

# The Dynamic Geometry of Interaction Machine: A Call-by-need Graph Rewriter

Koko Muroya and Dan Ghica

University of Birmingham, UK

## Abstract

Girard’s Geometry of Interaction (GoI), a semantics designed for linear logic proofs, has been also successfully applied to programming language semantics. One way is to use abstract machines that pass a token on a fixed graph along a path indicated by the GoI. These token-passing abstract machines are space efficient, because they handle duplicated computation by repeating the same moves of a token on the fixed graph. Although they can be adapted to obtain sound models with regard to the equational theories of various evaluation strategies for the lambda calculus, it can be at the expense of significant time costs. In this paper we show a token-passing abstract machine that can implement evaluation strategies for the lambda calculus, with certified time efficiency. Our abstract machine, called the *Dynamic GoI Machine* (DGoIM), rewrites the graph to avoid replicating computation, using the token to find the redexes. The flexibility of interleaving token transitions and graph rewriting allows the DGoIM to balance the trade-off of space and time costs. This paper shows that the DGoIM can implement call-by-need evaluation for the lambda calculus by using a strategy of interleaving token passing with as much graph rewriting as possible. Our quantitative analysis confirms that the DGoIM with this strategy of interleaving the two kinds of possible operations on graphs can be classified as “efficient” following Accattoli’s taxonomy of abstract machines.

## 1 Introduction

### 1.1 Token-passing Abstract Machines for $\lambda$ -calculus

Girard’s Geometry of Interaction (GoI) [16] is a semantic framework for linear logic proofs [15]. One way of applying it to programming language semantics is via “token-passing” abstract machines. A term in the  $\lambda$ -calculus is evaluated by representing it as a graph, then passing a token along a path indicated by the GoI. Token-passing GoI decomposes higher-order computation into local token actions, or low-level interactions of simple components. It can give strikingly innovative implementation techniques for functional programs, such as Mackie’s *Geometry of Implementation* compiler [20], Ghica’s *Geometry of Synthesis* (GoS) high-level synthesis tool [12], and Schöpp’s resource-aware program transformation to a low-level language [25]. The interaction-based approach is also convenient for the complexity analysis of programs, e.g. Dal Lago and Schöpp’s INTML type system of logarithmic-space evaluation [7], and Dal Lago et al.’s linear dependent type system of polynomial-time evaluation [5, 6].

Fixed-space execution is essential for GoS, since in the case of digital circuits the memory footprint of the program must be known at compile-time, and fixed. Using a restricted version of the call-by-name language Idealised Algol [13] not only the graph, but also the token itself can be given a fixed size. Surprisingly, this technique also allows the compilation of recursive programs [14]. The GoS compiler shows both the usefulness of the GoI as a guideline for unconventional compilation and the natural affinity between its space-efficient abstract machine and call-by-name evaluation. The practical considerations match the prior theoretical understanding of this connection [9].

In contrast, re-evaluating a term by repeating its token actions poses a challenge for call-by-value evaluation (e.g. [11, 24, 18, 3]) because duplicated computation must not lead to repeated evaluation. Moreover, in call-by-value repeating token actions raises the additional technical challenge of avoiding repeating any associated computational effects (e.g. [23, 22, 4]). A partial solution to this conundrum is to focus on the soundness of the equational theory, while deliberately ignoring the time costs [22]. However, Fernández and Mackie suggest that in a call-by-value scenario, the time efficiency of a token-passing abstract machine could also be improved, by allowing a token to jump along a path, even though a time cost analysis is not given [11].

For us, solving the the problem of creating a GoI-style abstract machine which computes efficiently with evaluation strategies other than call-by-name is a first step in a longer-range research programme. The compilation techniques derived from the GoI can be extremely useful in the case of unconventional computational platforms.

But if GoI-style techniques are to be used in a practical setting they need to extend beyond call-by-name, not just correctly but also efficiently.

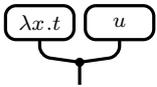
## 1.2 Interleaving Token Passing with Graph Rewriting

A token jumping, rather than following a path, can be seen as a simple form of short-circuiting that path, which is a simple form of graph-rewriting. This idea first occurs in Mackie’s work as a compiler optimisation technique [20] and is analysed in more depth theoretically by Danos and Regnier in the so-called *Interaction Abstract Machine* [9]. More general graph-rewriting-based semantics have been used in a system called *virtual reduction* [8], where rewriting occurs along paths indicated by GoI, but without any token-actions. The most operational presentation of the combination of token-passing and jumping was given by Fernández and Mackie [11]. The interleaving of token actions and rewriting is also found in Sinot’s interaction nets [26, 27]. We can reasonably think of the DGoIM as their abstract-machine realisation.

We build on these prior insights by adding more general, yet still efficient, graph-rewriting facilities to the setting of a GoI token-passing abstract machine. We call an abstract machine that interleaves token passing with graph rewriting the *Dynamic GoI Machine* (DGoIM), and we define it as a state transition system with transitions for token passing as well as transitions for graph rewriting. What connects these two kinds of transitions is the token trajectory through the graph, its path. By examining it, the DGoIM can detect redexes and trigger rewriting actions.

Through graph rewriting, the DGoIM reduces sub-graphs visited by the token, avoiding repeated token actions and improving time efficiency. On the other hand, graph rewriting can expand a graph by e.g. copying sub-graphs, so space costs can grow. To control this trade-off of space and time cost, the DGoIM has the flexibility of interleaving token passing with graph rewriting. Once the DGoIM detects that it has traversed a redex, it may rewrite it, but it may also just propagate the token without rewriting the redex.

As a first step in our exploration of the flexibility of this machine, we consider the two extremal cases of interleaving. The first extremal case is “passes-only,” in which the DGoIM never triggers graph rewriting, yielding an ordinary token-passing abstract machine. As a typical example, the  $\lambda$ -term  $(\lambda x.t) u$  is evaluated like this:



1. A token enters the graph on the left at the bottom open edge.
2. A token visits and goes through the left sub-graph  $\lambda x.t$ .
3. Whenever a token detects an occurrence of the variable  $x$  in  $t$ , it traverses the right sub-graph  $u$ , then returns carrying the resulting value.
4. A token finally exits the graph at the bottom open edge.

Step 3 is repeated whenever term  $u$  needs to be re-evaluated. This strategy of interleaving corresponds to call-by-name reduction.

The other extreme is “rewrites-first,” in which the DGoIM interleaves token passing with as much, and as early, graph rewriting as possible, guided by the token. This corresponds to both call-by-value and call-by-need reductions, the difference between the two being the trajectory of the token. In the case of call-by-value, the token will enter the graph from the bottom, traverse the left-hand-side sub-graph, which happens to be already a value, then visit sub-graph  $u$  even before  $x$  is used in a call. While traversing  $u$ , it will cause rewrites such that when the token exits, it leaves behind the graph of a machine corresponding to a value  $v$  such that  $u$  reduces to  $v$ . The difference with call-by-need is that the token will visit  $u$  only when  $x$  is encountered in  $\lambda x.t$ . In both cases, if repeated evaluation is required then the sub-graph corresponding now to  $v$  is copied, so that one copy can be further rewritten, if needed, while the original is kept for later reference.

## 1.3 Contributions

This work presents a DGoIM model for call-by-need, which can be seen as a case study of the flexibility achieved through controlled interleaving of rewriting and token-passing. This is achieved through a rewriting strategy which turns out to be as natural as the passes-only strategy is for implementing call-by-name. The DGoIM avoids re-evaluation of a sub-term by rewriting any sub-graph visited by a token so that the updated sub-graph represents the evaluation result, but, unlike call-by-value, it starts by evaluating the sub-graph corresponding to the function  $\lambda x.t$  first. We chose call-by-need mainly because of the technical challenges it poses. Adapting the technique to *call-by-value* is a straightforward exercise, and we discuss other alternative in the Conclusion.

We analyse the time cost of the DGoIM with the rewrites-first interleaving, using Accattoli et al.’s general methodology for quantitative analysis [2, 1]. Their method cannot be used “off the shelf,” because the DGoIM does not satisfy one of the assumptions used in [1, Sec. 3]. Our machine uses a more refined transition system,

in which several steps correspond to a single one in *loc. cit.*. We overcome this technical difficulty by building a weak simulation of Danvy and Zerny’s storeless abstract machine [10] to which the recipe does apply. The result of the quantitative analysis confirms that the DGoIM with the rewrites-first interleaving can be classified as “efficient,” following Accattoli’s taxonomy of abstract machines introduced in [1].

As we intend to use the DGoIM as a starting point for semantics-directed compilation, this result is an important confirmation that no hidden inefficiencies lurk within the fabric of the rather complex machinery of the DGoIM.

## 2 The Dynamic GoI Machine

### 2.1 Well-boxed Graphs

The graphs used to construct the DGoIM are essentially MELL proof structures [15] of the multiplicative and exponential fragment of linear logic. They are directed, and built over the fixed set of nodes called “generators” shown in Fig. 1.

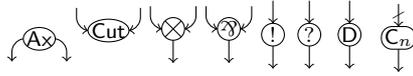


Figure 1: Generators of Graphs

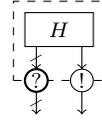


Figure 2: !-box  $H$

A  $C_n$ -node is annotated by a natural number  $n$  that indicates its in-degree, i.e. the number of incoming edges. It generalises a contraction node, whose in-degree is 2, and a weakening node, whose in-degree is 0, of MELL proof structures. In Fig. 1, a bunch of  $n$  edges is depicted by a single arrow with a strike-out.

Graphs must satisfy the well-formedness condition below. Note that, unlike the usual approach [15], we need not assign MELL formulas to edges, nor require a graph to be a valid proof net.

**Definition 2.1** (well-boxed). A directed graph  $G$  built over the generators in Fig. 1 is *well-boxed* if:

- it has no incoming edges
- each !-node  $v$  in  $G$  comes with a sub-graph  $H$  of  $G$  and an arbitrary number of ?-nodes  $\vec{u}$  such that:
  - the sub-graph  $H$  (called “!-box”) is well-boxed inductively and has at least one outgoing edges
  - the !-node  $v$  (called “principal door of  $H$ ”) is the target of one outgoing edge of  $H$
  - the ?-nodes  $\vec{u}$  (called “auxiliary doors of  $H$ ”) are the targets of all the other outgoing edges of  $H$
- each ?-node is an auxiliary door of exactly one !-box
- any two distinct !-boxes with distinct principal doors are either disjoint or nested

Note that a !-box might have no auxiliary doors. We use a dashed box to indicate a !-box together with its principal door and its auxiliary doors, as in Fig. 2. The auxiliary doors are depicted by a single ?-node with a thick frame and with single incoming and outgoing arrows with strike-outs. Directions of edges are omitted in the rest of the paper, if not ambiguous, to reduce visual clutter.

### 2.2 Pass Transitions and Rewrite Transitions

The DGoIM is formalised as a labelled transition system with two kinds of transitions, namely *pass* transitions  $\dashrightarrow$  and *rewrite* transitions  $\rightsquigarrow$ . Labels of transitions are  $\mathbf{b}, \mathbf{s}, \mathbf{o}$  that stand for “beta,” “substitution,” and “overheads” respectively.

Let  $\mathcal{L}$  be a fixed countable (infinite) set of *names*. The state of the transition system  $s = (\mathbb{G}, p, h, m)$  consists of the following elements:

- a *named* well-boxed graph  $\mathbb{G} = (G, \ell_G)$ , that is a well-boxed graph  $G$  with a *naming*  $\ell_G$  that assigns a unique name  $\alpha \in \mathcal{L}$  to each node of  $G$
- a pair  $p = (e, d)$  called *position*, of an edge  $e$  of  $G$  and a *direction*  $d \in \{\uparrow, \downarrow\}$
- a *history stack*  $h$  defined by the grammar  $h ::= \square \mid \mathbf{Ax}_\alpha : h \mid \mathbf{Cut}_\alpha : h \mid \otimes_\alpha : h \mid \wp_\alpha : h \mid !_\alpha : h \mid \mathbf{D}_\alpha : h \mid \mathbf{C}_\alpha^n : h$ , where  $\alpha \in \mathcal{L}$  and  $n$  is some positive natural number.
- a *multiplicative stack*  $m$  defined by the BNF grammar  $m ::= \square \mid \mathbf{l} : m \mid \mathbf{r} : m$ .

We refer to a node by its name, i.e. we say “a node  $\alpha$ ” instead of “a node whose name is  $\alpha$ .”

A pass transition  $(\mathbb{G}, p, h, m) \dashrightarrow_{\circ} (\mathbb{G}, p', h', m')$  changes a position using a multiplicative stack, pushes to a history stack, and keeps a named graph unchanged. All pass transitions have the label  $\circ$ .

Fig. 3 shows pass transitions graphically, omitting irrelevant parts of graphs. A position  $p = (e, d)$  is represented by a bullet  $\bullet$  (called “token”) on the edge  $e$  together with the direction  $d$ . Recall that an edge with a strike-out represents a bunch of edges. The transition in the last line of Fig. 3 (where we assume  $n > 0$ ) moves a token from one of the incoming edges of a  $C_n$ -node to the outgoing edge of the node. Node names  $\alpha \in \mathcal{L}$  are indicated wherever needed.

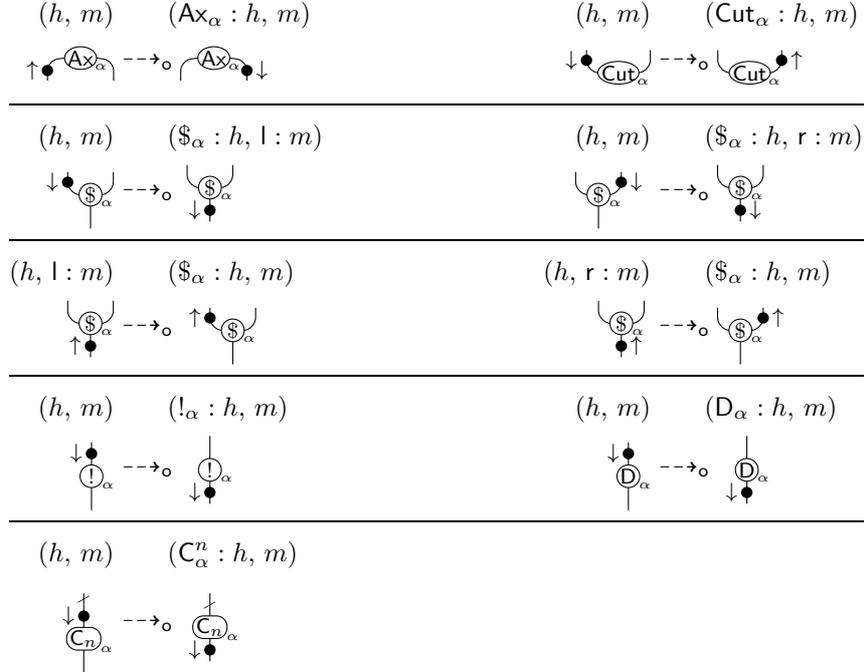


Figure 3: Pass Transitions ( $\$ \in \{\otimes, \wp\}$ ,  $n > 0$ )

A rewrite transition  $(\mathbb{G}, (e, d), h, m) \rightsquigarrow_x (\mathbb{G}', (e', d), h', m)$  consumes some elements of a history stack, rewrites a sub-graph of a named graph, and updates a position (or, more precisely, its edge). The label  $x$  of a rewrite transition  $\rightsquigarrow_x$  is either  $b$ ,  $s$  or  $\circ$ . Fig. 4 shows rewrite transition in the same manner as Fig. 3. Multiplicative stacks are not present in the figure since they are irrelevant. The  $\#$ -node represents some arbitrary node (incoming edges omitted). We can see that no rewrite transition breaks the well-boxed-ness of a graph.

The rewrite transitions (1),(2),(3), and (4) are exactly taken from MELL cut elimination [15]. The rewrite transition (5) is a variant of (1). It acts on a connected pair of a Cut-node and an Ax-node that arises as a result of the transition (6) or (7) but cannot be rewritten by the transition (1). These transitions (6) and (7) are inspired by the MELL cut elimination process for (binary) contraction nodes; note that we assume  $n > 0$  in Fig. 4.

The rewrite transition (6) in Fig. 4 deserves further explanation. The sub-graph  $H^\approx$  is a copy of the  $!$ -box  $H$  where all the names are replaced with fresh ones. The thick  $C_{g+f-1}$ -node and  $C_{g+2f-1}$ -node represent families  $\{C_{g(j)+f-1}\}_{j=0}^m$ ,  $\{C_{g(j)+2f-1}\}_{j=0}^m$ , of C-nodes respectively. They are connected to  $?$ -nodes  $\vec{\epsilon} = \epsilon_0, \dots, \epsilon_l$  and  $\vec{\mu} = \mu_0, \dots, \mu_l$  in such a way that:

- the natural numbers  $l, m$  satisfy  $l \geq m$ , and come with a surjection  $f: \{0, \dots, l\} \rightarrow \{0, \dots, m\}$  and a function  $g: \{0, \dots, m\} \rightarrow \mathbb{N}$  to the set  $\mathbb{N}$  of natural numbers
- each  $?$ -node  $\epsilon_i$  and each  $?$ -node  $\mu_i$  are both connected to the C-node  $\phi_{f(i)}$
- each C-node  $\phi_j$  has  $g(j)$  incoming edges whose source is none of the  $?$ -nodes  $\vec{\epsilon}, \vec{\mu}$ .

Some rewrite transitions introduce new nodes to a graph. We *require* that the uniqueness of names throughout a whole graph is not violated by these transitions. Under this requirement, the introduced names  $\nu, \vec{\mu}$  and the renaming  $H^\approx$  in Fig. 4 can be arbitrary.

**Definition 2.2.** We call a state  $((G, \ell_G), p, h, m)$  *rooted at  $e_0$*  for an open (outgoing) edge  $e_0$  of  $G$ , if there exists a finite sequence  $((G, \ell_G), (e_0, \uparrow), \square, \square) \dashrightarrow^* ((G, \ell_G), p, h, m)$  of pass transitions such that the position  $p$  appears only last in the sequence.

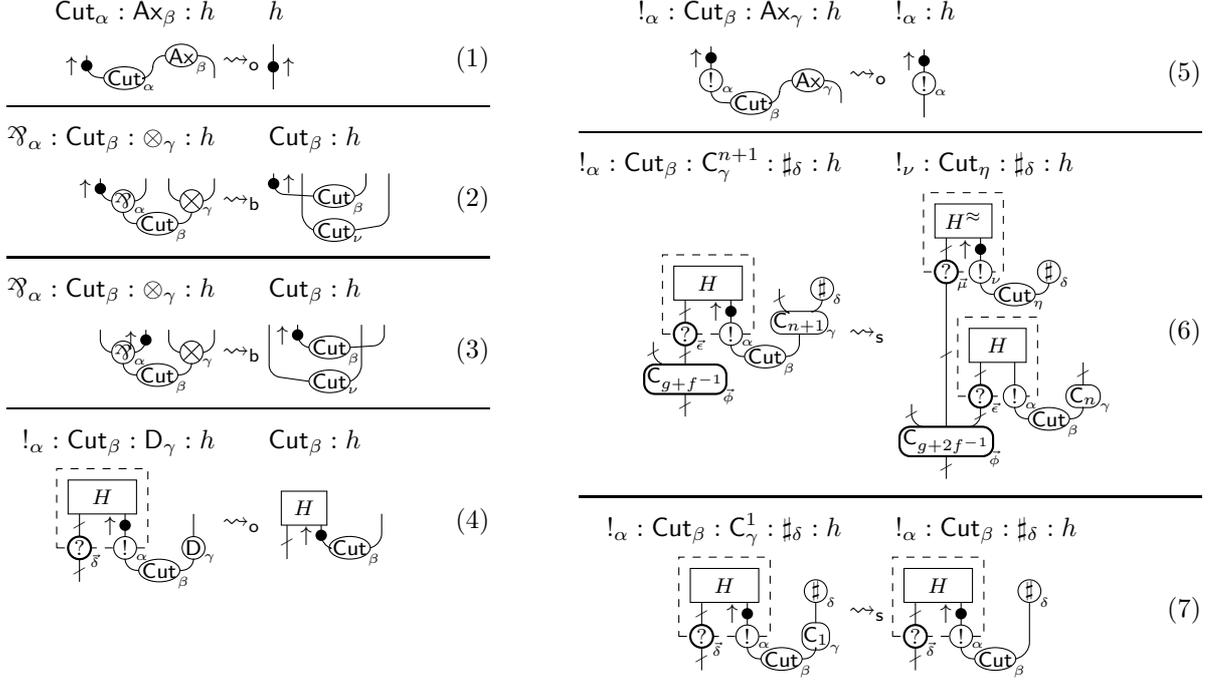


Figure 4: Rewrite Transitions ( $n > 0$ )

Lem. 2.3(1) below implies that, the DGoIM can determine whether a rewrite transition is possible at a rooted state by only examining a history stack. The rooted property is preserved by transitions.

**Lemma 2.3** (rooted states). *Let  $((G, l_G), (e, d), h, m)$  be a rooted state at  $e_0$  with a (finite) sequence*

$$((G, l_G), (e_0, \uparrow), \square, \square) \dashrightarrow^* ((G, l_G), (e, d), h, m).$$

1. *History stack represents an (undirected and possibly cyclic) path of the graph  $G$  connecting edges  $e_0$  and  $e$ .*
2. *If a transition  $((G, l_G), (e, d), h, m) (\dashrightarrow \cup \rightsquigarrow) ((G', l_{G'}), p', h', m')$  is possible, the open edges of  $G'$  are bijective to those of  $G$ , and the state  $((G', l_{G'}), p', h', m')$  is rooted at the open edge corresponding to  $e_0$ .*

*Proof.* (Sketch.) The proof of the first part is by induction on the length of the sequence of move transitions. For the second part, rewrite transitions  $\rightsquigarrow$  modify open edges of a graph in a bijective way. The edge that a state is rooted at can be modified only by the rewrite transitions (1) and (5) involving  $\text{Ax}$ -nodes.  $\square$

### 2.3 Cost Analysis of the DGoIM

The time cost of updating stacks is constant, as each transition changes only a fixed number of top elements of stacks. Updating a position is local and needs constant time, as it does not require searching beyond the next edge in the graph from the current edge. We can conclude all pass transitions take constant time.

We estimate the time cost of rewrite transitions by counting updated nodes. The rewrite transitions (1)–(3) involve a fixed number of nodes, and the transition (7) eliminates one  $\text{C}_1$ -node. Only the transitions (4) and (6) have non-constant time cost. The number of doors deleted in the transition (4) can be arbitrary, and so is the number of nodes introduced in the transition (6).

Pass transitions and rewrite transitions are separately deterministic (up to the choice of new names). However, both a pass transition and a rewrite transition are possible at some states. We here opt for the following “rewrites-first” way to interleave pass transitions with as much rewrite transitions as possible:

$$s \rightarrow_x s' \stackrel{\text{def.}}{\iff} \begin{cases} s \rightsquigarrow_x s' & (\text{if } \rightsquigarrow_x \text{ possible}) \\ s \dashrightarrow_x s' & (\text{if only } \dashrightarrow_x \text{ possible}). \end{cases}$$

The DGoIM with this strategy yields a deterministic labelled transition system  $\rightarrow$  up to the choice of new names in rewrite transitions. We denote it by  $\text{DGoIM}_{\rightarrow}$ , making the strategy explicit. Note that there can be other strategies of interleaving although we do not explore them here.

Before we conclude, several considerations about space cost analysis. Space costs are generally bound by time costs, so from our analysis there is an implicit guarantees that space usage will not explode. But if a more refined space cost analysis is desired, the following might prove to be useful.

Terms	$t ::= x \mid \lambda x.t \mid tt \mid t[x \leftarrow t]$	Pure terms	$\bar{t} ::= x \mid \lambda x.\bar{t} \mid \bar{t}\bar{t}$
Values	$v ::= \lambda x.t$	Pure values	$\bar{v} ::= \lambda x.\bar{t}$
Evaluation contexts	$E ::= \langle \cdot \rangle \mid E\bar{t} \mid E[x \leftarrow \bar{t}] \mid E\langle x \rangle[x \leftarrow E]$		
Substitution contexts	$A ::= \langle \cdot \rangle \mid A[x \leftarrow \bar{t}]$		

$(\bar{t}\bar{u}, E)_{term} \rightarrow_o (\bar{t}, E\langle \cdot \rangle\bar{u})_{term}$	(8)
$(x, E_1\langle E_2[x \leftarrow \bar{t}]\rangle)_{term} \rightarrow_o (\bar{t}, E_1\langle E_2\langle x \rangle[x \leftarrow \langle \cdot \rangle]\rangle)_{term}$	(9)
$(\bar{v}, E)_{term} \rightarrow_o (\bar{v}, E)_{ctxt}$	(10)
$(\lambda x.\bar{t}, E\langle A\bar{u}\rangle)_{ctxt} \rightarrow_b (\bar{t}, E\langle A\langle \cdot \rangle[x \leftarrow \bar{u}]\rangle)_{term}$	(11)
$(\bar{v}, E_1\langle E_2\langle x \rangle[x \leftarrow A]\rangle)_{ctxt} \rightarrow_s (\bar{v}^\approx, E_1\langle A\langle E_2[x \leftarrow \bar{v}]\rangle\rangle)_{ctxt}$	(if $x \in \text{FV}_\emptyset(E_2)$ ) (12)
$(\bar{v}, E_1\langle E_2\langle x \rangle[x \leftarrow A]\rangle)_{ctxt} \rightarrow_s (\bar{v}, E_1\langle A\langle E_2\rangle\rangle)_{ctxt}$	(if $x \notin \text{FV}_\emptyset(E_2)$ ) (13)

Figure 5: Call-by-need Storeless Abstract Machine (SAM)

The space required in implementing a named well-boxed graph is bounded by the number of its nodes. The number of edges is linear in the number of nodes, because each generator has a fixed out-degree and every edge of a well-boxed graph has its source.

Additionally a !-box can be represented by associating its auxiliary doors to its principal door. This adds connections between doors to a graph that are as many as ?-nodes. It enables the DGoIM to identify nodes of a !-box by following edges from its principal and auxiliary doors. Nodes in a !-box that are not connected to doors can be ignored, since these nodes are never visited by a token (i.e. pointed by a position) as long as the DGoIM acts on rooted states.

Only the rewrite transition (6) can increase the number of nodes of a graph by copying a !-box with its doors. Rewrite transitions can copy !-boxes and eliminate the !-box structure, but they never create new !-boxes or change existing ones. This means that, in a sequence of transitions that starts with a graph  $G$ , any !-boxes copied by the rewrite transition (6) are sub-graphs of the graph  $G$ . Therefore the number of nodes of a graph increases linearly in the number of transitions.

Elements of history stacks and multiplicative stacks, as well as a position, are essentially pointers to nodes. Because each pass/rewrite transition adds at most one element to each stack, the lengths of stacks also grow linearly in the number of transitions.

### 3 Weak Simulation of the Call-by-Need SAM

#### 3.1 Storeless Abstract Machine (SAM)

We show the  $\text{DGoIM}_{\rightarrow}$  implements call-by-need evaluation by building a weak simulation of the call-by-need Storeless Abstract Machine (SAM) defined in Fig. 5. It simplifies Danvy and Zerny’s storeless machine [10, Fig. 8] and accommodates a partial mechanism of garbage collection (namely, transition (13)). We will return to a discussion of garbage collection at the end of this section.

The SAM is a labelled transition system between *configurations*  $(\bar{t}, E)$ . They are classified into two groups, namely *term* configurations and *context* configurations, that are indicated by annotations *term*, *ctxt* respectively. Pure terms (resp. pure values) are terms (resp. values) that contain no explicit substitutions  $t[x \leftarrow u]$ ; we sometimes omit the word “pure” and the overline in denotation as long as that raises no confusion.

Each evaluation context  $E$  contains exactly one open hole  $\langle \cdot \rangle$ , and replacing it with a term  $t$  (or an evaluation context  $E'$ ) yields a term  $E\langle t \rangle$  (or an evaluation context  $E\langle E' \rangle$ ) called *plugging*. In particular an evaluation context  $E'\langle x \rangle[x \leftarrow E]$  replaces the open hole of  $E'$  with  $x$  and keeps the open hole of  $E$ .

Labels of transitions are the same as those used for the DGoIM (i.e. **b**, **s** and **o**). The transition (11), with the label **b**, corresponds to the  $\beta$ -reduction where evaluation and substitution of function arguments are delayed. Substitution happens in the transitions (12) and (13), with the label **s**, that replaces exactly one occurrence of a variable. The other transitions with the label **o**, namely  $(\bar{t}, E) \rightarrow_o (\bar{t}', E')$ , search a redex by rearranging a configuration. The two pluggings  $E\langle \bar{t} \rangle$  and  $E'\langle \bar{t}' \rangle$  indeed yield exactly the same term.

We characterise “free” variables using multisets of variables. Multisets make explicit how many times a variable is duplicated in a term (or an evaluation context). This information of duplication is later used in translating terms to graphs.

**Notation** (multiset). A multiset  $\mathbf{x} := [x, \dots, x]$  consists of a finite number of  $x$ . The multiplicity of  $x$  in a multiset  $M$  is denoted by  $M(x)$ . We write  $x \in^k M$  if  $M(x) = k$ ,  $x \in M$  if  $M(x) > 0$  and  $x \notin M$  if  $M(x) = 0$ . A multiset  $M$  comes with its support set  $\text{supp}(M)$ . For two multisets  $M$  and  $M'$ , their sum and difference are

denoted by  $M + M'$  and  $M - M'$  respectively. Removing all  $x$  from a multiset  $M$  yields the multiset  $M \setminus x$ , e.g.  $[x, x, y] \setminus x = [y]$ .

Each term  $t$  and each evaluation context  $E$  are respectively assigned multisets of variables  $\text{FV}(t), \text{FV}_M(E)$ , with  $M$  a multiset of variables. The multisets  $\text{FV}$  are defined inductively as follows.

$$\begin{aligned}
\text{FV}(x) &:= [x], \\
\text{FV}(\lambda x.t) &:= \text{FV}(t) \setminus x, \\
\text{FV}(tu) &:= \text{FV}(t) + \text{FV}(u), \\
\text{FV}(t[x \leftarrow u]) &:= (\text{FV}(t) \setminus x) + \text{FV}(u). \\
\text{FV}_M(\langle \cdot \rangle) &:= M, \\
\text{FV}_M(Et) &:= \text{FV}_M(E) + \text{FV}(t), \\
\text{FV}_M(E[x \leftarrow t]) &:= (\text{FV}_M(E)) \setminus x + \text{FV}(t), \\
\text{FV}_M(E' \langle x \rangle [x \leftarrow E]) &:= (\text{FV}_{[x]}(E')) \setminus x + \text{FV}_M(E).
\end{aligned}$$

The following equations can be proved by a straightforward induction on  $E$ .

**Lemma 3.1** (decomposition).

$$\begin{aligned}
\text{FV}(E\langle t \rangle) &= \text{FV}_{\text{FV}(t)}(E) \\
\text{FV}_M(E\langle E' \rangle) &= \text{FV}_{\text{FV}_M(E')} (E)
\end{aligned}$$

A variable  $x$  is *bound* in a term  $t$  if it appears in the form of  $\lambda x.u$  or  $u[x \rightarrow u']$ . A variable  $x$  is *captured* in an evaluation context  $E$  if it appears in the form of  $E'[x \leftarrow \bar{t}]$  (but not in the form of  $E' \langle x \rangle [x \leftarrow E'']$ ). The transitions (12) and (13) depend on whether or not the bound variable  $x$  appears in the evaluation context  $E_2$ . If the variable  $x$  appears, the value  $\bar{v}$  is kept for later use and its copy  $\bar{v}^\approx$  is substituted for  $x$ . If not, the value  $\bar{v}$  itself is substituted for  $x$ .

The SAM does not assume the  $\alpha$ -equivalence, but explicitly deals with it in copying a value. The copy  $\bar{v}^\approx$  in has all its bound variables replaced by distinct fresh variables (i.e. distinct variables that do not appear in a whole configuration). This implies that the SAM is deterministic up to the choice of new variables introduced in copying.

A term  $t$  is *closed* if  $\text{FV}(t) = \emptyset$ ; and is *well-named* if each variable gets bound at most once in  $t$ , and each bound variable  $x$  in  $t$  satisfies  $x \notin \text{FV}(t)$ . An *initial* configuration is a term configuration  $(\bar{t}_0, \langle \cdot \rangle)_{term}$  where  $\bar{t}_0$  is closed and well-named. A finite sequence of transitions from an initial configuration is called an *execution*. A *reachable* configuration  $(\bar{t}, E)$ , that is a configuration coming with an execution from some initial configuration to itself, satisfies the following invariant properties.

**Lemma 3.2** (reachable configurations). *Let  $(\bar{t}, E)$  be a reachable configuration from an initial configuration  $(\bar{t}_0, \langle \cdot \rangle)_{term}$ . The term  $\bar{t}$  is a sub-term of the initial term  $\bar{t}_0$  up to  $\alpha$ -equivalence, and the plugging  $E\langle \bar{t} \rangle$  is closed and well-named.*

*Proof.* (Sketch.) The proof is by induction on the length of the execution  $(\bar{t}_0, \langle \cdot \rangle)_{term} \rightarrow^* (\bar{t}, E)$ . Not only the term  $\bar{t}$  but also the term  $u'$  in any sub-term  $t' u'$  or  $t'[x \leftarrow u']$  of the plugging  $E\langle \bar{t} \rangle$  is a sub-term of the initial term  $\bar{t}_0$ . The transition (12) renames a value in the way that preserves closedness and the well-named-ness of pluggings. In the transition (13) where an explicit substitution for a bound variable  $x$  is eliminated, the induction hypothesis ensures that the variable  $x$  does not occur in the plugging  $E_1 \langle A \langle E_2 \langle \bar{v} \rangle \rangle \rangle$ .  $\square$

We now conclude with a brief consideration on *garbage collection*. Transition (13) eliminates an explicit substitution and therefore implements a partial mechanism of garbage collection. The mechanism is partial because only an explicit substitution that is looked up in an execution can be eliminated, as illustrated below. The explicit substitution  $[x \leftarrow \lambda z.z]$  is eliminated in the first example, but not in the second example because the bound variable  $x$  does not occur.

$$\begin{aligned}
((\lambda x.x)(\lambda z.z), \langle \cdot \rangle)_{term} &\rightarrow^* (\lambda z.z, \langle \cdot \rangle)_{ctxt} \\
((\lambda x.\lambda y.y)(\lambda z.z), \langle \cdot \rangle)_{term} &\rightarrow^* (\lambda y.y, \langle \cdot \rangle [x \leftarrow \lambda z.z])_{ctxt}
\end{aligned}$$

We incorporate this partial garbage collection to make clear the behaviour of the  $\text{DGoIM}_{\rightarrow}$ , in particular the use of the rewrite transitions (6) and (7).

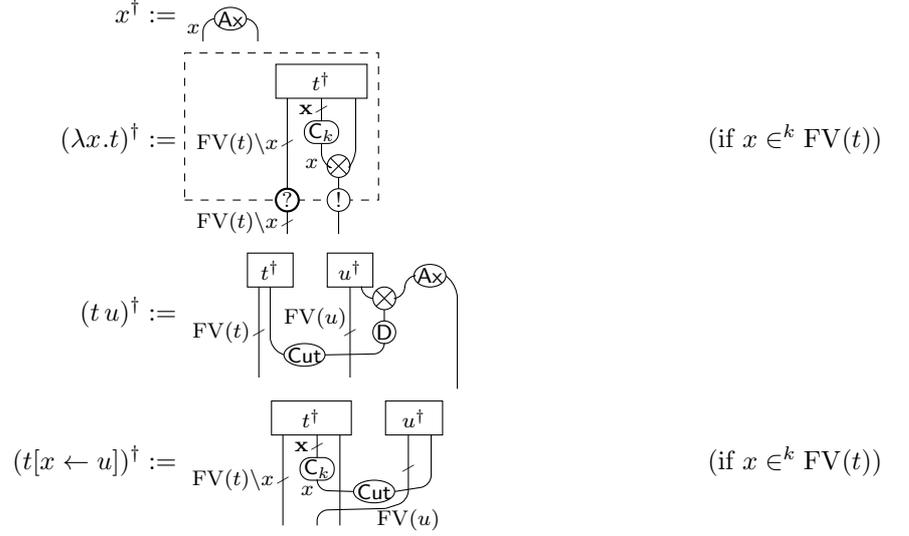


Figure 6: Inductive Translation  $(\cdot)^\dagger$  of Terms to Well-boxed Graphs

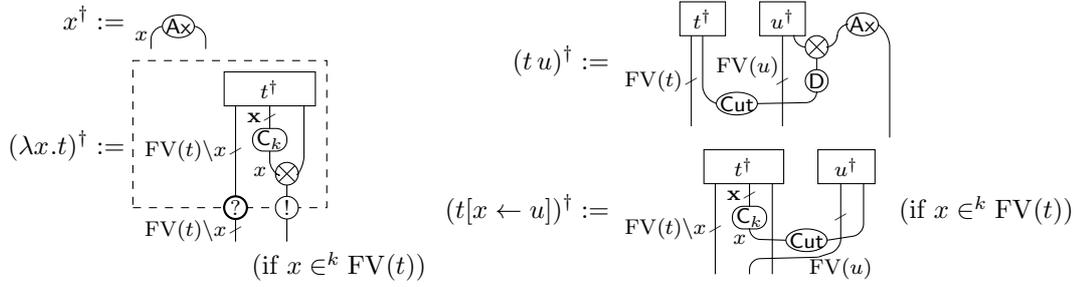


Figure 7: Inductive Translation  $(\cdot)^\dagger_M$  of Evaluation Contexts to Graphs

### 3.2 Translation and Weak Simulation

A weak simulation is built on top of translations of terms and evaluation contexts. The translations  $(\cdot)^\dagger$  are inductively defined in Fig. 6 and Fig. 7. What underlies them is the so-called “call-by-value” translation of intuitionistic logic to linear logic. This translates all and only values to !-boxes that can be copied by rewrite transitions.

The translation  $\text{FV}(t) \frac{\boxed{t^\dagger}}{\vdash}$  of a term  $t$  is a well-boxed graph, where some edges are annotated with variables to help understanding. We continue representing a bunch of edges by a single edge and a strike-out, with annotations denoted by a multiset, and a bunch of nodes by a single thick node. The translation  $M \frac{\boxed{E_M^\dagger}}{\text{FV}_M(E) \vdash}$

of an evaluation context  $E$ , given a multiset  $M$  of variables, is not a well-boxed graph because it has incoming edges. Lem. 3.3 is analogous to Lem. 3.1; their proof is by straightforward induction on  $E$ .

**Lemma 3.3** (decomposition).

$$(E\langle t \rangle)^\dagger = \text{FV}(t) \frac{\boxed{\bar{t}^\dagger}}{\text{FV}_{\text{FV}(t)}(E) \vdash}$$

$$(E\langle E' \rangle)_M^\dagger = M \frac{\boxed{(E')_M^\dagger}}{\text{FV}_M(E) \vdash}$$

$$\text{FV}_{\text{FV}_M(E)}(E) \frac{\boxed{E_{\text{FV}_M(E)}^\dagger}}{\vdash}$$



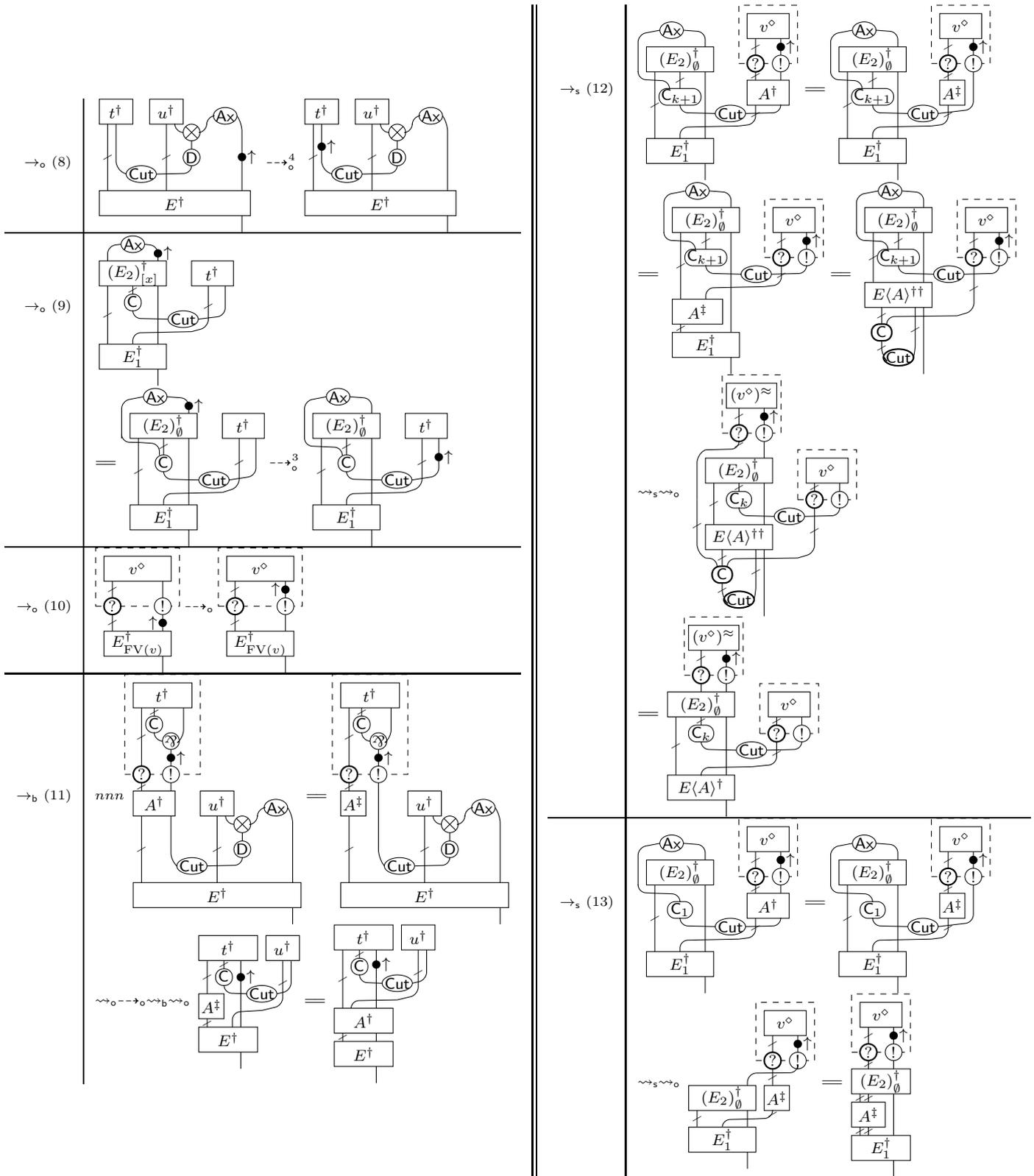


Figure 8: Illustration of Simulation ( $k > 0$ )

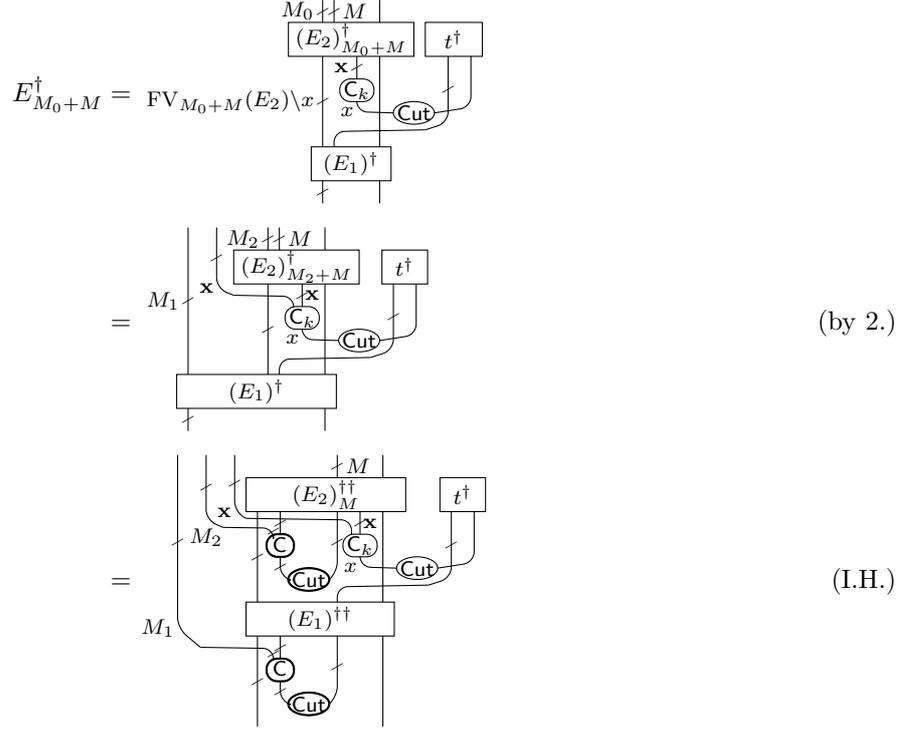


Figure 9: Decomposing  $E_{M_0+M}^\dagger$

*Proof.* The proofs for 1. and 2. are by straightforward inductions on  $A$  and  $E$  respectively. The proof for 3. is by induction on the dimension of  $M_0$ , i.e. the size of the support set  $\text{supp}(M_0)$ . The base case where  $M_0 = \emptyset$  is obvious. In the inductive case, let  $x$  satisfy  $x \in M_0$ . Since  $x$  is captured exactly once in  $E$  by assumption, the evaluation context  $E$  can be decomposed as  $E = E_1 \langle E_2[x \leftarrow t] \rangle$  such that  $x$  is not captured in  $E_1$  or  $E_2$ . The evaluation context  $E_2$  satisfies  $x \in^k \text{FV}_{M_0+M}(E_2)$  for some positive multiplicity  $k$ . Moreover the multiset  $M_0$  can be decomposed as  $M_0 = M_1 + \mathbf{x} + M_2$  such that all the variables in  $M_1$  (resp.  $M_2$ ) are only captured in  $E_1$  (resp.  $E_2$ ). The translation  $E_{M_0+M}^\dagger$  can be decomposed as in Fig. 9. Finally, we let  $E_M^{\dagger\dagger}$  consist of  $(E_2)_M^{\dagger\dagger}, t^\dagger, (E_1)^{\dagger\dagger}$ .  $\square$

## 4 Time Cost Analysis of Rewrites-First Interleaving

### 4.1 Recipe for Time Cost Analysis

Our time cost analysis of the  $\text{DGoIM}_{\rightarrow}$  follows Accattoli’s recipe, described in [2, 1], of analysing complexity of abstract machines. This section recalls the recipe and explains how it applies to the  $\text{DGoIM}_{\rightarrow}$ .

The time cost analysis focuses on how efficiently an abstract machine implements an evaluation strategy. In other words, we are not interested in minimising the number of  $\beta$ -reduction steps simulated by an abstract machine. Our interest is in making the number of transitions of an abstract machine “reasonable,” compared to the number of necessary  $\beta$ -reduction steps determined by a given evaluation strategy.

Accattoli’s recipe assumes that an abstract machine has three groups of transitions: 1) “ $\beta$ -transitions” that correspond to  $\beta$ -reduction in which substitution is delayed, 2) transitions perform substitution, and 3) other “overhead” transitions. We incorporate this classification using the labels  $\mathbf{b}, \mathbf{s}, \mathbf{o}$  of transitions.

Another assumption of the recipe is that, each step of  $\beta$ -reduction is simulated by a single transition of an abstract machine, and so is substitution of each occurrence of a variable. This is satisfied by many known abstract machines including the SAM, however not by the  $\text{DGoIM}_{\rightarrow}$ . The  $\text{DGoIM}_{\rightarrow}$  has “finer” transitions and can take several transitions to simulate a single step of reduction (hence a single transition of the SAM, as we can observe in Thm. 3.5). In spite of this mismatch we can still follow the recipe, thanks to the weak simulation  $\preceq$ . It discloses what transitions of the  $\text{DGoIM}$  exactly correspond to  $\beta$ -reduction and substitution, and gives a concrete number of overhead transitions that the  $\text{DGoIM}_{\rightarrow}$  needs to simulate  $\beta$ -reduction and substitution. The recipe for the time cost analysis is:

1. Examine the number of transitions, by means of the size of input and the number of  $\beta$ -transitions.

2. Estimate time cost of single transitions.
3. Derive a bound of the overall execution time cost.
4. Classify an abstract machine according to its execution time cost.

The last step is accompanied by the following taxonomy of abstract machines introduced in [1].

**Definition 4.1** (classes of abstract machines [1, Def. 7.1]).

1. An abstract machine is *efficient* if its execution time cost is linear in both the input size and the number of  $\beta$ -transitions.
2. An abstract machine is *reasonable* if its execution time cost is polynomial in the input size and the number of  $\beta$ -transitions.
3. An abstract machine is *unreasonable* if it is not reasonable.

The input size in our case is given by the *size*  $|t|$  of a term  $t$ , inductively defined by:

$$\begin{aligned} |x| &:= 1 & |\lambda x.t| &:= |t| + 1 \\ |tu| &:= |t| + |u| + 1 & |t[x \leftarrow u]| &:= |t| + |u| + 1. \end{aligned}$$

Given a sequence  $r$  of transitions (of either the SAM or the  $DGoIM_{\rightarrow}$ ), we denote the number of transitions with a label  $x$  in  $r$  by  $|r|_x$ . Since we use the fixed set  $\{\mathbf{b}, \mathbf{s}, \mathbf{o}\}$  of labels, the length  $|r|$  of the sequence  $r$  is equal to the sum  $|r|_{\mathbf{b}} + |r|_{\mathbf{s}} + |r|_{\mathbf{o}}$ .

## 4.2 Number of Transitions

We first estimate the number of transitions of the SAM, and then derive estimation for the  $DGoIM_{\rightarrow}$ .

**Lemma 4.2** (quantitative bounds for SAM). *Each execution  $e$  from an initial configuration  $(\bar{t}_0, E)_{term}$ , comes with the following inequalities:*

$$\begin{aligned} |e|_{\mathbf{s}} &\leq |e|_{\mathbf{b}} \\ |e|_{\mathbf{o}} &\leq |\bar{t}_0| \cdot (5 \cdot |e|_{\mathbf{b}} + 2) + (3 \cdot |e|_{\mathbf{b}} + 1). \end{aligned}$$

*Proof.* The proof is analogous to the discussion in [2, Sec. 11]. Let  $|e|^{(8)}$ ,  $|e|^{(9)}$  and  $|e|^{(10)}$  be the numbers of transitions (8), (9) and (10) in  $e$ , respectively. The number  $|e|_{\mathbf{o}}$  is equal to the sum of these three numbers.

The transition (11) introduces one explicit substitution, and the other transitions never increase the number of explicit substitution. In particular the transition (12) copies a pure value, that means no explicit substitutions are copied. Therefore we can bound the number of transitions that concern explicit substitutions, and obtain  $|e|^{(9)} \leq |e|_{\mathbf{b}}$  and  $|e|_{\mathbf{s}} \leq |e|_{\mathbf{b}}$ .

Each occurrence of the transition (10) in an execution is either the last transition of the execution or followed by the transitions (11), (12) and (13). This yields the inequality  $|e|^{(10)} \leq |e|_{\mathbf{b}} + |e|_{\mathbf{s}} + 1$  and hence  $|e|^{(10)} \leq 2 \cdot |e|_{\mathbf{b}} + 1$ .

The transition (8) reduces the size of a pure term that is the first component of a configuration. The pure term is always a sub-term of the initial term  $\bar{t}_0$  (Lem. 3.2). This means each maximal subsequence of an execution that solely consists of the transition (8) has at most the length  $|\bar{t}_0|$ . Such a maximal subsequence either occurs at the end of an execution or is followed by transitions other than the transition (8). Therefore the number of these maximal sub-sequences is no more than  $|e|_{\mathbf{b}} + |e|_{\mathbf{s}} + |e|^{(9)} + |e|^{(10)} + 1$  that can be bounded by  $5 \cdot |e|_{\mathbf{b}} + 2$ . A bound of the number  $|e|^{(8)}$  can be given by multiplying these two bounds, namely we obtain  $|e|^{(8)} \leq |\bar{t}_0| \cdot (5 \cdot |e|_{\mathbf{b}} + 2)$ .  $\square$

Combining these bounds for the SAM with the weak simulation  $\preceq$ , we can estimate the number of transitions of the  $DGoIM_{\rightarrow}$  as below.

**Proposition 4.3** (quantitative bounds for  $DGoIM_{\rightarrow}$ ). *Let  $r: s_0 \rightarrow^* s$  be a sequence of transitions of the  $DGoIM_{\rightarrow}$ . If there exists an execution  $(\bar{t}_0, \langle \cdot \rangle)_{term} \rightarrow^* (\bar{t}, E)$  of the SAM such that  $s_0 \preceq (\bar{t}_0, \langle \cdot \rangle)_{term}$  and  $s \preceq (\bar{t}, E)$ , the sequence  $r$  comes with the following inequalities:*

$$\begin{aligned} |r|_{\mathbf{s}} &\leq |r|_{\mathbf{b}} \\ |r|_{\mathbf{o}} &\leq 4 \cdot |\bar{t}_0| \cdot (5 \cdot |r|_{\mathbf{b}} + 2) + (16 \cdot |r|_{\mathbf{b}} + 4). \end{aligned}$$

*Proof.* This is a direct consequence of Lem. 4.2 and Thm. 3.5.  $\square$

### 4.3 Execution Time Cost

We already discussed time cost of single transitions of the DGoIM in Sec. 2.3. It is worth noting that the discussion in Sec. 2.3 is independent of any particular choice of a rewriting and token-passing interleaving strategy.

Thm. 4.4 below gives a bound of execution time cost of the  $DGoIM_{\rightarrow}$ . We can conclude that, according to Accattoli’s taxonomy (see Def. 4.1), the  $DGoIM_{\rightarrow}$  is “efficient” as an abstract machine for the call-by-need evaluation.

**Theorem 4.4** (time cost). *Let  $C, D$  be fixed natural numbers, and  $r: s_0 \rightarrow^* s$  be a sequence of transitions of the  $DGoIM_{\rightarrow}$ . If there exists an execution  $(\bar{t}_0, \langle \cdot \rangle)_{term} \rightarrow^* (\bar{t}, E)$  of the SAM such that  $s_0 \preceq (\bar{t}_0, \langle \cdot \rangle)_{term}$  and  $s \preceq (\bar{t}, E)$ , the total time cost  $T(r)$  of the sequence  $r$  satisfies:*

$$T(r) = \mathcal{O}((|\bar{t}_0| + C) \cdot (|r|_b + D)).$$

*Proof.* We estimated in Sec. 2.3 that time cost of single transitions, except for the rewrite transitions (4) and (6), is constant. Time cost of these rewrite transitions (4) and (6) depends on the number of doors and/or nodes of a !-box.

Since the sequence  $r$  of transitions simulates an execution of the SAM, every !-box concerned in  $r$  arises as the translation of a value  $\bar{v}$ . By definition of the translation  $(\cdot)^\dagger$  (Fig. 6), the graph  $\bar{v}^\dagger$  is a !-box, with as many auxiliary doors as occurrences of free variables in  $\bar{v}$ . The number of auxiliary doors is no more than the number of nodes in the !-box, due to the well-boxed-ness condition, that is linear in the size  $|\bar{v}|$  of the value.

Moreover the value  $\bar{v}$  appears as the first component of a context configuration, and therefore it is a sub-term of the initial term  $\bar{t}_0$  (Lem. 3.2). As a result, time cost of each occurrence of the rewrite transitions (4) and (6) in the sequence  $r$  is linear in the size  $|\bar{v}|$ .

The bound of the total time cost  $T(r)$  of the sequence  $r$  is given by combining these estimations for single transitions with the results of Prop. 4.3.  $\square$

**Corollary 4.5.** *The  $DGoIM_{\rightarrow}$  is an efficient abstract machine, in the sense of Def. 4.1.*

## 5 Conclusions

We introduced the DGoIM, which can interleave token passing with graph rewriting informed by the trajectory of the token. We focused on the rewrites-first interleaving and proved that it enables the DGoIM to implement the call-by-need evaluation strategy. The quantitative analysis of time cost certified that the  $DGoIM_{\rightarrow}$  gives an “efficient” implementation in the sense of Accattoli’s classification. The proof of Thm. 4.4 pointed out that eliminating and copying !-boxes are two main sources of time cost. Our results are built on top of a weak simulation of the SAM, that relates several transitions of the DGoIM to each computational task such as  $\beta$ -reduction and substitution.

The main feature of the DGoIM is the flexible combination of interaction and rewriting. We here briefly discuss how the flexibility can enable the DGoIM to implement evaluation strategies other than the call-by-need.

As mentioned in Sec. 1.2, the passes-only interleaving yields an ordinary token-passing abstract machine that is known to implement the call-by-name evaluation. Because no rewrites are triggered, as oppose to the rewrites-first interleaving, a token not only can pass a principal door but also can go inside a !-box and pass an auxiliary door. These behaviours are in fact not possible with the DGoIM presented in this paper; the transitions and data carried by a token are tailored to the rewrites-first interleaving. To recover an ordinary token-passing machine, we therefore need to add pass transitions that involve auxiliary doors and data structures (so-called “exponential signatures”) that deal with !-boxes, for example.

The only difference between the call-by-need and the call-by-value evaluations lies in when function arguments are evaluated. In the DGoIM, this corresponds to changing a trajectory of a token so that it visits function arguments immediately after it detects function application. Therefore, to implement the call-by-value evaluation, the DGoIM can still use the rewrites-first interleaving, but it should use a modified set of pass transitions. Further refinements, not only of the evaluation strategies but also of the graph representation could yield even more efficient implementation, such as *full lazy evaluation*, as hinted in [26].

Our final remarks concern programming features that have been modelled using token-passing abstract machines. Ground-type constants are handled by attaching memories to either nodes of a graph or a token, in e.g. [20, 18, 3] — this can be seen as a simple form of graph rewriting. Algebraic effects are also accommodated using memories attached to nodes of a graph in token machines [18], but their treatment would be much simplified in the DGoIM as effects are evaluated out of the term via rewriting.

**Acknowledgements.** We are grateful to Ugo Dal Lago and anonymous reviewers for encouraging and insightful comments on earlier versions of this work.

## References

- [1] Beniamino Accattoli. The complexity of abstract machines. In *WPTE 2016*, volume 235 of *EPTCS*, pages 1–15, 2017.
- [2] Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. Distilling abstract machines. In *ICFP 2014*, pages 363–376. ACM, 2014.
- [3] Ugo Dal Lago, Claudia Faggian, Benoît Valiron, and Akira Yoshimizu. Parallelism and synchronization in an infinitary context. In *LICS 2015*, pages 559–572. IEEE, 2015.
- [4] Ugo Dal Lago, Claudia Faggian, Benoît Valiron, and Akira Yoshimizu. The Geometry of Parallelism: classical, probabilistic, and quantum effects. In *POPL 2017*, pages 833–845. ACM, 2017.
- [5] Ugo Dal Lago and Marco Gaboardi. Linear dependent types and relative completeness. In *LICS 2011*, pages 133–142. IEEE Computer Society, 2011.
- [6] Ugo Dal Lago and Barbara Petit. Linear dependent types in a call-by-value scenario. In *PPDP 2012*, pages 115–126. ACM, 2012.
- [7] Ugo Dal Lago and Ulrich Schöpp. Computation by interaction for space-bounded functional programming. *Inf. Comput.*, 248:150–194, 2016.
- [8] Vincent Danos and Laurent Regnier. Local and asynchronous beta-reduction (an analysis of Girard’s execution formula). In *LICS 1993*, pages 296–306. IEEE Computer Society, 1993.
- [9] Vincent Danos and Laurent Regnier. Reversible, irreversible and optimal lambda-machines. *Elect. Notes in Theor. Comp. Sci.*, 3:40–60, 1996.
- [10] Olivier Danvy and Ian Zerny. A synthetic operational account of call-by-need evaluation. In *PPDP 2013*, pages 97–108. ACM, 2013.
- [11] Maribel Fernández and Ian Mackie. Call-by-value lambda-graph rewriting without rewriting. In *ICGT 2002*, volume 2505 of *LNCS*, pages 75–89. Springer, 2002.
- [12] Dan R. Ghica. Geometry of Synthesis: a structured approach to VLSI design. In *POPL 2007*, pages 363–375. ACM, 2007.
- [13] Dan R. Ghica and Alex Smith. Geometry of Synthesis III: resource management through type inference. In *POPL 2011*, pages 345–356. ACM, 2011.
- [14] Dan R. Ghica, Alex Smith, and Satnam Singh. Geometry of Synthesis IV: compiling affine recursion into static hardware. In *ICFP*, pages 221–233, 2011.
- [15] Jean-Yves Girard. Linear logic. *Theor. Comp. Sci.*, 50:1–102, 1987.
- [16] Jean-Yves Girard. Geometry of Interaction I: interpretation of system F. In *Logic Colloquium 1988*, volume 127 of *Studies in Logic & Found. Math.*, pages 221–260. Elsevier, 1989.
- [17] Georges Gonthier, Martín Abadi, and Jean-Jacques Lévy. The geometry of optimal lambda reduction. In *POPL 1992*, pages 15–26. ACM, 1992.
- [18] Naohiko Hoshino, Koko Muroya, and Ichiro Hasuo. Memoryful Geometry of Interaction: from coalgebraic components to algebraic effects. In *CSL-LICS 2014*, pages 52:1–52:10. ACM, 2014.
- [19] John Lamping. An algorithm for optimal lambda calculus reduction. In *POPL 1990*, pages 16–30. ACM Press, 1990.
- [20] Ian Mackie. The Geometry of Interaction machine. In *POPL 1995*, pages 198–208. ACM, 1995.
- [21] Robin Milner. *Communication and concurrency*. PHI Series in Computer Science. Prentice Hall, 1989.
- [22] Koko Muroya, Naohiko Hoshino, and Ichiro Hasuo. Memoryful Geometry of Interaction II: recursion and adequacy. In *POPL 2016*, pages 748–760. ACM, 2016.
- [23] Ulrich Schöpp. Computation-by-interaction with effects. In *APLAS 2011*, volume 7078 of *Lect. Notes Comp. Sci.*, pages 305–321. Springer, 2011.

- [24] Ulrich Schöpp. Call-by-value in a basic logic for interaction. In *APLAS 2014*, volume 8858 of *Lect. Notes Comp. Sci.*, pages 428–448. Springer, 2014.
- [25] Ulrich Schöpp. Organising low-level programs using higher types. In *PPDP 2014*, pages 199–210. ACM, 2014.
- [26] François-Régis Sinot. Call-by-name and call-by-value as token-passing interaction nets. In *TLCA 2005*, volume 3461 of *Lect. Notes Comp. Sci.*, pages 386–400. Springer, 2005.
- [27] François-Régis Sinot. Call-by-need in token-passing nets. *Math. Struct. in Comp. Sci.*, 16(4):639–666, 2006.