Efficient implementation of evaluation strategies via token-guided graph rewriting

Koko Muroya Dan R. Ghica
University of Birmingham, UK
{k.muroya,d.r.ghica}@cs.bham.ac.uk

In implementing evaluation strategies of the lambda-calculus, both correctness and efficiency of implementation are valid concerns. While the notion of correctness is determined by the evaluation strategy, regarding efficiency there is a larger design space that can be explored, in particular the trade-off between space versus time efficiency. We contributed to the study of this trade-off by the introduction of an abstract machine for call-by-need, inspired by Girard's Geometry of Interaction, a machine combining token passing and graph rewriting. This work presents a conservative extension of the machine, to additionally accommodate left-to-right and right-to-left call-by-value strategies. We show soundness and completeness of the extended machine with respect to each of the call-by-need and two call-by-value strategies. Analysing time cost of its execution classifies the machine as "efficient" in Accattoli's taxonomy of abstract machines.

1 Introduction

1.1 Efficiency of Implementing Evaluation Strategies

The lambda-calculus is a simple yet rich model of computation, relying on a single mechanism to activate a function in computation—beta-reduction, that replaces function arguments with actual input. While in the lambda-calculus itself beta-reduction can be applied in an unrestricted way, it is evaluation strategies that determine the way beta-reduction is applied when the lambda-calculus is used as a programming language. Evaluation strategies often imply how intermediate results are copied, discarded, cached or reused. For example, everything is repeatedly evaluated as many times as requested in the call-by-name strategy. In the call-by-need strategy, once a function requests its input, the input is evaluated and the result is cached for later use. The call-by-value strategy evaluates function input and caches the result even if the function does not require the input.

The implementation of any evaluation strategy must be correct, first of all, i.e. it has to produce results as stipulated by the strategy. Once correctness is assured, the next concern is efficiency. One may prefer better space efficiency, or better time efficiency, and it is well known that one can be traded off for the other. For example, time efficiency can be improved by caching more intermediate results, which increases space cost. Conversely, bounding space requires repeating computations, which adds to the time cost. Whereas correctness is well defined for any evaluation strategy, there is a certain freedom in managing efficiency. The challenge here is how to produce a unified framework which is flexible enough to analyse and guide the choices required by this trade-off. Recent studies by Accattoli et al. [3, 2, 1] clearly establish classes of efficiency for a given evaluation strategy. They characterise efficiency by means of the number of beta-reduction applications required by the strategy, and introduce two efficiency classes, namely "efficient" and "reasonable." The expected efficiency of an abstract machine gives us a starting point to quantitatively analyse the trade-offs required in an implementation.

Submitted to:
WPTE 2017

1.2 GoI-style Token Passing, Interleaved with Graph Rewriting

We employ Girard's Geometry of Interaction (GoI) [10], a semantics of linear logic proofs, as a framework for studying the trade-off between time and space efficiency. In particular we focus on GoI-style abstract machines for the lambda-calculus, pioneered by Danos and Regnier [6] and Mackie [13]. These machines evaluate a term of the lambda-calculus by translating the term to a graph, a network of simple transducers, which executes by passing a data-carrying token around.

The token simulates graph rewriting without actually rewriting, which is in fact a particular instance of the trade-off we mentioned above. The token-passing machines keep the underlying graph fixed and use the data stored in the token to route it. They therefore favour space efficiency at the cost of time efficiency. The same computation is repeated when, instead, intermediate results could have been cached by saving copies of certain sub-graphs representing values.

Our intention is to lift the GoI-style token passing to a framework to analyse the trade-off of efficiency, by strategically interleaving it with graph rewriting. The key idea is that the token holds control over graph rewriting, by visiting redexes and triggering rewrite rules. Graph rewriting offers fine control over caching and sharing intermediate results, however fetching cached results can increase the size of the graph. In short, introduction of graph rewriting sacrifices space while favouring time efficiency. We expect the flexibility given by a fine-grained control over interleaving will enable a careful balance between space and time efficiency.

This idea was first introduced in [15], by developing an abstract machine that interleaves token passing with as much graph rewriting as possible. We showed this interleaving strategy gives an abstract machine which implements call-by-need evaluation, which is classified as "efficient". We further develop this idea by proposing a conservative extension of the graph-rewriting abstract machine, to accommodate other evaluation strategies, namely left-to-right and right-to-left call-by-value. In our framework, both call-by-value strategies involve similar tactics for caching intermediate results as the call-by-need strategy, with the only difference being the timing of cache creation.

1.3 Contributions

We extend the token-guided graph-rewriting abstract machine for the call-by-need strategy [15] to the left-to-right and right-to-left call-by-value strategies. The presentation of the machine is revised by using term graphs instead of proof nets [9], to make clearer sense of evaluation strategies in the graphical representation of terms. The extension is conservative, by introducing nodes that correspond to different evaluation strategies, rather than modifying the behaviour of existing nodes to suite different evaluation strategy demands.

We prove the soundness and completeness of the extended machine with respect to the call-by-need strategy and the two call-by-value strategies, separately, using a "sub-machine" semantics, where the word 'sub' indicates both a focus on substitution and its status as an intermediate representation. The sub-machine semantics is based on Sinot's "token-passing" semantics [18, 19] that makes explicit the two main tasks of abstract machines: searching redexes and substituting variables. The time cost analysis classifies the machine as "efficient" in Accattoli's taxonomy of abstract machines [1].

2 A term calculus with sub-machine semantics

We aim at three evaluation strategies of the lambda-calculus, namely call-by-need, left-to-right call-by-value, and right-to-left call-by-value. The following is an untyped term calculus that accommodates these

strategies by dedicated constructors for function application, namely @ (call-by-need), $\overrightarrow{@}$ (left-to-right call-by-value) and $\overleftarrow{@}$ (right-to-left call-by-value).

$$\begin{aligned}
t,u &::= x \mid \lambda x.t \mid t \mathbin{@} u \mid t \mathbin{\overrightarrow{@}} u \mid t \mathbin{\overleftarrow{@}} u \mid t[x \leftarrow u] && \text{(terms)}\\
v &::= \lambda x.t && \text{(values)}\\
A &::= \langle\cdot\rangle \mid A[x \leftarrow t] && \text{(answer contexts)}\\
E &::= \langle\cdot\rangle \mid E \mathbin{@} t \mid E \mathbin{\overrightarrow{@}} t \mid A\langle v\rangle \mathbin{\overrightarrow{@}} E \mid t \mathbin{\overleftarrow{@}} E \mid E \mathbin{\overleftarrow{@}} A\langle v\rangle \mid E[x \leftarrow t] \mid E\langle x\rangle[x \leftarrow E] && \text{(evaluation contexts)}
\end{aligned}$$

The term calculus uses all strategies so that we do not have to present three almost identical calculi. But we are not interested in their interaction, but in each strategy separately. In the rest of the paper, we therefore assume that each term contains function applications of a single strategy. The calculus accommodates explicit substitutions $[x \leftarrow u]$. A term with no explicit substitutions is said to be "pure."

The sub-machine semantics is used to establish the soundness of the graph-rewriting abstract machine. It is an adaptation of Sinot's lambda-term rewriting system [18, 19], used to analyse a token-guided rewriting system for interaction nets. It imitates an abstract machine by making explicit the process of searching for a redex and of decomposing the meta-level substitution into on-demand linear substitution, also resembling a storeless abstract machine (e.g. [7, Fig. 8]). However the semantics is still too "abstract" as an abstract machine, in the sense that it works modulo alpha-equivalence to avoid variable captures.

Fig. 1 defines the sub-machine semantics of our calculus. It is given by labelled relations between *enriched* terms $E\langle(\!|t|\!)\rangle$. In an enriched term $E\langle(\!|t|\!)\rangle$, a sub-term t is not plugged directly into the evaluation context, but into a "window" $(\!|\cdot|\!)$ which makes it syntactically obvious where the reduction context is situated. Forgetting the window turns an enriched term into an ordinary term. Basic rules $\mapsto$ are labelled with β, σ or ε. The basic rules (2), (5) and (8), labelled with β, apply beta-reduction and delay substitution of a bound variable. Substitution is done one by one, and on demand, by the basic rule (10) with label σ. Each application of the basic rule (10) replaces exactly one bound variable with a value, and keeps a copy of the value for later use. All other basic rules, with label ε, search for a redex by moving the window without changing the underlying term. Finally, reduction is defined by congruence of basic rules with respect to evaluation contexts, and labelled accordingly. Any basic rules and reductions are indeed between enriched terms, because the window $(\!|\cdot|\!)$ is never duplicated or discarded.

An *evaluation* of a pure term t (i.e. a term with no explicit substitution) is a sequence of reductions starting from $\langle(\!|t|\!)\rangle$, that we simply write $(\!|t|\!)$. In any evaluation, a sub-term in the window $(\!|\cdot|\!)$ is always pure.

3 Token-guided graph-rewriting machine

3.1 Graph states

A graph is given by a set of nodes and a set of directed edges. Nodes are classified into *proper* nodes and *link* nodes. Each edge is directed, and at least one of its two endpoints is a link node. An *interface* of a graph is given by two sets of link nodes, namely *input* and *output*. Each link node is a source of at most one edge, and a target of at most one edge. Input links are the only links that are not a target of any edge, and output links are the only ones that are not a source of any edge. When a graph G has n input link nodes and m output link nodes, we sometimes write $G(n,m)$ to emphasise its interface. If a graph has exactly one input, we refer to the input link node as "root."

Basic rules $\mapsto_\beta$, $\mapsto_\sigma$ and $\mapsto_\varepsilon$:

$$(\!|t \mathbin{@} u|\!) \mapsto_\varepsilon (\!|t|\!) \mathbin{@} u \tag{1}$$
$$A\langle(\!|\lambda x.t|\!)\rangle \mathbin{@} u \mapsto_\beta A\langle(\!|t|\!)[x \leftarrow u]\rangle \tag{2}$$
$$(\!|t \mathbin{\overrightarrow{@}} u|\!) \mapsto_\varepsilon (\!|t|\!) \mathbin{\overrightarrow{@}} u \tag{3}$$
$$A\langle(\!|\lambda x.t|\!)\rangle \mathbin{\overrightarrow{@}} u \mapsto_\varepsilon A\langle\lambda x.t\rangle \mathbin{\overrightarrow{@}} (\!|u|\!) \tag{4}$$
$$A\langle\lambda x.t\rangle \mathbin{\overrightarrow{@}} A'\langle(\!|v|\!)\rangle \mapsto_\beta A\langle(\!|t|\!)[x \leftarrow A'\langle v\rangle]\rangle \tag{5}$$
$$(\!|t \mathbin{\overleftarrow{@}} u|\!) \mapsto_\varepsilon t \mathbin{\overleftarrow{@}} (\!|u|\!) \tag{6}$$
$$t \mathbin{\overleftarrow{@}} A\langle(\!|v|\!)\rangle \mapsto_\varepsilon (\!|t|\!) \mathbin{\overleftarrow{@}} A\langle v\rangle \tag{7}$$
$$A\langle(\!|\lambda x.t|\!)\rangle \mathbin{\overleftarrow{@}} A'\langle v\rangle \mapsto_\beta A\langle(\!|t|\!)[x \leftarrow A'\langle v\rangle]\rangle \tag{8}$$
$$E\langle(\!|x|\!)\rangle[x \leftarrow A\langle u\rangle] \mapsto_\varepsilon E\langle x\rangle[x \leftarrow A\langle(\!|u|\!)\rangle] \tag{9}$$
$$E\langle x\rangle[x \leftarrow A\langle(\!|v|\!)\rangle] \mapsto_\sigma A\langle E\langle(\!|v|\!)\rangle[x \leftarrow v]\rangle \tag{10}$$

Reductions $\multimap_\beta$, $\multimap_\sigma$ and $\multimap_\varepsilon$: $\dfrac{\tilde{t} \mapsto_\chi \tilde{u}}{E\langle\tilde{t}\rangle \multimap_\chi E\langle\tilde{u}\rangle}$ $(\chi \in \{\beta, \sigma, \varepsilon\})$

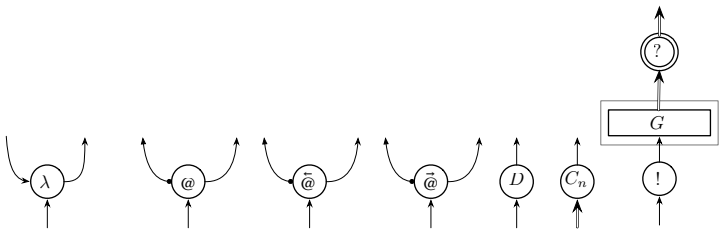
Figure 1: "Sub-machine" operational semantics

The idea of using link nodes, as distinguished from proper nodes, comes from a graphical formalisation of string diagrams [12]. String diagrams consist of "boxes" that are connected to each other by "wires." In the formalisation, boxes are modelled by "box-vertices" (corresponding to proper nodes in our case), and wires are modelled by consecutive edges connected via "wire-vertices" (corresponding to link nodes in our case). The segmentation of wires into edges can introduce an arbitrary number of consecutive link nodes, however these consecutive link nodes are identified by the notion of "wire homeomorphism." We will later discuss these consecutive link nodes, from the perspective of the graph-rewriting machine. From now on we simply call a proper node "node," and a link node "link."

In drawing graphs, we follow the convention that input links are placed at the bottom and output links are at the top, and links are usually not drawn explicitly. The latter point means that edges are simply drawn from a node to a node, with intermediate links omitted. In particular if an edge is connected to an interface link, the edge is drawn as an open edge missing an endpoint. Additionally, we use a double-stroke edge/node to represent a bunch of parallel edges/nodes.

Nodes are labelled, and a node with a label X is called an "X-node." We use two sorts of labels. One sort corresponds to the constructors of the calculus presented in Sec. 2, namely λ (abstraction), @ (call-by-need application), $\overrightarrow{@}$ (left-to-right call-by-value application) and $\overleftarrow{@}$ (right-to-left call-by-value application). These three application nodes are the novelty of this work. The token, travelling in a graph, reacts to these nodes in different ways, and hence implements different evaluation orders. We believe that this is a more extensible way to accommodate different evaluation orders, than to let the token react to the same node in different ways depending on situation. The other sort consists of !, ?, D and C_n for any natural number n, used in the management of copying sub-graphs. This sort is inspired by proof nets of the multiplicative and exponential fragment of linear logic [9], where C_n-nodes generalise the standard binary contraction and incorporate weakening.

The number of input/output and incoming/outgoing edges for a node is determined by the label, as indicated below:



We distinguish two outputs of each of the three application nodes (@, $\overrightarrow{@}$ and $\overleftarrow{@}$), calling one "composition output" and the other "argument output" (cf. [4]). A bullet • in the diagram above is used to specify a function output. Additionally, the outline box indicates a sub-graph $G(1,m)$ ("!-box") that is connected to one !-node ("principal door") and m ?-nodes ("auxiliary doors"). This !-box structure, taken from proof nets, aids duplication of sub-graphs by specifying those that can be copied.

We define a graph-rewriting abstract machine as a labelled transition system between *graph states*.

Definition 3.1 (Graph states)**.** A *graph state* $((G(1,0),e),\delta)$ is formed of a graph $G(1,0)$ with its distinguished link e, and token data $\delta = (d,f,S,B)$ that consists of:

- a *direction*, defined by $d ::= \uparrow \mid \downarrow$
- a *rewrite flag*, defined by $f ::= \Box \mid \lambda \mid !$
- a *computation stack*, defined by $S ::= \Box \mid \star : S \mid \lambda : S \mid @ : S$
- a *box stack*, defined by $B ::= \Box \mid \star : B \mid ! : B \mid \diamond : B \mid e' : B$, where e' is any link of the graph G.

The distinguished link e in the above definition is called the "position" of the token. A token reacts to a node in a graph using its data, which determines its path. The *initial* state $\mathit{Init}(G)$ on a graph G is given by $((G,e_0),(\uparrow,\Box,\Box,\star:\Box))$, and the *final* state $\mathit{Final}(G)$ on the graph G is given by $((G,e_0),(\downarrow,\Box,\Box,!:\Box))$, where e_0 is the root of G. An *execution* on a graph G is a sequence of transitions starting from the initial state $\mathit{Init}(G)$.

3.2 Transitions

Each transition $((G,e),\delta) \to_\chi ((G',e'),\delta')$ between graph states is labelled by either β, σ or ε. Transitions are deterministic, and classified into *pass* transitions that search for redexes and trigger rewriting, and *rewrite* transitions that actually rewrite a graph as soon as a redex is found.

A pass transition $((G,e),(d,\Box,S,B)) \to_\varepsilon ((G,e'),(d',f',S',B'))$, always labelled with ε, applies to a state whose rewrite flag is $\Box$. It simply moves the token over one node, and updates its data by modifying the top elements of stacks, while keeping an underlying graph unchanged. When the token passes a λ-node or a !-node, a rewrite flag is changed to λ or !, which triggers rewrite transitions.

Fig. 2 defines pass transitions, by showing only the relevant node for each transition. The position of the token is drawn as a black triangle, pointing towards the direction of the token. In the figure, $X \neq \star$, and n is a natural number. The pass transition over a C_{n+1}-node pushes the old position e, a link node, to a box stack.

The way the token reacts to application nodes (@, $\overrightarrow{@}$ and $\overleftarrow{@}$) corresponds to the way the window $(\!|\cdot|\!)$ moves in evaluating these function applications in the sub-machine semantics (Fig. 1). When the token moves on to the composition output of an application node, the top element of a computational stack is either @ or $\star$. The element $\star$ makes the token return from a λ-node, which corresponds to reducing the function part of application to a value (i.e. abstraction). The element @ lets the token proceed at a λ-node,

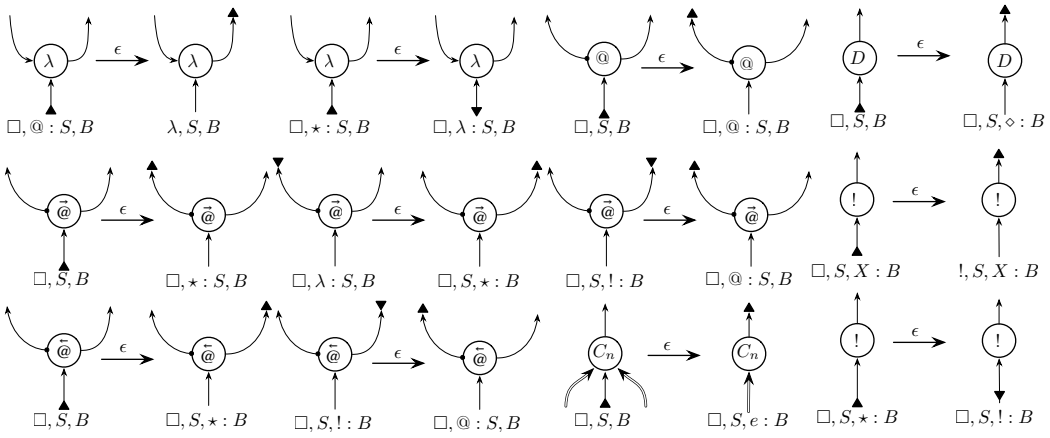


Figure 2: Pass transitions

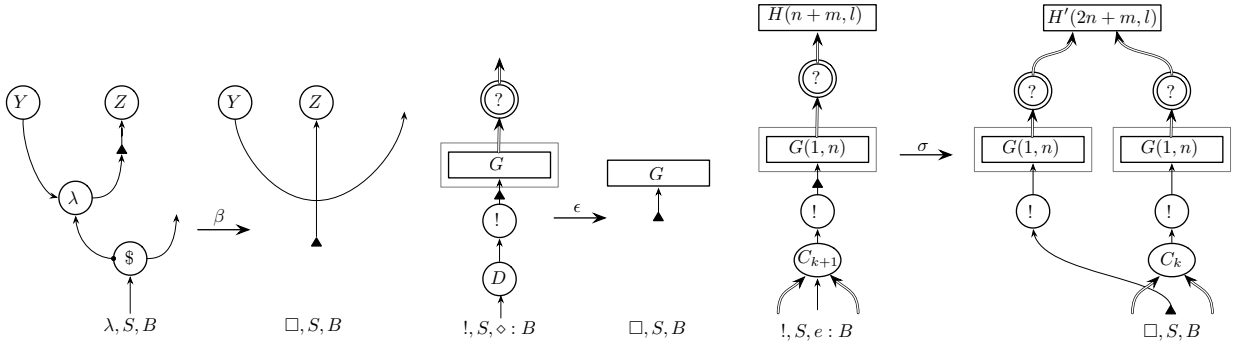


Figure 3: Rewrite transitions

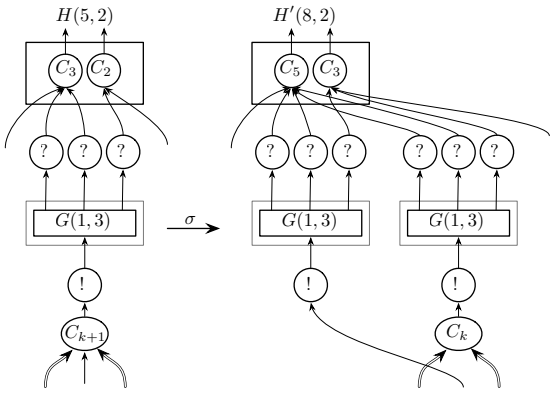
raises the rewrite flag λ, and hence triggers a rewrite transition that corresponds to beta-reduction. The call-by-value application nodes ($\overrightarrow{@}$ and $\overleftarrow{@}$) send the token to their argument output, pushing the element $\star$ to a box stack. This makes the token bounce at a !-node and return to the application node, which corresponds to evaluating the argument part of function application to a value. Finally, pass transitions through D-nodes, C_n-nodes and !-nodes prepare copying of values, and eventually raise the rewrite flag ! that triggers on-demand duplication.

A rewrite transition $((G,e),(d,f,S,B)) \to_\chi ((G',e'),(d',f',S,B'))$, labelled with $\chi \in \{\beta,\sigma,\varepsilon\}$, applies to a state whose rewrite flag is either λ or !. It changes a specific sub-graph while keeping its interface, changes the position accordingly, and pops an element from a box stack. Fig. 3 defines rewrite transitions by showing a sub-graph ("redex") to be rewritten. Before we go through each rewrite transition, we note that rewrite transitions are not exhaustive in general, as a graph may not match a redex even though a rewrite flag is raised. However we will see that there is no failure of transitions in implementing the term calculus.

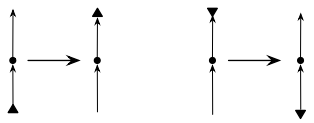
The first rewrite transition in Fig. 3, with label β, occurs when a rewrite flag is λ. It implements beta-reduction by eliminating a pair of an abstraction node (λ) and an application node ($\$ \in \{@, \overrightarrow{@}, \overleftarrow{@}\}$ in the figure). Outputs of the λ-node are required to be connected to arbitrary nodes (labelled with Y and Z in the figure), so that edges between links are not introduced, preserving the desired shape of the graph.

The other rewrite transitions are for the rewrite flag !, and they together realise the copying process of a sub-graph (namely a !-box). The second rewrite transition in Fig. 3, labelled with ε, finishes off each copying process by eliminating doors of the !-box G. It sets the root of G as the new position of the token, and pops the top element $\diamond$ of a box stack. This rewrite transition also does not introduce an edge between links.

The last rewrite transition in the figure, with label σ, actually copies a !-box. It requires the top element e of the old box stack to be one of input links of the C_{k+1}-node (where k is a natural number). The link e is popped from the box stack and becomes the new position of the token, and the C_{k+1}-node becomes a C_k-node by keeping all the inputs except for the link e. The sub-graph $H(n+m,l)$ consists of l parallel C-nodes that altogether have $n+m$ inputs. Among these inputs, n are connected to auxiliary doors of the !-box $G(1,n)$, and m are connected to nodes that are not in the redex. The sub-graph $H(n+m,l)$ is turned into $H'(2n+m,l)$ by introducing n inputs to these C-nodes as follows: if an auxiliary door of the !-box G is connected to a C-node in H, two copies of the auxiliary door are both connected the corresponding C-node in H'. Therefore the two sub-graphs consist of the same number l of C-nodes, whose indegrees are possibly increased. The m inputs, connected to nodes outside a redex, are kept unchanged. For example, copying a graph $G(1,3)$ for $H(5,2)$ will give an $H'(8,2)$ as shown below:



When a graph has an edge between links, the token is just passed along, with the data unchanged:



With this pass transition over a link at hand, the equivalence relation between graphs that identifies consecutive links with a single link—so-called "wire homeomorphism" [12]—lifts to a weak bisimulation between graph states. Therefore, behaviourally, we can safely ignore consecutive links.

From the perspective of time cost analysis, we benefit from the fact that rewrite transitions are designed not to introduce any edge between links. This means, by assuming that an execution starts with a graph with no consecutive links, we can analyse time cost of the execution without caring the extra pass transition over a link.

4 Implementation of evaluation strategies

4.1 Inductive translations to graphs

The implementation of the term calculus, by means of the dynamic GoI, starts with translating (enriched) terms into graphs. The definition of the translation uses multisets of variables, to track how many times each variable occurs in a term. We assume that terms are alpha-converted in a form in which all binders introduce distinct variables.

Notation (Multiset)**.** The empty multiset is denoted by $\emptyset$, and the sum of two multisets M and M' is denoted by $M+M'$. We write $x \in^k M$ if the multiplicity of x in a multiset M is k. Removing *all* x from a multiset M yields the multiset $M\backslash x$, e.g. $[x,x,y]\backslash x = [y]$. We abuse the notation and refer to a multiset $[x,\dots,x]$ of a finite number of x's, simply as x.

Definition 4.1 (Free variables)**.** The map FV of terms to multisets of variables is inductively defined by:

$$\begin{aligned} FV(x) &:= [x], \\ FV(\lambda x.t) &:= FV(t)\backslash x, \\ FV(t\,\$\,u) &:= FV(t)+FV(u), && (\$ \in \{@, \overrightarrow{@}, \overleftarrow{@}\}) \\ FV(t[x \leftarrow u]) &:= (FV(t)\backslash x)+FV(u). \end{aligned}$$

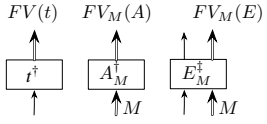
Similarly, given a multiset M of variables, the map FV_M of evaluation contexts to multisets of variables is inductively defined by:

$$\begin{aligned} FV_M(\langle\cdot\rangle) &:= M, \\ FV_M(E\,@\,t) = FV_M(E\,\overrightarrow{@}\,t) &:= FV_M(E)+FV(t), \\ FV_M(A\langle v\rangle\,\overrightarrow{@}\,E) &:= FV(A\langle v\rangle)+FV_M(E), \\ FV_M(t\,\overleftarrow{@}\,E) &:= FV(t)+FV_M(E), \\ FV_M(E\,\overleftarrow{@}\,A\langle v\rangle) &:= FV_M(E)+FV(A\langle v\rangle), \\ FV_M(E[x \leftarrow t]) &:= (FV_M(E)\backslash x)+FV(t), \\ FV_M(E'\langle x\rangle[x \leftarrow E]) &:= (FV(E'\langle x\rangle)\backslash x)+FV_M(E). \end{aligned}$$

A term t is said be *closed* if $FV(t)=\emptyset$. Consequences of the above definition are the following equations, where M' is not captured in E.

$$FV(E\langle t\rangle) = FV_{FV(t)}(E), \qquad FV_M(E\langle E'\rangle) = FV_{FV_M(E')}(E), \qquad FV_{M+M'}(E) = FV_M(E)+M'.$$

We give translations of terms, answer contexts, and evaluation contexts separately. Fig. 4 and Fig. 5 define two mutually recursive translations $(\cdot)^\dagger$ and $(\cdot)^\ddagger$, the first one for terms and answer contexts, and the second one for evaluation contexts. In the figures, $\$ \in \{@, \overrightarrow{@}, \overleftarrow{@}\}$, and m is the multiplicity of x. The general form of the translations is as below.



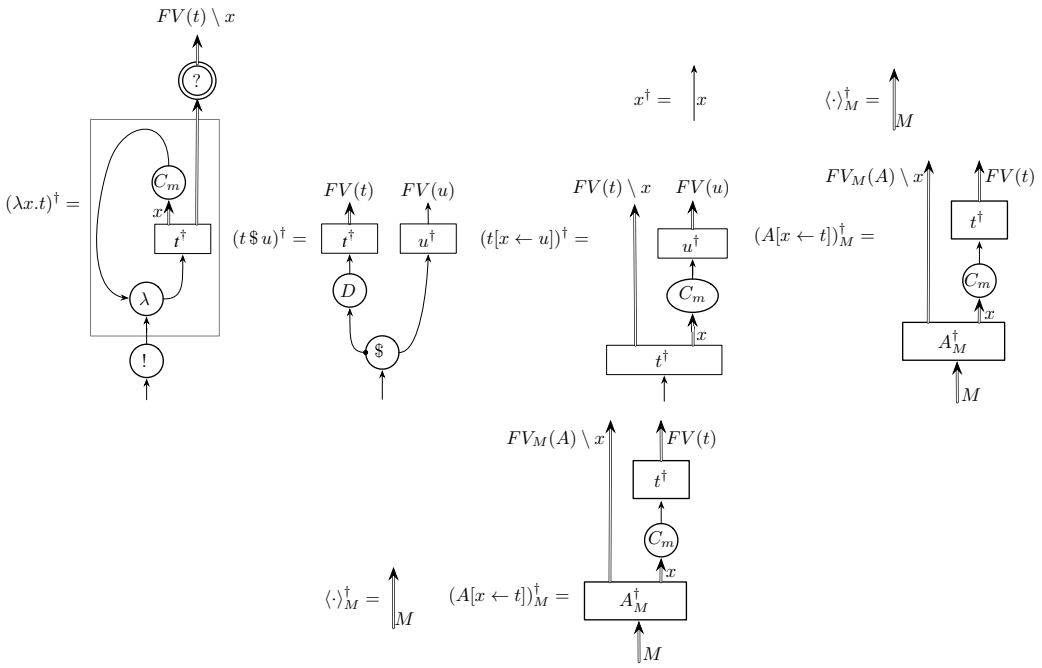
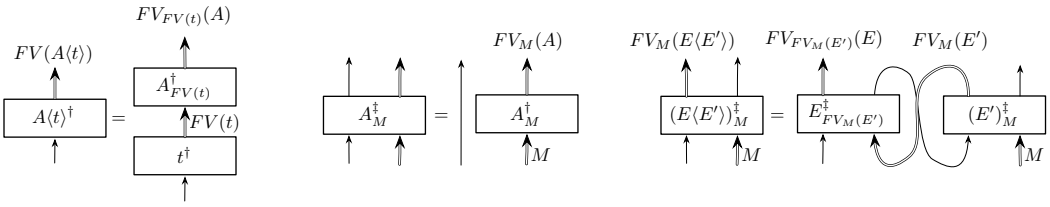


Figure 4: Inductive translation of terms and answer contexts

The annotation of double-stroke edges means each edge of a bunch is labelled with an element of the annotating multiset, in a one-to-one manner. In particular if a double-stroke edge is annotated by a variable x, all edges in the bunch are annotated by the variable x. These annotations are only used to define the translations, and are subsequently ignored during execution.

The translations are based on the so-called "call-by-value" translation of linear logic to intuitionistic logic (studied in e.g. [14]). Notably only the translation of abstraction can be accompanied by a !-box, which captures the fact that only values (i.e. abstractions) can be duplicated (see the basic rule (10) in Fig. 1).

The two mutually recursive translations $(\cdot)^\dagger$ and $(\cdot)^\ddagger$ are related by the following decompositions, which can be checked by straightforward induction.



Note that the decomposition property does not hold for $E\langle t\rangle$ in general, because a translation $(A\langle\lambda x.t\rangle \overrightarrow{@} E)^\ddagger_M$ lacks a !-box structure, compared to a translation $(A\langle\lambda x.t\rangle \overrightarrow{@} u)^\dagger$.

4.2 Weak simulation

On top of the inductive translations of terms and contexts, we define a binary relation between closed enriched terms and graph states.

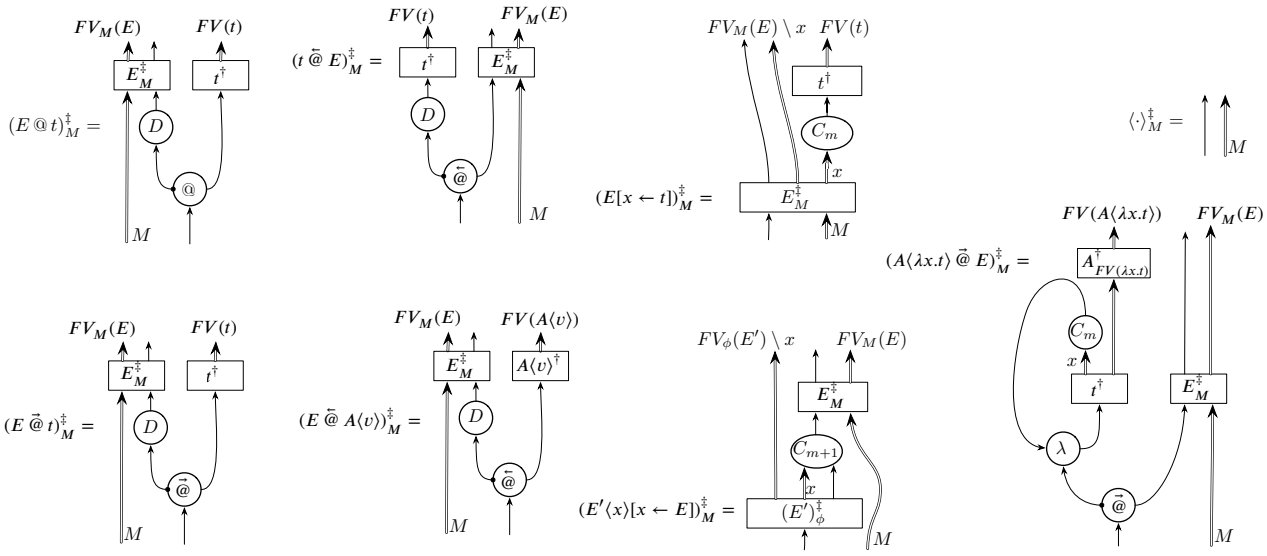
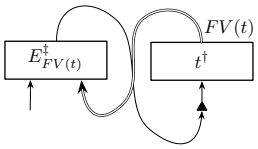


Figure 5: Inductive translation of evaluation contexts

Definition 4.2 (Binary relation $\preceq$)**.** The binary relation $\preceq$ is defined by $E\langle\!(t)\!\rangle \preceq ((E^{\ddagger} \circ t^{\dagger}, e), (\uparrow, \Box, S, B))$, where: (i) $E\langle\!(t)\!\rangle$ is a closed enriched term, and $(E^{\ddagger} \circ t^{\dagger}, e)$ is given by  with no edges between links, and (ii) there is an execution $\mathit{Init}(E^{\ddagger} \circ t^{\dagger}) \to^{*} ((E^{\ddagger} \circ t^{\dagger}, e), (\uparrow, \Box, S, B))$ such that the position e appears only at the last of the sequence.

A special case is $(\!|t|\!) \preceq \mathit{Init}(t^{\dagger})$, which relates the starting points of an evaluation and an execution. We require the graph $E^{\ddagger} \circ t^{\dagger}$ to have no edges between links, which is based on the discussion at the end of Sec. 3.2 and essential for time cost analysis. Although the definition of the translations relies on edges between links (e.g. the translation $x^{\dagger}$), we can safely replace any consecutive links in the composition of translations $E^{\ddagger}$ and $t^{\dagger}$ with a single link, and yield the graph $E^{\ddagger} \circ t^{\dagger}$ with no consecutive links.

The binary relation $\preceq$ gives a weak simulation of the sub-machine semantics by the graph-rewriting machine. The weakness, i.e. the extra transitions compared with reductions, comes from the locality of pass transitions and the bureaucracy of managing !-boxes.

Theorem 4.3 (Weak simulation with global bound)**.**

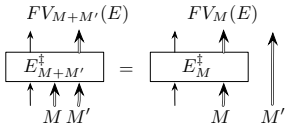
1. *If $E\langle\!(t)\!\rangle \multimap_{\chi} E'\langle\!(t')\!\rangle$ and $E\langle\!(t)\!\rangle \preceq ((E^{\ddagger} \circ t^{\dagger}, e), \delta)$, there exists a graph state $(((E')^{\ddagger} \circ (t')^{\dagger}, e'), \delta')$ and a number $n \leq 3$ such that $((E^{\ddagger} \circ t^{\dagger}, e), \delta) \to_{\varepsilon}^{n} \to_{\chi} (((E')^{\ddagger} \circ (t')^{\dagger}, e'), \delta')$ and $E'\langle\!(t')\!\rangle \preceq (((E')^{\ddagger} \circ (t')^{\dagger}, e'), \delta')$.*
2. *If $A\langle\!(v)\!\rangle \preceq ((A^{\ddagger} \circ v^{\dagger}, e), \delta)$, the graph state $((A^{\ddagger} \circ v^{\dagger}, e), \delta)$ is initial, from which only the transition $\mathit{Init}(A^{\ddagger} \circ v^{\dagger}) \to_{\varepsilon} \mathit{Final}(A^{\ddagger} \circ v^{\dagger})$ is possible.*

Proof outline. We here show how the token data changes in simulating each case of the reduction, aiming at disclosing the number of transitions and when rewrite transitions are triggered. In the following, each

case of reduction is followed by a sequence of transitions that simulates it.

$$E_0\langle(\!|t \mathbin{@} u|\!)\rangle \multimap_\varepsilon E_0\langle(\!|t|\!) \mathbin{@} u\rangle \tag{1}$$
$$(\uparrow,\Box,S,B) \to_\varepsilon (\uparrow,\Box,@:S,B) \to_\varepsilon (\uparrow,\Box,@:S,\diamond:B)$$
$$E_0\langle A\langle(\!|\lambda x.t|\!)\rangle \mathbin{@} u\rangle \multimap_\beta E_0\langle A\langle(\!|t|\!)[x \leftarrow u]\rangle\rangle \tag{2}$$
$$(\uparrow,\Box,@:S,\diamond:B) \to_\varepsilon (\uparrow,!,@:S,\diamond:B) \to_\varepsilon (\uparrow,\Box,@:S,B) \to_\varepsilon (\uparrow,\lambda,S,B) \to_\beta (\uparrow,\Box,S,B)$$
$$E_0\langle(\!|t \mathbin{\overrightarrow{@}} u|\!)\rangle \multimap_\varepsilon E_0\langle(\!|t|\!) \mathbin{\overrightarrow{@}} u\rangle \tag{3}$$
$$(\uparrow,\Box,S,B) \to_\varepsilon (\uparrow,\Box,\star:S,B) \to_\varepsilon (\uparrow,\Box,\star:S,\diamond:B)$$
$$E_0\langle A\langle(\!|\lambda x.t|\!)\rangle \mathbin{\overrightarrow{@}} u\rangle \multimap_\varepsilon E_0\langle A\langle\lambda x.t\rangle \mathbin{\overrightarrow{@}} (\!|u|\!)\rangle \tag{4}$$
$$(\uparrow,\Box,\star:S,\diamond:B) \to_\varepsilon (\uparrow,!,\star:S,\diamond:B) \to_\varepsilon (\uparrow,\Box,\star:S,B) \to_\varepsilon (\downarrow,\Box,\lambda:S,B) \to_\varepsilon (\uparrow,\Box,S,\star:B)$$
$$E_0\langle A\langle\lambda x.t\rangle \mathbin{\overrightarrow{@}} A'\langle(\!|v|\!)\rangle\rangle \multimap_\beta E_0\langle A\langle(\!|t|\!)[x \leftarrow A'\langle v\rangle]\rangle\rangle \tag{5}$$
$$(\uparrow,\Box,S,\star:B) \to_\varepsilon (\downarrow,\Box,S,!:B) \to_\varepsilon (\uparrow,\Box,@:S,B) \to_\varepsilon (\uparrow,\lambda,S,B) \to_\beta (\uparrow,\Box,S,B)$$
$$E_0\langle(\!|t \mathbin{\overleftarrow{@}} u|\!)\rangle \multimap_\varepsilon E_0\langle t \mathbin{\overleftarrow{@}} (\!|u|\!)\rangle \tag{6}$$
$$(\uparrow,\Box,S,B) \to_\varepsilon (\uparrow,\Box,S,\star:B)$$
$$E_0\langle t \mathbin{\overleftarrow{@}} A\langle(\!|v|\!)\rangle\rangle \multimap_\varepsilon E_0\langle(\!|t|\!) \mathbin{\overleftarrow{@}} A\langle v\rangle\rangle \tag{7}$$
$$(\uparrow,\Box,S,\star:B) \to_\varepsilon (\downarrow,\Box,S,!:B) \to_\varepsilon (\uparrow,\Box,@:S,B) \to_\varepsilon (\uparrow,\Box,@:S,\diamond:B)$$
$$E_0\langle A\langle(\!|\lambda x.t|\!)\rangle \mathbin{\overleftarrow{@}} A'\langle v\rangle\rangle \multimap_\beta E_0\langle A\langle(\!|t|\!)[x \leftarrow A'\langle v\rangle]\rangle\rangle \tag{8}$$
$$(\uparrow,\Box,@:S,\diamond:B) \to_\varepsilon (\uparrow,!,@:S,\diamond:B) \to_\varepsilon (\uparrow,\Box,@:S,B) \to_\varepsilon (\uparrow,\lambda,S,B) \to_\beta (\uparrow,\Box,S,B)$$
$$E_0\langle E\langle(\!|x|\!)\rangle[x \leftarrow A\langle u\rangle]\rangle \multimap_\varepsilon E_0\langle E\langle x\rangle[x \leftarrow A\langle(\!|u|\!)\rangle]\rangle \tag{9}$$
$$(\uparrow,\Box,S,B) \to_\varepsilon (\uparrow,\Box,S,e:B)$$
$$E_0\langle E\langle x\rangle[x \leftarrow A\langle(\!|v|\!)\rangle]\rangle \multimap_\sigma E_0\langle A\langle E\langle(\!|v|\!)\rangle[x \leftarrow v]\rangle\rangle \tag{10}$$
$$(\uparrow,\Box,S,e:B) \to_\varepsilon (\uparrow,!,S,e:B) \to_\sigma (\uparrow,\Box,S,B)$$

Tracking underlying graphs relies on the fact that reductions with labels β and σ work modulo alpha-equivalence to avoid name captures. In above reductions, we assume that (i) free variables of u (resp. $A'\langle v\rangle$) is not captured by A in the reduction (2) (resp. (5) and (8)), and (ii) the variable x is not captured by E and free variables of E is not captured by A. These assumptions are exploited together with the following decomposition of translations, in which M' is not captured in E.



Additionally in the above decomposition, if $x \in^k M$ is captured by E, any input annotated with x is connected to a C-node. □

4.3 Execution cost analysis

We analyse how time-efficiently the token-guided graph-rewriting machine implements evaluation strategies, following the methodology developed by Accattoli et al. [2, 5, 1]. The methodology tracks the number of beta-reduction steps in an evaluation in three steps:

1. bound the number of transitions required in implementing evaluation strategies
2. estimate time cost of each transition
3. bound overall time cost of implementing evaluation strategies, by multiplying the number of transitions with time cost for each transition.

Given a pure term t, the time cost of an execution on the graph $t^\dagger$ is estimated by means of: (i) the number of reductions labelled with β in the evaluation of the term t, and (ii) the *size* $|t|$ of the term t, inductively defined as below.

$$|x| := 1, \qquad |\lambda x.t| := |t| + 1,$$
$$|t \mathbin{@} u| = |t \mathbin{\overrightarrow{@}} u| = |t \mathbin{\overleftarrow{@}} u| := |t| + |u| + 1, \qquad |t[x \leftarrow u]| := |t| + |u| + 1.$$

Given an evaluation *Eval*, the number of occurrences of a label χ is denoted by $|Eval|_\chi$. The sub-machine semantics comes with the following quantitative bounds.

Proposition 4.4. *For any evaluation* $Eval\colon (\!|t|\!) \to^* A\langle (\!|v|\!) \rangle$ *that terminates, the number of reductions are bounded as below.*

$$|Eval|_\sigma = \mathcal{O}(|Eval|_\beta), \qquad |Eval|_\varepsilon = \mathcal{O}(|t| \cdot |Eval|_\beta).$$

Proof outline. A term uses a single evaluation strategy, either call-by-need, left-to-right call-by-value, or right-to-left call-by-value. The proof is by developing the one-to-one correspondence between an evaluation by the sub-machine semantics and a "derivation" in the linear substitution calculus, in the same way Accattoli et al. analyse various abstract machines in [2]. The second equation is an analogy of [2, Thm. 11.3 & Thm. 11.5]. The first equation is a direct application of the bounds about the linear substitution calculus [5, Cor. 1 & Thm. 2]. □

We use the same notation $|Exec|_\chi$, as for an evaluation, to denote the number of occurrences of each label χ in an execution *Exec*. Additionally the number of rewrite transitions with the label ε is denoted by $|Exec|_{\varepsilon R}$. The following proposition completes the first step of the cost analysis.

Proposition 4.5 (Soundness & completeness, with number bounds)**.** *For any pure closed term t, an evaluation* $Eval\colon (\!|t|\!) \to^* A\langle (\!|v|\!) \rangle$ *terminates with* $A\langle (\!|v|\!) \rangle$ *if and only if an execution* $Exec\colon Init(t^\dagger) \to^* Final(A^\ddagger \circ v^\dagger)$ *terminates with the graph* $A^\ddagger \circ v^\dagger$*. Moreover the number of transitions are bounded as below.*

$$|Exec|_\beta = |Eval|_\beta, \quad |Exec|_\sigma = \mathcal{O}(|Eval|_\beta), \quad |Exec|_\varepsilon = \mathcal{O}(|t| \cdot |Eval|_\beta), \quad |Exec|_{\varepsilon R} = \mathcal{O}(|Eval|_\beta).$$

Proof. This proposition is a direct consequence of Thm. 4.3 and Prop. 4.4, except for the last bound. The last bound of $|Exec|_{\varepsilon R}$ follows the fact that each rewrite transition labelled with β is always preceded by one rewrite transition labelled with ε. □

The next step in the cost analysis is to estimate the time cost of each transition. We assume that graphs are implemented in the following way:

- Each link is given by two pointers to its child and its parent.
- Each node is given by its label and pointers to its outputs. Abstraction nodes (λ) and application nodes ($@$, $\overrightarrow{@}$ and $\overleftarrow{@}$) have two pointers that are distinguished, and all the other nodes have only one pointer to their unique output. Additionally each !-node has pointers to inputs of its associated ?-nodes, to represent a !-box structure.

Accordingly, a position of the token is a pointer to a link, a direction and a rewrite flag are two symbols, a computation stack is a stack of symbols, and finally a box stack is a stack of symbols and pointers to links. Using these assumptions of implementation, time cost of each transition is estimated as below.

- All pass transitions have constant cost. Each pass transition looks up one node and its outputs (that are either one or two) next to the current position. It consults and updates a fixed number of elements of the token data.
- Rewrite transitions with the label β have constant cost, as they change a constant number of nodes and links, and only a rewrite flag of the token data.
- Rewrite transitions with the label ε remove a !-box structure, and hence have cost bounded by the number of the auxiliary doors.
- Rewrite transitions with the label σ copy a !-box structure. Copying cost is bounded by the size of the !-box, i.e. the number of nodes and links in the !-box. Updating cost of the sub-graph H' (see Fig. 3) is bounded by the number of auxiliary doors, that is less than the size of the copied !-box. The assumption about the implementation of graphs enables us to conclude updating cost of the C-node is constant.

Finally we reach the last step of the cost analysis, and give the overall time cost of executions using the results of the previous two steps.

Theorem 4.6 (Soundness & completeness, with cost bounds)**.** *For any pure closed term t, an evaluation $Eval\colon (\!|t|\!) \to^* A\langle (\!|v|\!)\rangle$ terminates with $A\langle (\!|v|\!)\rangle$ if and only if an execution $Exec\colon Init(t^\dagger) \to^* Final(A^\ddagger \circ v^\dagger)$ terminates with the graph $A^\ddagger \circ v^\dagger$. The overall time cost of the execution $Exec$ is bounded by $\mathcal{O}(|t| \cdot |Eval|_\beta)$.*

Proof. Non-constant cost of rewrite transitions are either the number of auxiliary doors of a !-box or the size of a !-box. Because rewrite transitions can only copy or discard a !-box, and cannot expand or reduce a single !-box, any !-boxes involved in the execution $Exec$ has no more size than !-boxes included in the initial graph $t^\dagger$. The size of the initial graph $t^\dagger$ can be bounded by the size $|t|$ of the initial term. Therefore any non-constant cost of each rewrite transition, in the execution $Exec$, can be also bounded by $|t|$. The overall time cost of rewrite transitions labelled with β is $\mathcal{O}(|Eval|_\beta)$, and that of the other rewrite transitions and pass transitions is $\mathcal{O}(|t| \cdot |Eval|_\beta)$. □

Thm. 4.6 classifies the graph-rewriting machine as "efficient," by Accattoli's taxonomy [1, Def. 7.1] of analysing time efficiency of abstract machines. The efficiency benefits from the graphical representation of environments (i.e. explicit substitutions in our setting). In particular the translations $(\cdot)^\dagger$ and $(\cdot)^\ddagger$ are carefully designed to exclude any two sequentially-connected C-nodes, which yields the constant cost to look up a bound variable and its associated computation in environments.

5 Related work and conclusions

The idea of using the token as a guide of graph rewriting was also proposed by Sinot [18, 19] for interaction nets. He shows how using a token can make the rewriting system implement the call-by-name, call-by-need and call-by-value evaluation strategies. Our development in this work can be seen as a realisation of the rewriting system as an abstract machine, in particular with explicit control over copying sub-graphs.

The GoI-style token passing itself has been adapted to implement the call-by-value evaluation strategy. Fernández and Mackie allow the token to jump along a path in a graph, and yield a token-passing abstract machine that implements the call-by-value strategy in [8]. Their machine keeps the underlying graph fixed during execution, but jumps of the token enable the machine to recover time efficiency, although no quantitative analysis is provided. Jumps can be seen as a form of graph rewriting that eliminates nodes, and some jumps are to or from edges with an index that are effectively "virtual" copies of edges.

Another popular way to implement the call-by-value strategy is to use the CPS transformation [16], as adopted in [17] and [11], that focuses on correctness. However this method leads to an abstract machine with inefficient overhead cost, at least in the case of [11].

To wrap up, we presented a graph-rewriting abstract machine, with token passing as a guide, that can efficiently implement three evaluation strategies that have different control over caching intermediate results. The token-guided graph rewriting is a flexible framework in which we can carry out the study of space-time trade-off in abstract machines for various evaluation strategies of the lambda-calculus. Starting with [15] and continuing with the present work, our focus was primarily on time-efficiency, to complement existing work on GoI-style operational semantics which usually achieves space-efficiency. We believe that more refined strategies of interleaving token routing and graph reduction can be formulated to serve particular objectives in the space-time execution efficiency trade-off.

One remark on this flexible interleaving is that leaving redexes not rewritten will require additional token-passing rules and extra token data, compared with the current set of transitions and data, so that the token does not get stuck. For example, getting rid of the rewrite transition labelled with β will require an extra pass transition over an abstraction node, from its output to its input.

References

[1] Beniamino Accattoli (2017): *The complexity of abstract machines*. In: *WPTE 2016*, *EPTCS* 235, pp. 1–15.

[2] Beniamino Accattoli, Pablo Barenbaum & Damiano Mazza (2014): *Distilling abstract machines*. In: *ICFP 2014*, ACM, pp. 363–376.

[3] Beniamino Accattoli & Ugo Dal Lago (2016): *(Leftmost-outermost) beta reduction is invariant, indeed*. *Logical Methods in Comp. Sci.* 12(1).

[4] Beniamino Accattoli & Stefano Guerrini (2009): *Jumping boxes*. In: *CSL 2009*, *Lect. Notes Comp. Sci.* 5771, Springer, pp. 55–70.

[5] Beniamino Accattoli & Claudio Sacerdoti Coen (2014): *On the value of variables*. In: *WoLLIC 2014*, *Lect. Notes Comp. Sci.* 8652, Springer, pp. 36–50.

[6] Vincent Danos & Laurent Regnier (1996): *Reversible, irreversible and optimal lambda-machines*. *Elect. Notes in Theor. Comp. Sci.* 3, pp. 40–60.

[7] Olivier Danvy, Kevin Millikin, Johan Munk & Ian Zerny (2012): *On inter-deriving small-step and big-step semantics: a case study for storeless call-by-need evaluation*. *Theor. Comp. Sci.* 435, pp. 21–42.

[8] Maribel Fernández & Ian Mackie (2002): *Call-by-value lambda-graph rewriting without rewriting*. In: *ICGT 2002*, *LNCS* 2505, Springer, pp. 75–89.

[9] Jean-Yves Girard (1987): *Linear logic*. *Theor. Comp. Sci.* 50, pp. 1–102.

[10] Jean-Yves Girard (1989): *Geometry of Interaction I: interpretation of system F*. In: *Logic Colloquium 1988*, *Studies in Logic & Found. Math.* 127, Elsevier, pp. 221–260.

[11] Naohiko Hoshino, Koko Muroya & Ichiro Hasuo (2014): *Memoryful Geometry of Interaction: from coalgebraic components to algebraic effects*. In: *CSL-LICS 2014*, ACM, pp. 52:1–52:10.

[12] Aleks Kissinger (2012): *Pictures of processes: automated graph rewriting for monoidal categories and applications to quantum computing*. arXiv preprint arXiv:1203.0202.

[13] Ian Mackie (1995): *The Geometry of Interaction machine*. In: *POPL 1995*, ACM, pp. 198–208.

[14] John Maraist, Martin Odersky, David N. Turner & Philip Wadler (1999): *Call-by-name, call-by-value, call-by-need and the linear lambda calculus*. *Theor. Comp. Sci.* 228(1-2), pp. 175–210.

[15] Koko Muroya & Dan R. Ghica (2017): *The dynamic Geometry of Interaction machine: a call-by-need graph rewriter*. In: *CSL 2017*. To appear.

[16] Gordon Plotkin (1975): *Call-by-name, call-by-value and the lambda-calculus*. *Theor. Comp. Sci.* 1(2), pp. 125–259.

[17] Ulrich Schöpp (2014): *Call-by-value in a basic logic for interaction*. In: *APLAS 2014*, *Lect. Notes Comp. Sci.* 8858, Springer, pp. 428–448.

[18] François-Régis Sinot (2005): *Call-by-name and call-by-value as token-passing interaction nets*. In: *TLCA 2005*, *Lect. Notes Comp. Sci.* 3461, Springer, pp. 386–400.

[19] François-Régis Sinot (2006): *Call-by-need in token-passing nets*. *Math. Struct. in Comp. Sci.* 16(4), pp. 639–666.