions*

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

where ε is arbitrarily chosen in $\{+, -\}$ and v_i' is a newly generated node.

Note that the node v_i' is a new node to be added to the graph together with its outgoing edges (v_i, v_i') and (v_s, v_i'). Therefore, the new edges α_i and β_i' represent the new payload of the computation and are the forthcoming actions, as we see in the next definition.

The *elementary computational step* associated with the action α consists in the computation of the set of residuals of the action with respect to its context in the graph G. Therefore, it involves the computation of pairwise action-edge interactions as in Figure 5, and in collecting actions produced by the DVR steps:

Definition 10 (Residuals of an action). *Given an action α and a polarised dynamic graph G, we define the set of all residual actions originated by α with respect to G as the set*

$$\mathtt{Res}_G(\alpha) := \{\alpha_{i_1}, \beta_{i_1}', \ldots, \alpha_{i_n}, \beta_{i_n}'\} \tag{5}$$

where $\{\beta_1, \ldots, \beta_m\}$ is the context of α in G and, for each i_j such that $a_{i_j}' b_{i_j}'^* \neq 0$, the pair of actions $(\alpha_{i_j}, \beta_{i_j}')$ is the result of the DVR step between α and β_{i_j}.

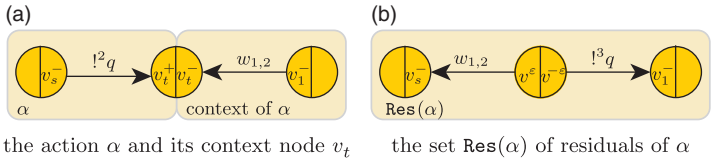

Figure 6. Elementary computational step of an action α acting on a context with one edge.

Remark 2. Since the number of residual pairs is possibly less than the number of edges β_i in the context, 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 $M = \Lambda^*$, $a = !^2 q$ and $b = w_{1,2}$. Let $\alpha = \langle (+,-), (v_t, v_s), a \rangle$ and assume that its context is $\{\beta\}$, namely the singleton containing the edge $\beta = ((-,-), (v_t, v_1), b)$. Recall from Equation (2) that $b^* a = a' b'^*$ where $a' = !^3 q$ and $b' = w_{1,2}$. We have in Figure 6, the elementary step of computation associated with the action α with respect to the context $\{\beta\}$ which produces the set of actions $\mathtt{Res}(\alpha)$.

Remark 3 (The role of polarity in DVR). In the DVR procedure, introduced in Danos et al. (1997), 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, each 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. 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_i followed by a_j^*, which is a distinct premise a_j of the same link (with $i \neq j$) taken in the reversed direction. 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, so as 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. Such a mechanism is automatised by restricting the computation to those edges with opposite target polarities. In the more general framework 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.

4. Examples

We show a pair of applications that can be embedded in the algebraic setting of dynamic monoids and computed through the PELCR evaluation protocol. The purpose of these examples is twofold: to give evidence that the machine we are going to define may be used in a wider range of situations and not only in the λ-calculus evaluation, and to support the interest in the parallel version of the machine, which is irrespective of the algebraic specification.

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 7(a), whereas in Figure 7(b) we show the representation of the successor of n. We illustrate how to obtain, through the same GoI reduction

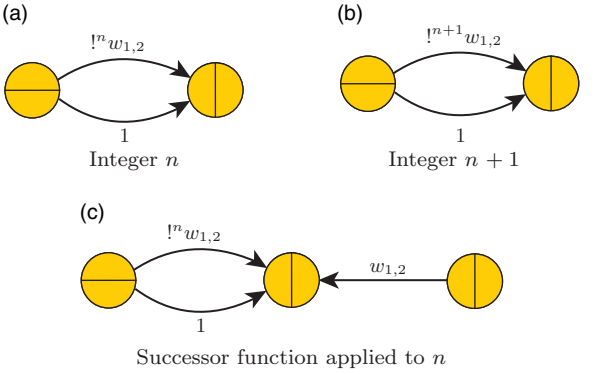

Figure 7. Polarised dynamic graphs representing the two integer numbers n and $n+1$, and the successor function applied to n.

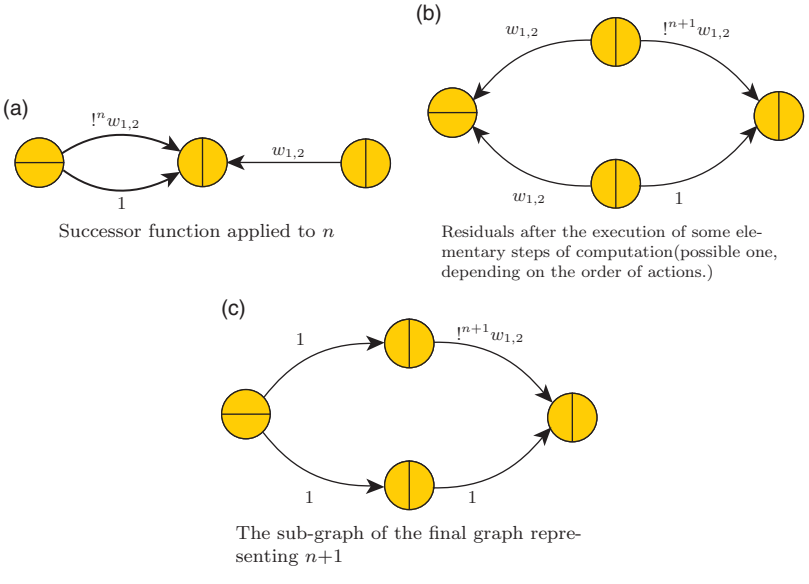

Figure 8. The pending actions represented in (a) and (b) give rise to a part of the polarised dynamic graphs representing intermediate states during the reduction of the successor function applied to an integer number n. The graph in (c) is the part of the final graph which corresponds to the result of the computation.

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 split nodes in two sets separated by a line (not necessarily vertical) in the figures.

The application of the successor function to n is represented by the interaction between $G_{\Lambda^*}(n)$ and the action labelled with $w_{1,2}$ (note that in GoI this interaction represents the reduction of a commutative cut). Figure 7c displays the application of the successor function to the integer n, whilst

Figure 8 shows two groups of pending actions ((a) and (b)) which at different time steps trigger an elementary computational step.

In Figure 8c, it is depicted the sub-graph of the execution formula (i.e. the final graph) of the successor of n that represents $n+1$. The reader may notice that the sub-graph is not exactly the representation of $n+1$ as given in Figure 7b, containing identity labelled edges not appearing in the definition. To overcome this type of problem, however, we resort to a modified interaction treated in Pedicini and Quaglia (2007) and called *optimisation of one* (see Figure 9a). In the case of residuals labelled by identity, such an interaction works by creating only one residual instead

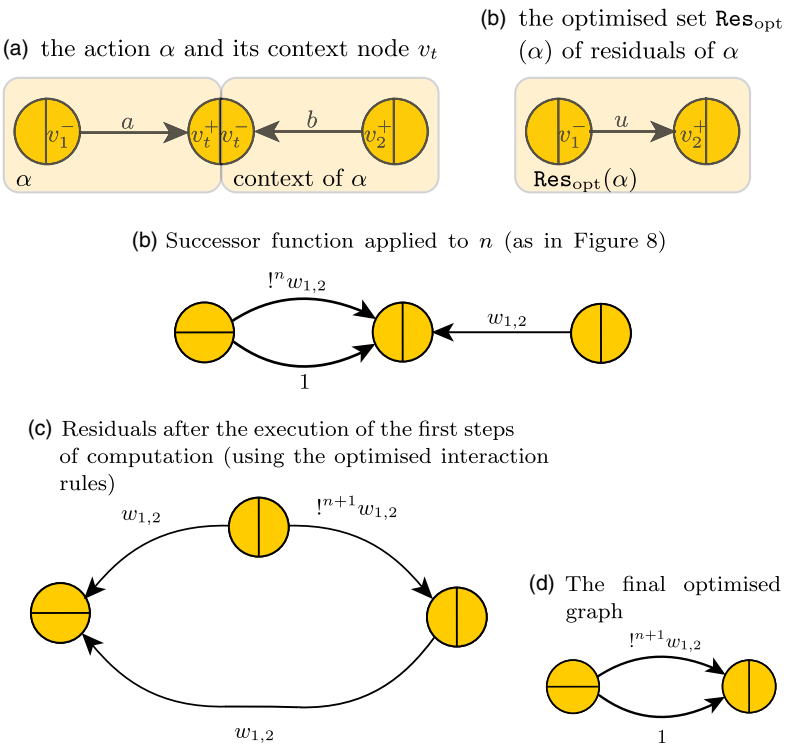

Figure 9. Optimisation of one. Optimised interaction rule: the elementary computational step (for the dynamic graph on the left) in general (if the product of the labels involved in the interaction is different from 0) includes the creation of a new source node v_s and of a pair of edges targeting the 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).

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 the evidence 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 an encoding into dynamic graphs for deterministic finite-state automata. This approach is informally justified by taking into account the particular case of paths preserved by reduction. The task of this example is to offer some clues about non-standard applications of the GoI machinery; however, the rigorous discussion of the general case is postponed to 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. Note that we assume by definition that T is a total correspondence, whereas some authors admit partial transition functions making the correspondence of non-accepting state if the undefined transition occurs.

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

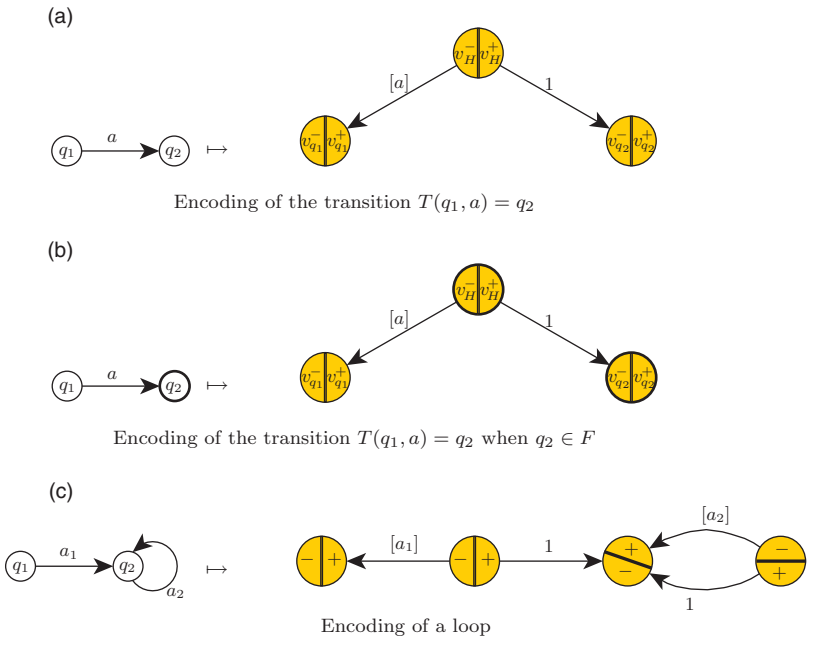

Figure 10. Graph representations of some automata and their encoding as dynamic graph.

Definition 11. *For any automaton $\mathcal{A}$ let us define its dynamic graph $[\mathcal{A}] := G \in \mathbb{G}^{+}_{\Lambda^*}$ equipped with additional information (v_0, V_F); the polarised dynamical graph G has nodes in $V_Q \cup V_H$,*

- *each state $q \in Q$ is associated with a vertex $v_q \in V_Q$ and in particular the initial node q_0 is encoded into the vertex v_0; final states $q \in F$ are encoded as nodes of the set $v_q \in V_F \subset V_Q$;*
- *each element of the alphabet $a_i \in A$ is encoded as the element $!w_i$ of Λ^* with lift $l(w_i) = 1$ for any i; we adopt the same notation as for the encoding of automaton itself, that is $[a_i] :=!w_i$;*
- *for any transition $T(q_1, a) = q_2$ two edges belong to the graph*

$$e_1 = ((+, -), (v_{q_1}, v_H), [a]) \quad e_2 = ((-, +), (v_{q_2}, v_H), 1)$$

where $v_H \in V_H$ is one of the auxiliary nodes associated univocally with the transition, shown in Figure 10(a);
- *source nodes of identity labelled edges with target node in V_F belong to V_F too.*

The evaluation of the graph proceeds by sequences of elementary steps of computation starting from the pair $([\mathcal{A}], \emptyset)$, as in Equation (1); moreover, the set of final nodes V_F is updated as prescribed in Figure 11. For any pair of edges involved in a DVR step, we have that they have common target node, the two edges have opposite polarities and one has weight 1 and the other one has weight $[a]$ for some $a \in A$. Whenever the source node of the edge which is not labelled by the identity belongs to V_F^t, then we have that the new source node $v_h \in V_F^{t+1}$ (i.e. the set of final nodes is increased with the new node v_h). This dynamical annotation of the set of final nodes is important in the statement and proof of Theorem 1 which the definition of the read-back mapping crucially depends on.

As an example, consider the automaton $\mathcal{A}_{\texttt{even}}$ in Figure 12 and its encoding according to the rules in Figure 13.

We denote by

$$L_{\mathcal{A}} = \{u \mid u \in A^* \text{ accepted by } \mathcal{A}\},$$

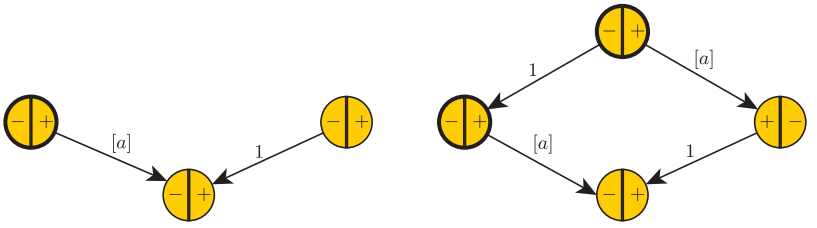

Figure 11. The rule for dynamically updating the set V_F of final nodes (here, represented by bold shaped nodes) before (on the left) and after (on the right) performing one interaction.

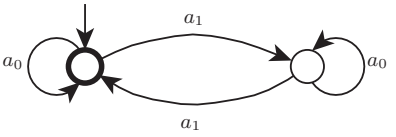

Figure 12. The automaton $\mathcal{A}_{\text{even}}$ of alphabet $\{a_0, a_1\}$ which decides the language of words containing an even number of a_1's.

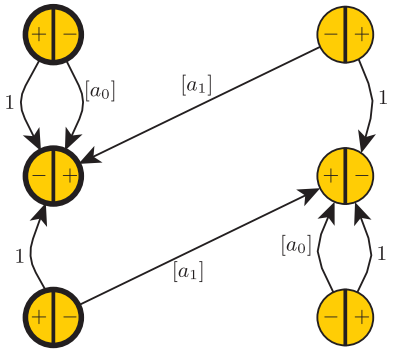

Figure 13. Dynamic graph $[\mathcal{A}_{\text{even}}]$ encoding the automaton $\mathcal{A}_{\text{even}}$ in Figure 12.

for any automaton $\mathcal{A}$ of alphabet A. With this notation, we have that $L_{\mathcal{A}_{\text{even}}}$, the set of words accepted by the automaton given in Figure 12, is the set of words containing an even number of a_1's.

We showed the encoding function of any automaton $\mathcal{A}$ as a dynamic graph $[\mathcal{A}] \in \mathbb{G}^{+}_{\Lambda^*}$ now we need to specify how accepted words can be read-back from a dynamic graph during the evaluation. Computation starts from the pair $([\mathcal{A}], \emptyset)$, which is the pair given by the set of edges corresponding to the encoding of the automaton $\mathcal{A}$ and the empty graph; the initial set of final nodes is also required to perform the evaluation.

Definition 12. *For any automaton $\mathcal{A}$ of alphabet A, let us consider the dynamic graph $[\mathcal{A}]$. Then*

1) *A path in $[\mathcal{A}]$ is a sequence of edges $\varphi = \alpha_1\alpha_2 \ldots \alpha_n$ such that $\alpha_i = v_i \xrightarrow{a_i} v_{i+1}$ for any $1 \leq i \leq n-1$. Note that an edge can be regarded as a path of length 1.*
2) *For any path $\varphi \in [\mathcal{A}]$, we define the word $[\varphi]^{-1} \in A^*$ as the concatenation of the preimages $[\alpha_i]^{-1}$ of the algebraic weights a_i associated with each edge α_i.*

Note that $[\alpha_i]^{-1}$ is the preimage of the algebraic label of the edge $\alpha_i = v_i \xrightarrow{a_i} v_{i+1}$, i.e. $u_i \in A$ if $[u_i] = a_i$ (by definition, the label of an edge with algebraic label 1 is $[1]^{-1}$, i.e. the empty word).

Now we are in position to state the soundness and completeness of the computation of $L_{\mathcal{A}}$ with respect to the execution of the abstract machine: launched by the interpretation of the automaton, the machine generates each word in $L_{\mathcal{A}}$ as the word $[\varphi]^{-1}$ associated with some path φ starting

in a node $v \in V_F^t$ (for some $t \in \mathbb{N}$) and ending in v_0 (the node corresponding to the initial state). Through this approach, we may regard the computation of the language decided by an automaton as the model, while the abstract machine is the syntactic object performing the computation which starts from the automaton interpretation as a dynamic graph (to be executed).

Theorem 1. *Let be $\mathcal{A}$ an automaton of alphabet A.*

a) (soundness) For any $t \in \mathbb{N}$, any path φ starting in a node $v \in V_F^t$ and reaching v_0 is such that $[\varphi]^{-1} \in L_{\mathcal{A}}$.
b) (completeness) For any $u \in L_{\mathcal{A}}$, there exists a t such that there exists a sequence of elementary steps of computation applied to the initial configuration $([\mathcal{A}], \emptyset, V_F)$, so that it reaches a configuration (A_t, G_t, V_F^t) containing a path $v \xrightarrow{\varphi} v_0$ with $v \in V_F^t$ such that $[\varphi]^{-1} = u$.

Proof. We prove soundness by induction on the length of a path $v_q \xrightarrow{\varphi} v_0$, where $v_q \in V_F^t$ for some t; the thesis to be proved is that $[\varphi]^{-1} \in L_{\mathcal{A}}$.

We employ same notations for a configuration (A_t, G_t, V_F^t) at step t; therefore, A_t is the set of actions. For what concerns the basis of induction we consider a φ of length 1: if $[\varphi]^{-1}$ is the empty word, then the base of induction trivially follows. We then discuss the case $[\varphi]^{-1} = a$. We show that the smallest time step t_1 at which $v_q \in V_F^{t_1}$ must be the initial one, then in such case being $[\varphi]^{-1} = a$ the statement is proved since by definition q is a final state: $a \in L_{\mathcal{A}}$.

Let us prove now that assuming $t_1 > 0$ yields a contradiction. To this end, note that $t_1 > 0$ implies that v_q is a newly generated node by the reduction of some pair of actions

$$\alpha := \langle (-,+), (v'_q, v_{q_0}), 1 \rangle, \ \beta := \langle (+,-), (v'_q, v_*), [a] \rangle \in A_{t_2},$$

with $v_* \in V_F^{t_2}$, for some $t_2 < t_1$. This implies that v_{q_0} is source node of some action of the initial encoding (sources of residuals are always new nodes) and this is not possible by construction of the encoding of the automaton.

Now assume as inductive hypothesis that for any given automaton $\mathcal{A}$ if $v_q \xrightarrow{\varphi} v_0$ with $v_q \in V_F^t$ for some t and $|\varphi| = n$, then $[\varphi]^{-1} \in L_{\mathcal{A}}$. Let us consider a path of length $n+1$, namely $[\varphi]^{-1} = au$ with $v_q \xrightarrow{\varphi} v_0$ with $v_q \in V_F^t$ for some t and for some $a \in A$ and some $u \in A^n$. Let q_a be such that $T(q_0, a) = q_a$ and note that then the last edge of φ necessarily connects v_{H_a}, the auxiliary node associated with the transition $T(q_0, a)$, to v_{q_0}. Also note that the the action $\alpha_a := \langle (-,+), (v_{q_a}, v_{H_a}), 1 \rangle$, which, by construction, belongs to the encoding of $\mathcal{A}$, enjoys the following path-preserving property. If $v \in V_F^t$ and $v \xrightarrow{\varphi'} v_{q_a}$, then the interaction between α_a and all the edges in φ' yields a path φ'' from a final state $v' \in V_F^{t_1}$, with $t > t_1$, to v_{H_a} with the same read-back $[\varphi']^{-1} = [\varphi'']^{-1}$. Up to a reordering of the actions, we can assume that for some $t_1 < t$ we have a path $v_q \xrightarrow{\varphi'} v_{q_a}$ whose read-back is u and that the action $\alpha_a := \langle (+,-), (v_{q_a}, v_{H_a}), 1 \rangle$, which, by construction, belongs to the encoding of $\mathcal{A}$, has not yet been consumed, namely $\alpha_a \in A_{t_1}$.

Now take the automaton $\mathcal{A}_a := (T, q_a, F)$, whose language is $L_{\mathcal{A}_a} := \{u \mid au \in L_{\mathcal{A}}\}$. We then can see that φ' is a path of length n from a final state of $[\mathcal{A}_a]$ to its initial state; hence, by inductive hypothesis $u = [\varphi']^{-1} \in L_{\mathcal{A}_a}$. Now we are done, since by construction $au = [\varphi]^{-1}$ and by definition of $L_{\mathcal{A}_a}$, $au \in L_{\mathcal{A}}$.

We show completeness by induction: an accepted word of length 1 is such that $u \in L_{\mathcal{A}}$ and $T(q_0, a) = q \in F$ where $u = a$, then by definition in $[\mathcal{A}]$ we have an edge (which is a path of length 1) $\varphi = v \xrightarrow{[a]} v_0$ and $v \in V_F^t$; therefore, $[\varphi]^{-1} = a$.

By induction hypothesis, we know that the property holds for any automaton and any word of length n. We consider a word $u' = au \in L_{\mathcal{A}}$ with length of u equals to n, and the state $q = T(q_0, a)$, then we have that the word $u \in L_{\mathcal{A}_q}$ with $\mathcal{A}_q = (T, q, F)$, i.e. with same transitions and final states of $\mathcal{A}$ but with initial state q; therefore, there exists a path $v \xrightarrow{\varphi} v_q$ such that $[\varphi]^{-1} = u$. Since the encoding of the transition $q = T(q_0, a)$ is given by the two edges

$$e_1 = ((+, -), (v_0, v_H), [a]) \quad e_2 = ((-, +), (v_q, v_H), 1)$$

we have that e_2 interacts with any edge of the path φ starting in v and ending in v_q.

At the end of this sequence of interactions, the new path starting in the node v' produced by the interaction of the last edge of φ is a path from the final node v' to v_0 and it is labeled as $a[\varphi]^{-1} = au$. □

5. Streams

In what follows we resort to streams to distribute the computational load on many devices and this means that we are to define streams of *actions*. For any given set A, we regard A as the set of all possible actions the computational device can perform. To make the exposition of the execution equivalence in Section 7 easier, we consider $\mathbf{A}$ as the set of formal sums of elements of A. In particular, there exists a null element (the empty sum) $\mathbf{0}$, such that $\mathbf{0} + \alpha = \alpha$. Following Rutten (2005), we give the following:

Definition 13. *A* stream *S on A is a sequence $S : \mathbb{N} \to A$ of elements of A. We define the set $\mathbf{A}^\omega$ of all streams as $\mathbf{A}^\omega := \{S \mid S : \mathbb{N} \to \mathbf{A}\}$.*

For a stream S, we call $S(0)$ the *initial value* of S and we adopt the following notations: if $S = (S(0), S(1), S(2), \dots)$, then

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

and nil as the stream defined by the equation

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

A *finite stream* is a stream eventually coinciding with nil.

We consider the following operations on streams.

Definition 14 (Shift and zip). *The* shift *operation (also called* derivative *or* tail*) is defined by*

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

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

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

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

Remark 4. Note that by definition

$$\mathtt{zip}(S, T)(2i) = S(i),$$
$$\mathtt{zip}(S, T)(2i + 1) = T(i)$$

for all $i \in \mathbb{N}$. Also note that $\mathtt{zip}$ is neither commutative nor associative: in general $S_1 \ltimes (S_2 \ltimes S_3) \neq (S_1 \ltimes S_2) \ltimes S_3$. In what follows, we adopt the following notation $S_1 \ltimes S_2 \ltimes S_3 := (S_1 \ltimes S_2) \ltimes S_3$.

We give a rather obvious coinductive definition of sub-stream as follows:

Definition 15 (Sub-stream). *S_2 is a* sub-stream *of S_1 if one of the two conditions holds:*

- *either $S_2(0) = S_1(0)$ and $\mathtt{shift}(S_2)$ is a sub-stream of $\mathtt{shift}(S_1)$,*
- *or $S_2(0) \neq S_1(0)$ and S_2 is a sub-stream of $\mathtt{shift}(S_1)$.*

A *proper sub-stream* S_2 of S_1 is a sub-stream S_2 of S_1 which is different from S_1.

Definition 16 (Strip). *If S_2 is a proper sub-stream of a stream S_1, then the* strip of S_2 from S_1 *is*

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

Definition 17 (Weak-bisimulation and weakly bisimilar streams). *A* weak-bisimulation *on A is a relation $\gamma \subset \mathbf{A}^\omega \times \mathbf{A}^\omega$ such that, for all streams S and T on A, if $(S, T) \in \gamma$ then following holds:*

$$S(0) = T(0) \text{ and } (\mathtt{shift}(S), \mathtt{shift}(T)) \in \gamma \tag{7a}$$
$$S(0) = \mathbf{0} \text{ and } (\mathtt{shift}(S), T) \in \gamma \tag{7b}$$
$$T(0) = \mathbf{0} \text{ and } (S, \mathtt{shift}(T)) \in \gamma \tag{7c}$$

Two streams S and T defined on A are weakly-bisimilar*, denoted $S \approx T$, if there exists a weak-bisimulation γ such that $(S, T) \in \gamma$.*

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

Remark 5. In the rest of the paper, we consider only streams of actions in $A^{\mathbf{0}} := A \cup \{\mathbf{0}\}$ of this sub-type:

$$A^\omega := \{S \mid S : \mathbb{N} \to A^{\mathbf{0}}\} \subset \mathbf{A}^\omega.$$

We need to introduce now the notion of *view of base v* (or *node view*) of a stream S of actions on a dynamic monoid M:

Definition 18. *Given a stream of actions $S \in A_M^\omega$ 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:*

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

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

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

Definition 19. *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:*

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

The elementary computational step associated with an action α yields the set of residuals $\mathtt{Res}(\alpha)$ (Definition 10). Such a set is then transformed into the stream $\mathsf{execute}(\alpha)$ to be combined with the stream of actions to be processed as specified in the abstract machine definitions (Figures 14, 16, 17 and 18).

Definition 20. *Given a set $X \subset A$ we define*

$$\mathtt{set2stream}(X) := \begin{cases} x :: \mathtt{set2stream}(X \backslash x) & \text{for some } x \in X, \\ \mathsf{nil} & \text{if } X = \emptyset. \end{cases}$$

In other words, the function `set2stream` *recasts sets to streams, up to permutations. In particular, if the input is a finite set, then the function returns a finite stream.*

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

$$\mathsf{execute}(\alpha) := \mathtt{set2stream}(\mathtt{Res}(\alpha)).$$

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

$$\mathsf{execute}(\alpha) = \mathtt{set2stream}(\emptyset) = \mathsf{nil}.$$

Remark 6. This non-deterministic definition stems from one of the main features of local and asynchronous execution displayed in VRs: parallel implementation circumvents the typical confluence and the synchronisation difficulties in distributed systems, inasmuch as the underlying algebraic machinery ensures the correctness of the computation.

5.1 Streams and sequential abstract machines

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 a finite sequence of actions used to store the current action;
- the *environment* E that is a node of the graph D providing the local environment where the current action has to be performed, or it is not determined (`NULL`);
- the *core stream* C that is a stream of actions either provided as initial input or created during the execution of other actions, to be executed in the context of the graph D;
- the *dump* D that is the current dynamic graph containing the environment for the next actions.

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

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

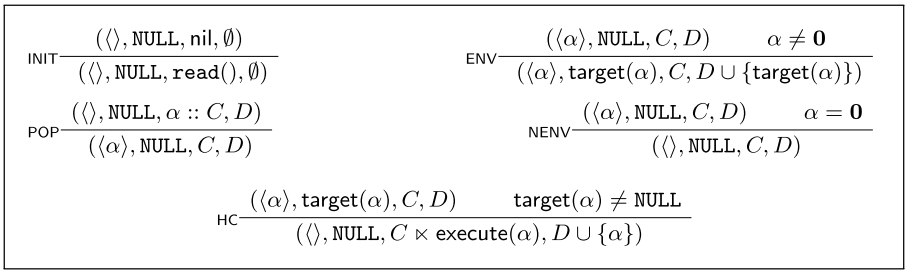

Figure 14. Sequential abstract machine.

where S is a tuple of actions of $A_M^{\mathbf{0}}$ (see Definition 7 and Remark 5), D is a dynamic graph, E is a vertex of D, and C is a possibly infinite sequence of actions from A_M. As it is clear by the definition above, at any step of computation only a finite number of edges do connect to a given node. Therefore S, which is a set of residuals with the same target node, contains a finite number of actions. On the other hand, the number of involved edges can increase, so that the number n of actions to be stacked in S is not bounded.

For the sake of simplicity, we adopt the following notation when we have to add nodes to the dump: $D \cup \{v_t, v_s\} = (V_D \cup \{v_t, v_s\}, E_D)$, or when 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. NULL). We also introduce the notation

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

This function takes as a value a node of the graph, considered as the environment where the action represented by α has to be done (if $\alpha = \mathbf{0}$ we get the NULL environment). Then, $\mathsf{target}(\alpha)$ is added to the graph as a node. Again if $\alpha = \mathbf{0}$ we have

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

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

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

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 $\mathfrak{c} = (S, E, C, D)$. Then, we obtain the new configuration $\mathfrak{c}' = (S', E', C', D')$ by applying the transition R_1 and we write $\mathsf{R}_1(\mathfrak{c}) = \mathfrak{c}'$.

Figure 14 displays the five types of transition which fully describe the sequential abstract machine. The infinite execution loop for it is given by

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

to be applied to the initial configuration $\mathfrak{c}_0 := (\langle\rangle, \texttt{NULL}, \mathsf{nil}, \emptyset)$. Each time the machine reaches the initial configuration $\mathfrak{c}_0$ a call to the read() function is issued. Such a function returns a stream of actions which becomes the new core stream. The read() can be specialised to match different uses of the machine: it can be realised as the connection of the machine to a communication channel in order to intercept the result of another execution, or to connect to the interpreter of a program which injects the stream of actions corresponding to the program.

Algorithm 3 Restatement of the execution cycle (Equation (8)) of the sequential machine in Figure 14.

```
configuration ← c_0
while true do
    configuration ← INIT(configuration)
    while ¬T_0 do
        configuration ← POP(configuration)
        if T_1 then
            configuration ← NENV(configuration)
        else
            configuration ← ENV;HC(configuration)
        end if
    end while
end while
```

Remark 7. In the setting of Example 2, the stream returned by the function `read()` is the GoI interpretation of the λ-term (ΔI) concatenated with the stream nil.

Note that the infinite loop in Equation (8) is the only way to concatenate the transitions; in fact, if we consider the distinct cases of configurations before applying the rules, we have the following mutually exclusive conditions:

$$\big(\mathit{configuration} = \mathfrak{c}_0\big) \tag{T_0}$$

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

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

$$\big(\mathit{configuration} = (\langle\alpha\rangle, \mathsf{target}(\alpha), C, D)\big) \text{ and } \mathsf{target}(\alpha) \neq \texttt{NULL} \tag{T_3}$$

We can restate the machine in a procedural style as reported in Algorithm 3. For an example of its application, we refer to Figure 15.

Remark 8. By construction, at any step of computation the stack S is either the *empty stack*, $S = \langle\rangle$, or it contains the next action acting on the graph, $S = \langle\alpha\rangle$. However, by definition S may contain any finite number of actions: this more general setting actually serves to define the Full Combustion Abstract Machine, defined in Section 5.2 and to prove the correctness of the parallel execution with respect to the sequential machine (see Theorem 2). We use the same notation to implement push and pop operations on stacks, so that:

$$\alpha :: \langle\alpha_1, \ldots, \alpha_n\rangle := \langle\alpha, \alpha_1, \ldots, \alpha_n\rangle.$$

Remark 9 (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.

Input stream is injected in the core stream C whenever the `INIT` rule is applied to a configuration and a `read()` function is called. Both cases display the problem of getting the result of the computation, for the result it has to be *extracted* from the dump D. This problem consists in partitioning the dump graph into terminal nodes and non-terminal ones. To effectively cope with the terminal nodes, during initialisation we explicitly tag them. Then, in the course of a computation, the tag for a terminal node is broadcasted to nodes which are source node of actions pointing to those nodes. As an alternative, actions pointing to terminal nodes can be instantly emitted through

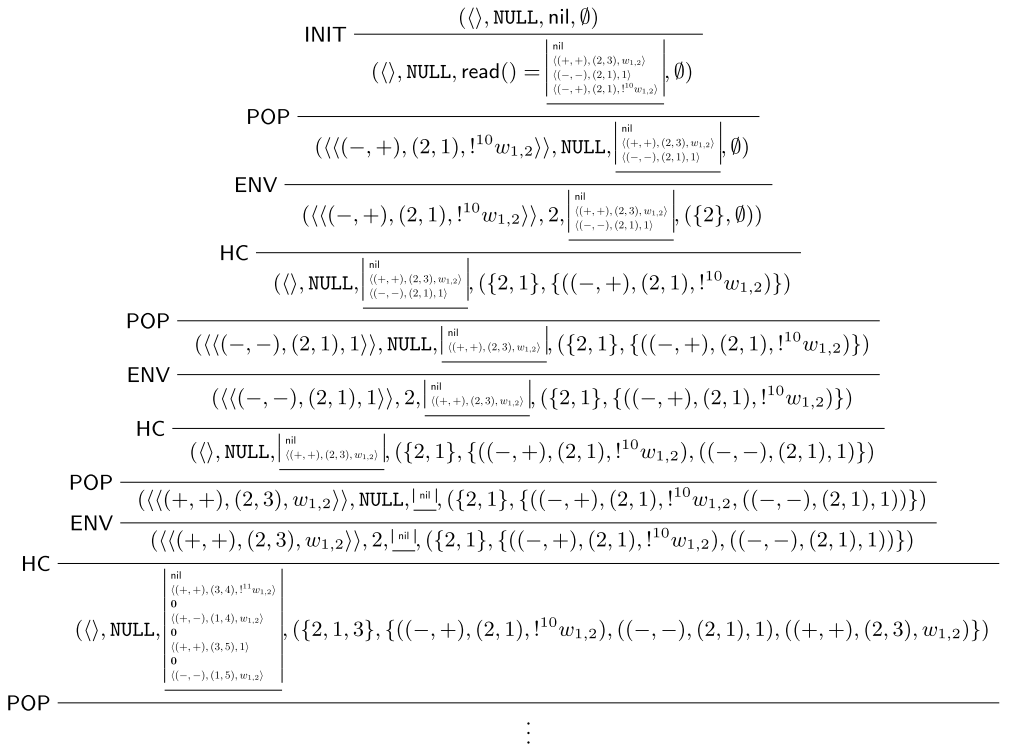

Figure 15. First steps of computation with the sequential abstract machine of the successor function applied to the integer $n = 10$ described in Example 4.

a devoted output stream without inserting them in the dump graph. Note that in the special case in which the input stream has a finite number of actions pointing to terminal nodes, the termination can be checked dynamically during the computation even if the input stream is infinite.

5.2 Generalised full combustion abstract machine

Now we introduce a variant of the SECD machine defined in Section 5.1. Such a variant processes all the actions in a view of base v before focusing on another node: this execution strategy generalizes the *full combustion strategy* for DVR to arbitrary dynamic monoids.

Operations of the *full combustion* machine differ from the ones 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 occur as source node of any action in the stream (as a consequence of the half combustion rule), no further action with v as target node can be residual of the action. Whenever an explicit reference to the node v is required, we write $\mathsf{POP}^{\dagger}_{v}$.

Definition 21 (Zero-out-valence node). *By* zero-out-valence node *v of a stream of actions C, we mean any node not occurring as source node of any action in C. By $\mathsf{ZOV}_C(v)$, we indicate the condition holding when v is a zero-out-valence node of C.*

While executing steps of the sequential full combustion machine, the choice of the node v in the POP rule must fall on a zero-out-valence node, otherwise the NENV rule is applied. By following this strategy we are granted that after processing any action in the stack, the node v is not involved in any further step of computation.

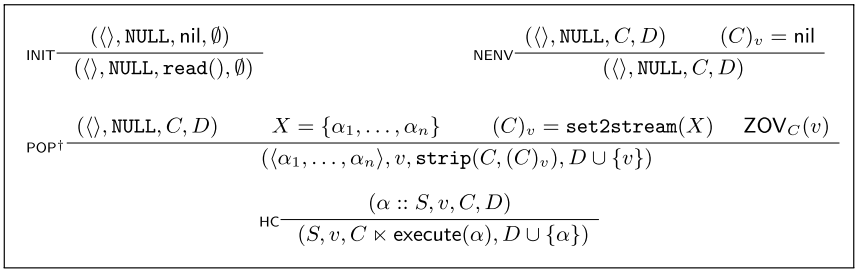

Figure 16. Sequential full combustion abstract machine.

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. This machine is given in Figure 16.

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

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

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

In Section 2.1, we supply the set-theoretic description of our computing device where the state is represented by the dynamic graph G and the task to be executed is represented by the sequence of pending actions A, see Equation (1). The machine consumes the first action of the sequence, modifies the state (i.e. the dynamic graph) and computes residual actions Δ_α which are then added to the sequence of pending actions. The computation of residuals of α has been formalised as the elementary computational step (Section 3.2) and included in the formal description of the SECD machine in Section 5.1.

If $M = \Lambda^*$ and the input is the encoding of a λ-term into a virtual net (see Example 2), then an elementary computational step coincides with a half-combustion step of DVR. In Danos and Regnier (1993), the theory of VRs 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 local confluence property was shown: in fact, the algebraic modification was sufficient to keep the computation coherent making the execution formula (the result of the computation) irrespective of the order of execution.

As a consequence, since DVR is obtained as a special case of VR, we get that 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 may be viewed at the origin of the implementation of PELCR as well (Danos et al. 1997; Pedicini and Quaglia 2007).

Remark 10. While introducing a model of machine evaluating in parallel (with two or more computational units exchanging data), we take in account two aspects: first, a term in untyped λ-calculus may not have a finite execution and second, by the compositional nature of declarative programming, the machine possibly have to execute the application of a term to the infinite result of a non-terminating λ-term reduction.

A further motivation for a stream-based approach is that already when there are two computational units, the second one is required to cope with the infinite sequence of actions produced by the first one. Hence, in the case of infinite execution, the second computational unit receives a stream of actions as an input. This is in accordance with the idea that the result of a computation must be usable as the input of a different computation.

6. Parallel abstract machines

A parallel abstract machine consists of a tuple of units. Any unit has its own configuration constituted by four components: a stack, an environment, a core stream and a dump. Note that to ensure that units are capable of performing elementary computational steps independently of each other, actions need to be properly distributed in the various units. In particular, actions in the core stream of each unit must have their target node in the dumped graph of the same unit.

We employ the symbol $\otimes$ to denote the Cartesian product of corresponding components (stacks, environments, cores, and dumps) belonging to different units. This notation is justified by considering that concurrent evaluation on each unit requires consistency conditions on the way the global configuration's decomposition is managed. Intuitively, the tensor should be regarded as a disjoint union of the components plus some constraints; for instance, the global dumped graph (of the machine) is in fact decomposed into a tuple of dynamic graphs, one for each unit; the core stream of the corresponding unit, which is a stream of actions, must contain only actions with target node in the dump of the same unit.

First of all, we can use a compact notation for the components of the machine assembled from k corresponding components of the units. For example, in the case of the stack component we have:

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

Then, in the special case where 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 $D = D_1 \otimes D_2 \otimes \cdots \otimes D_k$ is definable as the union of edge sets in the individual D_i that is

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

As tradition, we distinguish between synchronous and asynchronous parallel machines. In the first case, the computing units perform a step of computation at the same time, units are clock synchronised and the computation proceeds on each unit. An asynchronous computation can be modelled as a machine with independent units: for instance, the fact that a unit performs many steps of computation while other units perform one single step can be captured by introducing a scheduler selecting the unit from which the action is taken and the corresponding elementary computational step performed.

6.1 Synchronous case

The synchronous model of execution makes the machine run into the computational cycle in a synchronous way. In other terms, the same step of computation is performed (synchronously) on any computational unit. This forces to mix together the rules ENV and NENV: in Figure 17 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 each unit are **0**.

We also stress that at any step of computation the dumped graphs provide a partition of the global current graph $D = D_1 \cup \cdots \cup D_k$; the graph is decomposed into the disjoint union of its edges; any edge is attached to one and only one graph D_i for some i. In fact, the edge information is required at the unit hosting the target node, since it is there that an action meets edges of its

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

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

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

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

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

Figure 17. Parallel k-units synchronous abstract machine.

context. Therefore, it is possible that an edge belongs to D_i and its source node belongs to D_j with $i \neq j$. To deal the case of edges with a source node on a graph and target node, we adopt the following strategy, aiming to avoid duplication of nodes: 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 17 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:

$$\alpha_i = \left(\left(\varepsilon_t^i, \varepsilon_s^i\right), \left(v_t^i, \rho\left(v_s^i\right)\right), w_i\right), \tag{10}$$

which uniformly treats each edge of the graph and solves the question of the special edges which cross the partition of the graph.

Remark 11.

1. The computation of the stream $(\mathsf{execute}(\alpha_j))_{D_i}$ is performed by the jth computing unit, while the sub-stream relative to the nodes in the ith dumped graph D_i is zipped to the stream C_i on the ith computing unit: this leads to the communication of residual actions towards their respective computing units.
2. Actions in the set of residuals $\texttt{Res}(\alpha_i)$ of the action α_i possibly have target nodes in the dumped graph distributed on the units. In fact, the target node of these residual actions coincides with the source node v_s^i. The unit hosting the node v_s^i is selected by considering the view (Definition 18) 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.
3. In the HC rule we have the formula

$$\bigotimes_{i=1}^k \left(C_i \ltimes (\mathsf{execute}(\alpha_{i_1}))_{D_i} \ltimes \ldots \ltimes \left(\mathsf{execute}\left(\alpha_{i_d}\right)\right)_{D_i}\right)$$

expressing the essence of PELCR parallelism: units perform in parallel elementary computational steps associated with the action in the respective stack (if it is non-null) and produced residual actions are recombined with the core stream of the respective target node. In particular, this last recombination implicitly involves a communication step among units which have to transfer residual actions produced by a unit to the unit hosting the target node of each one.

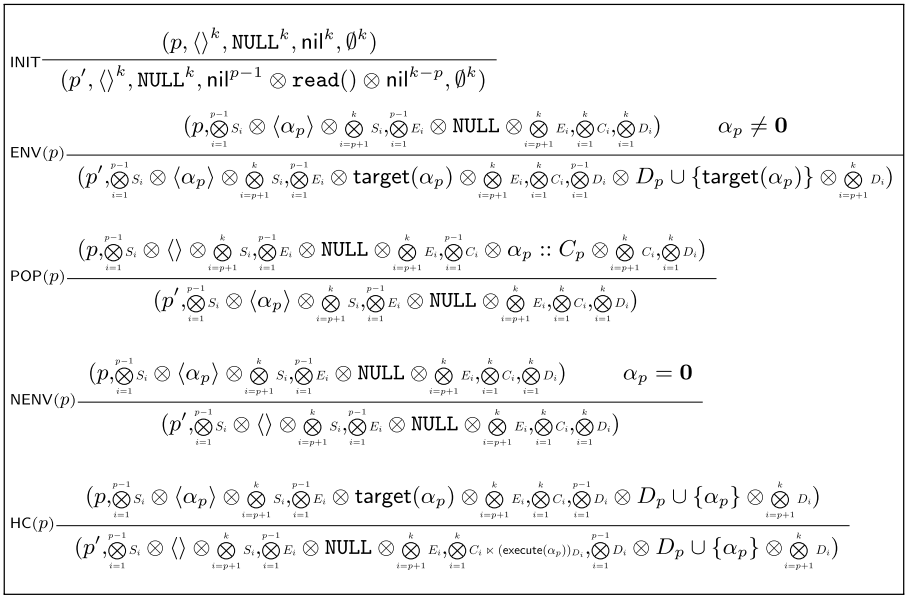

Figure 18. Parallel k-units asynchronous abstract machine.

4. Note that the source node of pairs of residuals coming from the same action-edge interaction is a newly created node v. 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.

6.2 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 behaviour is realised through an asynchronous scheduling mode establishing the order of execution.

The configuration of an asynchronous parallel machine with k computing units is represented by the four (SECD) components together with a *control number* taken in the set $\{1, \ldots, k\}$ of *unit identifiers*. The control number p attached to the configuration identifies the unit to be activated at the next transition step:

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

where $p \in \{1, \ldots, k\}$.

The asynchronous model of execution makes the units proceed in the computational cycle in an independent way. Each unit has to follow the execution cycle as specified in Equation (8), irrespective of the order of application at different units. At each transition the configuration is updated by following the rules in Figure 18, where also the control number is updated: the sequence of control numbers (which is referred as the *scheduling*) is in turn a stream (of unit identifiers) and it can be either explicitly specified (forcing the machine to follow a deterministic strategy in the choice of the unit) or it can be completely random, so that any order of updating may possibly occur.

This is to say that our asynchronous machine is very similar to the sequential one (and the two definitions coincide for $k = 1$). The reader may observe that conditions for determining the rule to be applied are the very same T_0, T_1, T_2 and T_3 of the sequential machine (see Algorithm 3), albeit parametrised by the control number p.

Remark 12. By choosing the scheduling constant to 1 and by allocating newly created nodes (see Remark 11.2) to the first 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 use full combustion strategy as a means to show invariance of execution with respect to the parallel version. To this aim, we introduce full combustion also in the case of k-units machines. This strategy is straightforwardly 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 in the full combustion strategy for the parallel asynchronous machine is determined by the chosen sequence of nodes v, i.e., the scheduling is a sequence of repeated control number $p(v)$, the number associated with the hosting unit of v, for the subsequent HC transitions, until the stack is emptied.

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

$$POP^\dagger(v)\ \frac{p := p(v) \qquad \left(p, \langle\rangle^k, \mathtt{NULL}^k, \bigotimes_{i=1}^{k} C_i, \bigotimes_{i=1}^{k} D_i\right) \qquad X = \{\alpha_1, \ldots, \alpha_n\} \qquad (C_p)_v = \mathtt{set2stream}(X) \qquad ZOV_{C_p}(v)}{\left(p, \langle\rangle^{p-1} \otimes \langle \alpha_1, \ldots, \alpha_n\rangle \otimes \langle\rangle^{k-p}, \mathtt{NULL}^{p-1} \otimes v \otimes \mathtt{NULL}^{k-p}, \bigotimes_{i=1}^{p-1} C_i \otimes \mathtt{strip}(C_p, (C_p)_v) \otimes \bigotimes_{i=p}^{k} C_i, \bigotimes_{i=1}^{p-1} D_i \otimes D_p \cup \{v\} \otimes \bigotimes_{i=p+1}^{k} D_i\right)}$$

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 $p(v)$:

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

Note that by HC, we mean a k-ary version of the half combustion rule in Figure 16 since in this mode of execution at most one stack is non-empty at any given step

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

Note also that in such a version of the machine the scheduling is determined by the choice of the node selected in the POP rule; therefore, it may be omitted when stating the POP and HC.

7. Execution equivalence

In this section, we assume that $M = \Lambda^*$ and that the input is the encoding of a λ-term into a virtual net. As mentioned in Section 5.3, 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 obtain the soundness of the parallel version by showing that for any input stream obtained by the `read()` function 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 are to introduce the notion of equivalence of the configurations of two machines.

Definition 22 (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 in D_1 if and only if $\phi(v_1)$ and $\phi(v_2)$ are adjacent in D_2).*

We extend this definition to polarised dynamic graphs by assuming that ϕ also preserves the labels of edges and the product of target polarities of pairs of edges incident 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 incident on the same node, the corresponding target polarities, say ε_{t_1} and ε_{t_2}, satisfy $\varepsilon_{t_1}\varepsilon_{t_2} = \varepsilon'_{t_1}\varepsilon'_{t_2}$ [1].

Definition 23. *Let $\mathfrak{c}_1 = (S_1, E_1, C_1, D_1)$ and $\mathfrak{c}_2 = (S_2, E_2, C_2, D_2)$ be configurations of the machines $\mathcal{M}_1$ and $\mathcal{M}_2$, respectively. We say that $\mathfrak{c}_1$ is* equivalent *to $\mathfrak{c}_2$ ($\mathfrak{c}_1 \simeq \mathfrak{c}_2$) whenever there exists an isomorphism ϕ of dynamic polarised graphs between D_1 and D_2 such that:*

1. *for any node $v \in D_1$ we have equivalent views on the two core streams when taking v and its corresponding node $\phi(v)$, that is $(C_1)_v \approx \sigma((C_2)_{\phi(v)})$ for some permutation of actions σ,*
2. *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 relation $\simeq$ between configurations is an equivalence relation.*

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

Another important fact is that any elementary step of reduction, performed on actions with the same target node, is executed on the same computational unit:

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 $\alpha \in \mathtt{Res}(\beta)$.*

Proof. Immediate after the definition of polarised view of the core stream with respect to a node (Definition 18). □

Remark 13. 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. This is 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 $v.\mathfrak{c}$ amounts to the configuration obtained starting with configuration $\mathfrak{c}$ after the execution of one step of full-combustion with base v, i.e.

$$v.\mathfrak{c} := \mathsf{POP}^{\dagger}_{v};\mathsf{HC}^{*}(\mathfrak{c})$$

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. Note also that the dumped graphs of the other units are left unchanged.

We are in position to prove that full combustion implemented in the parallel case is equivalent to sequential full combustion:

Theorem 2. *Let $\mathcal{M}_1$ be a sequential machine and $\mathcal{M}_2$ a parallel machine. If a configuration $\mathfrak{c}_1$ of $\mathcal{M}_1$ is equivalent to the configuration $\mathfrak{c}_2$ of $\mathcal{M}_2$ (i.e. there exists a graph isomorphism ϕ satisfying Assumptions 1 and 2 of Definition 23), then $v.\mathfrak{c}_1$ is equivalent to $\phi(v).\mathfrak{c}_2$.*

Proof. By hypothesis, $v.\mathfrak{c}_1$ is one step of full-combustion with base v. This implies that v is a zero-out valence node with respect to the core stream of the configuration $\mathfrak{c}_1$ (otherwise, by definition, we cannot perform the step of full combustion). On the other hand, the equivalence $\mathfrak{c}_1 \simeq \mathfrak{c}_2$ also

ensures that $\phi(v)$ is a zero-out valence node with respect to the core stream of the configuration $\mathfrak{c}_2$. After this step of computation, we can show that $\mathfrak{c}_1' := v.\mathfrak{c}_1$ is equivalent to $\mathfrak{c}_2' := \phi(v).\mathfrak{c}_2$ by establishing an isomorphism ϕ' between $\mathfrak{c}_1'$ and $\mathfrak{c}_2'$. For any pair (α_i, α_j) of opposite polarities in the view of base v, we possibly have a pair of residuals (α_i', α_j') after performing the full action $v.\mathfrak{c}_1$; these two residuals have a new common source node v_s; thus, it suffices to consider ϕ' coinciding with ϕ on nodes and edges in $\mathfrak{c}_1$ and defining $\phi'(v_s)$ as equal to the common source node of the pair of residuals of $(\phi(\alpha_i), \phi(\alpha_j))$. □

8. Conclusions

In this paper, we introduce a stream-based class of abstract machines whose elementary step generalises the half-combustion strategy for DVR. When one considers the classical GoI setting, the proposed approach supplies 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 exist implementations of functional languages that are generally more efficient in the sequential case, the advantage of PELCR is that it can execute those jobs whose huge size turns out to be intractable on sequential machines. Parallel implementations of optimal reductions are tricky, insofar as without optimisation they are not particularly efficient. Moreover, most of the significant optimisations only works in the sequential case, like in Asperti's implementation based on safe operators (Asperti and Chroboczek 1997), which employs a *sequential* safe-tagging algorithm. PELCR's capability of distributing *dynamically* the workload among the available processors displays *intrinsic* parallelism of programs at hand (thus requiring no annotation on the part of the programmer).

Starting from this work, it is our intention 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 input structures different from virtual nets. Indeed, an example of computation of the language recognised by an automaton and an encoding of natural numbers are proposed. In future work, we plan to investigate possible further relations between PELCR and other models of computations. We wish also to introduce a probabilistic dimension in the parallel evaluation of functional programming languages, with a very simple semantics and therefore a clear interpretation of what is a stochastic evaluation of λ-terms, independent from the evaluation strategy. The idea is to have a semantics relying on the adequacy of the local and asynchronous evaluation.

Author ORCIDs. Marco Pedicini https://orcid.org/0000-0002-9016-074X

Acknowledgements. 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, including a new result (Theorem 1) stating how abstract machines can be used as a computational device in the context of formal languages.

We thank the anonymous reviewers for their careful reading of our manuscript and their many insightful comments and suggestions.

Notes

1 Note that by construction, when considering pairs of dynamic graphs generated by the sequential abstract machine, the product of source polarities of edges outgoing from the same node is automatically preserved.

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*, 363–376.

Allombert, V. and Gava, F. (2018). An ML implementation of the MULTI-BSP model. In: *2018 International Conference on High Performance Computing Simulation (HPCS)*, 500–507. Orleans, France.

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*, vol. 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. (2014). Can a light typing discipline be compatible with an efficient implementation of finite fields inversion? *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8552:38–57.

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

Cesena, E., Pedicini, M. and Roversi, L. (2012). Typing a core binary-field arithmetic in a light logic. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7177 LNCS:19–35.

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, 519–538. ACM. San Diego, California, USA.

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

Curien, P.-L. (1991). An abstract framework for environment machines. *Theoretical Computer Science* 82(2) 389–402.

Dal Lago, U., Faggian, C., Valiron, B. and Yoshimizu, A. (2015). Parallelism and synchronization in an infinitary context. In: *2015 30th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2015)*, IEEE Computer Society, Los Alamitos, CA, 559–572.

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

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)*, IEEE Computer Society Press, 296–306.

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

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

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

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

Girard, J.-Y. (1995). Geometry of interaction III: Accommodating the additives. In *Advances in Linear Logic*, Cambridge University Press, 329–389.

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*, Albuquerque, New Mexico, 15–26.

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

Lamping, J. (1989). An algorithm for optimal lambda calculus reduction. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 16–30. San Francisco, California, USA.

Landin, P. J. (1964). The mechanical evaluation of expressions. *Computer Journal* 6(4) 308–320.

Lawson, M. V. (1998). *Inverse Semigroups: The Theory of Partial Symmetries*, World Scientific.

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*, 159–191.

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

Muroya, K. and Ghica, D. R. (2017). The dynamic geometry of interaction machine: a call-by-need graph rewriter. In: *Computer science logic 2017*, volume 82 of *LIPIcs. Leibniz Int. Proc. Inform.*, Art. No. 32, 15. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern.

Pedicini, M. (1998). *Exécution 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. (ed.) *Preproceedings of the 15th Symposium on Trends in. Functional Programming*. Soesterberg, the Netherlands.

Pedicini, M. and Piazza, M. (2018). Kálmar elementary complexity and von Neumann algebras. *Panamerican Mathematical Journal* 28(4) 1–28.

Pedicini, M. and Quaglia, F. (2000). A parallel implementation for optimal lambda-calculus reduction. In: *Proceedings of the 2nd ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming – PPDP '00*. Montreal, Quebec, Canada.

Pedicini, M. and Quaglia, F. (2002). Scheduling vs communication in PELCR. In: Monien, B. and Feldmann, R. (eds.) *Euro-Par 2002 Parallel Processing*, Springer, Berlin Heidelberg, 648–655.

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: Abramsky, S. (eds.) *Typed Lambda Calculi and Applications*, vol. 2044 of *Lecture Notes in Computer Science*, Springer, Berlin, 385–399.

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* 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.

Cite this article: Lai AC, Pedicini M and Piazza M (2019). Abstract machines, optimal reduction, and streams. *Mathematical Structures in Computer science* **29**, 1379–1410. https://doi.org/10.1017/S096012951900001X