(Not So) Boring Abstract Machines

Ugo Dal Lago[1] and Gabriele Vanoni[2]

[1] Università di Bologna, Italia & Inria, France
ugo.dallago@unibo.it
[2] Inria, France
vanonigabriele@gmail.com

Abstract. We study one of the two formulations of the interaction abstract machine, namely that obtained from the so-called call-by-value (or "boring") translation of intuitionistic logic into linear logic. We prove the correctness of the resulting *call-by-name* machine, at the same time establishing an improvement bisimulation with Krivine's abstract machine. The proof makes essential use of the definition of a novel relational property linking configurations of the two machines.

Keywords: λ-calculus · abstract machines · geometry of interaction.

1 Introduction

Abstract machines aim at closing the gap between the abstract, higher-order, rewriting theory of the λ-calculus, and low-level implementations. They are automata-like transition systems meant to simulate the evaluation of λ-terms and to be easily implemented on real architectures. Notable examples are Landin's SECD [25], and Krivine's KAM [24]. Different abstract machine can be designed to reduce terms following different reduction strategies, e.g. the SECD follows the call-by-value order, while the KAM proceeds in call-by-name. Moreover, abstract machines can achieve substantially different levels of efficiency.

Designing Abstract Machines. Given a λ-calculus and a reduction strategy, how can one devise an abstract machine that *implements* that evaluation? The first abstract machines, defined in the mid '60s, were designed by hand, relying only on the ingenuity of their creators. Only after having been defined, they were proven correct. After many years of research, there are now standard ways of deriving abstract machines, often directly from the operational semantics of the underlying calculus [31,6]. Moreover, the *Curry-Howard correspondence* provides another way of looking at the same problem: the normalization procedure of the underlying logic (e.g., cut-elimination) can be in turn seen as an abstract machine for the corresponding calculus. Linear logic [18] has been a breakthrough in this respect. Indeed, proof-nets, the proof theoretical syntax for linear logic, come with a cut-elimination procedure that resembles very much the one of an abstract machine.

Linear logic also provided another tool, dubbed *geometry of interaction* [19], which can be seen as a concrete but non-standard way of both interpreting proofs and implementing the cut-elimination process. This mechanism being effectively computable, it provides yet another way of computing the result of the evaluation of a λ-term.

The Interaction Abstract Machine. This latter idea has been developed in many directions in the '90s [27,15], and recently revisited and deeply scrutinized by Accattoli et al. [3]. The machine resulting from the geometry of interaction, called the Interaction Abstract Machine (IAM in the following) implements weak head reduction (thus proceeding essentially call-by-name) and has been historically presented on linear logic proof-nets, rather than on λ-terms. This is not a problem in itself, since λ-terms can be translated into proof-nets. In fact, there is more than one way to do so. The best known one is the (call-by-name) translation $(A \to B)^{\dagger} := !A^{\dagger} \multimap B^{\dagger}$ of intuitionistic logic inside linear logic. Another one, which is less common and dubbed *boring* by Girard in the original paper [18], is based on the translation $(A \to B)^{\ddagger} := !A^{\ddagger} \multimap !B^{\ddagger}$. This translation is also dubbed "call-by-value" in the literature (and in this paper), the reason being that it is adequate for call-by-value evaluation when considering surface proof-net reduction. But how about the IAM that one gets out of the boring translation? It turns out that the answer to this question is not boring at all: surprisingly, indeed, the IAM continues to implement weak head reduction! Indeed, if one wants to recover call-by-value adequacy, then one has to go beyond the pure geometry of interaction [16,10]. There is even more to that, however.

Adding Jumps. Danos and Regnier [15] considered an optimization technique for the IAM, called *jumping*. Moreover, they claimed that the machine obtained from the IAM adding jumps is isomorphic to the PAM (an abstract machine deeply connected with Hyland and Ong game semantics [22]) if one uses the call-by-name translation, and to the KAM if one uses instead the call-by-value one. The first claim was made formal by Accattoli et al. in [4]. The second one has never been made formal, and no fully fledged proof of it can be found in the literature. This is in fact what constitutes the bulk of this paper, in which we indeed prove that the boring IAM with jumps is (strongly bisimilar to) the KAM. As corollaries, we obtain a proof of the correctness of the boring IAM, and a proof of the fact that the number of steps of the KAM cannot be higher than that of the IAM.

Related Work. The literature on geometry of interaction and its applications is huge, and goes from, e.g., Girard's original papers [19], to Abramsky *et al*'s categorical reformulation of GoI [23,1] to Danos and Regnier's work on path algebras [14], through Ghica's applications to circuit synthesis [17], together with extensions by Hoshino, Muroya, and Hasuo to languages with various kinds of effects [21], and Laurent's extension to the additive connectives of linear logic [26]. The GoI has been studied in relationship with implementations of functional languages, by Gonthier, Abadi and Levy, who studied Lévy's

optimal evaluation [20], and by Mackie with his GoI machine for PCF [27]. The space-efficiency of GoI as discovered by Dal Lago and Schöpp [12] has later been exploited by Mazza in [28]. Dal Lago and coauthors have also introduced variants of the IAM acting on proof nets for a number of extensions of the λ-calculus [9,10,11,13]. Curien and Herbelin study abstract machines related to game semantics and the IAM in [7,8]. Moreover, Schöpp [29,30] has shown how GoI can be seen as an optimized form of CPS transformation, followed by defunctionalization. Finally, in a series of papers [3,4,5], Accattoli, Dal Lago and Vanoni have recently revisited the IAM, giving it a new dress, and obtaining general results about its time and space (in)efficiency.

2 λ-Calculus and Abstract Machines

Terms of the *λ-calculus* are defined as follows:

$$\lambda\text{-TERMS} \qquad t, u \ ::= \ x \in \mathcal{V} \ \big| \ \lambda x.t \ \big| \ tu$$

where $\mathcal{V}$ is a countable set of variables. *Free* and *bound variables* are defined as usual: $\lambda x.t$ binds x in t. A term is *closed* when there are no free occurrences of variables in it. Terms are considered modulo α-equivalence, and capture-avoiding (meta-level) substitution of all the free occurrences of x for u in t is noted $t\{x \leftarrow u\}$. Contexts are just λ-terms containing exactly one occurrence of a special symbol, the hole $\langle\cdot\rangle$, intuitively standing for a removed subterm. Here we adopt *levelled* contexts, whose index, i.e. the level, stands for the number of λ-abstractions (i.e. the number of !-boxes in linear logic terminology, if one translates the λ-calculus according to the call-by-value translation) the hole lies in.

LEVELLED CONTEXTS

$$C_0 \ ::= \ \langle\cdot\rangle \ \big| \ C_0 t \ \big| \ tC_0 \qquad\qquad C_{n+1} \ ::= \ \lambda x.C_n \ \big| \ tC_{n+1} \ \big| \ C_{n+1}t$$

We simply write C (or D) for a context whenever the level is not relevant. The operation consisting of replacing the hole $\langle\cdot\rangle$ with a term t in a context C is noted $C\langle t\rangle$ and called *plugging*. Please note that for every bound variable x, if this is isolated by its binding context $\lambda x.D_n$, then n is the number of λs occurring in the syntax tree of the term between the occurrence of x and its binder λx. In other words, n is the de Bruijn index of x. The operational semantics that we adopt here is weak head evaluation $\to_{wh}$, defined as follows:

$$(\lambda y.t)ur_1 \dots r_h \quad \to_{wh} \quad t\{y \leftarrow u\}r_1 \dots r_h.$$

We further restrict the setting by considering only closed terms, and refer to our framework as *Closed Call-by-Name* (shortened to Closed CbN). Basic well known facts are that in Closed CbN the normal forms are precisely the abstractions and that $\to_{wh}$ is deterministic.

Closures	Environments	Stacks	States
$c ::= (t, C, e)$	$e_0 ::= \epsilon \quad e_{n+1} ::= c{\cdot}e_n$	$\pi ::= \epsilon \mid c{\cdot}\pi$	$s ::= (t, C, e, \pi)$

Trm	Ctx	Env	Stack		Trm	Ctx	Env	Stack
tu	C	e	π	$\to_{\mathsf{sea}}$	t	$C\langle\langle\cdot\rangle u\rangle$	e	$(u, C\langle t\langle\cdot\rangle\rangle, e){\cdot}\pi$
$\lambda x.t$	C	e	$c{\cdot}\pi$	$\to_{\beta}$	t	$C\langle\lambda x.\langle\cdot\rangle\rangle$	$c{\cdot}e$	π
x	$C\langle\lambda x.D_n\rangle$	$e_n{\cdot}(t, C', e){\cdot}e'$	π	$\to_{\mathsf{sub}}$	t	C'	e	π

Fig. 1: Data structures and transitions of the Krivine Abstract Machine (KAM).

Abstract Machines Glossary. In this paper, an *abstract machine* M is a transition system $\to_{\mathrm{M}}$ over a set of states (we omit the subscript M when it is clear from the context). We use the standard notations $\to^*$ and $\to^n$ to denote the transitive and reflexive closure, and the composition n times of of $\to$, respectively. The machines considered in this paper move over the code, i.e. the initially fed λ-term, without ever changing it. A *position* in a term t is represented as a pair (u, C) of a sub-term u of t and a context C such that $C\langle u\rangle = t$. The shape of states depends on the specific machine, but they always include a position (u, C) plus some other data structures.

Initial states (on t) are always positioned at the root of the term $(t, \langle\cdot\rangle)$, where t is a closed term. Their precise shape depends on the actual machine. A state is *final* if no transitions apply. A *run* ρ is a possibly empty sequence of transitions. An *initial run* (from t) is a run from an initial state (on t). A state is *reachable* if it is the target state of an initial run. A *complete run* is an initial run ending on a final state. Given a machine M, we write $\mathrm{M}(t)\Downarrow$ if M reaches a final state starting from t, and $\mathrm{M}(t)\Uparrow$ otherwise. We say that M *implements Closed CbN* when $\mathrm{M}(t)\Downarrow$ if and only if $\to_{wh}$ terminates on t, for every closed term t.

The Krivine Abstract Machine. The Krivine abstract machine [24] (KAM) is a standard environment machine for Closed CbN. We present the KAM in a rather "mixed" way: while we stick with terms with variable names, we use de Bruijn indexes to retrieve closures stored inside the environment, as we now detail.

The KAM, defined in Fig. 1, records every β-redex that it encounters using two data structures, the *environment* e and the *stack* π. The basic idea is that, by saving encountered β-redexes in the environment e, the machine can simply look up in e for the argument of the binder λx whenever encountering a variable occurrence x. This is found looking at the entry in the environment corresponding to the de Bruijn index of x. The stack π is used to collect encountered arguments that still have to be paired to abstractions to form β-redexes, and then go into the environment e.

Closures, Stacks, and Environments. The mutually recursive grammars for *closures* and *environments*, plus the independent one for *stacks* are defined in

Logged Positions	Directions	Tapes
$l ::= \diamond \mid (\lambda x.C_n, L_{n+1})$	$d ::= \downarrow \mid \uparrow$	$T ::= \epsilon \mid \bullet \cdot T \mid \circ \cdot T \mid l \cdot T$

Logs	States
$L_0 ::= \epsilon \quad L_{n+1} ::= l \cdot L_n$	$q ::= (t, C, L, T, d)$

Term	Context	Log	Tape	
$\underline{tu}$	C	L	T	$\to_{\bullet 1}$
$\underline{t}$	$C\langle\langle\cdot\rangle u\rangle$	L	$\diamond \cdot \bullet \cdot T$	
$\underline{\lambda x.t}$	C	L	$l \cdot \bullet \cdot T$	$\to_{\bullet 2}$
$\underline{t}$	$C\langle\lambda x.\langle\cdot\rangle\rangle$	$l \cdot L$	T	
$\underline{x}$	$C\langle\lambda x.D_n\rangle$	$L_n \cdot l \cdot L$	$l' \cdot T$	$\to_{\mathsf{var}}$
$\lambda x.D_n\langle x\rangle$	$\underline{C}$	L	$l \cdot \circ \cdot (\lambda x.D_n, L_n \cdot l') \cdot T$	
$\underline{\lambda x.D_n\langle x\rangle}$	C	L	$l \cdot \circ \cdot (\lambda x.D_n, L_n \cdot l') \cdot T$	$\to_{\mathsf{bt2}}$
x	$\underline{C\langle\lambda x.D_n\rangle}$	$L_n \cdot l \cdot L$	$l' \cdot T$	

Fig. 2: Data structures and $\downarrow$-transitions of the BIAM.

Fig. 1, together with the definition of states. The idea is that every piece of code comes with an environment, forming a closure, which is why environments and closures are mutually defined. We denote by S_K the set of KAM states.

Transitions, Initial and Final States. Initial states of the KAM are in the form $s_t := (t, \langle\cdot\rangle, \epsilon, \epsilon)$. The transitions of the KAM are in Fig. 1—their union is noted $\to_{\text{KAM}}$. The idea is that the $\to_{\mathsf{sub}}$ transition looks in the environment for the argument of the variable under evaluation. The KAM evaluates the term t until the top abstraction of the weak head normal form of t is found, that is a run either never stops or ends in a state s of the shape $s = (\lambda x.u, C, e, \epsilon)$. This is guaranteed by the fact that terms are closed and the standard (but omitted) invariant ensuring that on reachable states the level of the context equals the length of the environment, so that the KAM never gets stuck on a $\to_{\mathsf{sub}}$ transition.

Theorem 1 ([24]). *The KAM implements Closed Call-by-Name.*

3 The Boring Interaction Abstract Machine

In this section we define the Boring Interaction Abstract Machine (BIAM). We adopt the λ-calculus presentation, as developed in [3]. As we have already said, the main difference is that here we translate λ-terms according to the call-by-value (or "boring") Girard's translation. This means that—with respect to the linear logic representation of a λ-term—*every* box encloses a λ-abstraction.

Bird's Eye view of the BIAM. Intuitively, the behavior of the BIAM can be seen as that of a token that travels around the syntax tree of the program under evaluation. Similarly to the KAM, it looks for the head variable of a term, but without storing the encountered β-redexes in an environment. When it finds the

Term	Context	Log	Tape		Term	Context	Log	Tape
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	L	$\diamond\cdot\bullet\cdot T$	$\to_{\bullet 3}$	tu	$\underline{C}$	L	T
t	$\underline{C\langle\lambda x.\langle\cdot\rangle\rangle}$	$l\cdot L$	T	$\to_{\bullet 4}$	$\lambda x.t$	$\underline{C}$	L	$l\cdot\bullet\cdot T$
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	L	$\diamond\cdot\circ\cdot T$	$\to_{\mathsf{arg}}$	$\underline{u}$	$C\langle t\langle\cdot\rangle\rangle$	L	T
u	$\underline{C\langle t\langle\cdot\rangle\rangle}$	L	T	$\to_{\mathsf{bt1}}$	$\underline{t}$	$C\langle\langle\cdot\rangle u\rangle$	L	$\diamond\cdot\circ\cdot T$

Fig. 3: $\uparrow$-transitions of the Boring Interaction Abstract Machine.

head variable, the BIAM looks for the argument which should replace it, because having no environment it cannot simply look it up. These two search mechanisms are realized by two different phases and directions of exploration of the code, noted $\downarrow$ and $\uparrow$. The functioning is actually more involved because there is also a backtracking mechanism, requiring to save and manipulate code positions in the token. Last, the machine never duplicates the code, but it distinguishes different uses of a same code (position) using *logs*. Unfortunately, there are no easy intuitions about how logs handle the different uses.

BIAM States. The BIAM travels on a λ-term t carrying some data structures, defined in Fig. 2, which store information about the computation and determine the next transition to apply. A key point is that navigation is done locally, moving only between adjacent positions.[3] The BIAM has also a *direction* of navigation that is either $\downarrow$ or $\uparrow$ (pronounced *down* and *up*). The token is given by two stacks, called *log* and *tape*, whose main components are *logged positions*. Roughly, a log is a trace of the relevant positions in the history of a computation. A logged position is either an occurrence of a distinguished symbol $\diamond$ (representing a dereliction in linear logic jargon), or a position plus a log, meant to trace the history that led to that position. Logs and logged positions are then defined by mutual induction, as it was the case for closures and environments in the KAM. We use $\cdot$ also to concatenate logs, writing, e.g., $L_n\cdot L$, using L for a log of unspecified length. The *tape* T is a list of logged positions plus occurrences of the special symbols $\bullet$ and $\circ$, needed to record the crossing of abstractions and applications. A *state* of the machine is given by a position, a log L and a tape T, together with a *direction*. Initial states have the form $q_t := (\underline{t}, \langle\cdot\rangle, \epsilon, \diamond)$.[4] Directions are often omitted and represented via colors and underlining: $\downarrow$ is represented by a red and underlined code term, $\uparrow$ by a blue and underlined code context. We denote by Q the set of BIAM states.

[3] Note that also the transition from the variable occurrence to the binder in $\to_{\mathsf{var}}$ and $\to_{\mathsf{bt2}}$ are local if λ-terms are represented by implementing occurrences as pointers to their binders, as in the proof net representation of λ-terms, upon which some concrete implementation schemes are based, see [2].

[4] Initial non-empty states could seem weird at first. However, they have a natural linear logical interpretation. In the call-by-value translation values are always !-boxed, and derelictions are needed to open such boxes. The symbol $\diamond$ indeed represents a dereliction, i.e. a query to open a box, thus needed also in the initial state.

Transitions. Intuitively, the machine evaluates the term t until the head abstraction of its head normal form is found. The transitions of the BIAM are split into two groups, $\downarrow$-transitions, which depend on the structure of the code term, in Fig. 2, and $\uparrow$-transitions, which depend on the structure of the code context, in Fig. 3. Their union is noted $\to_{\text{BIAM}}$. The idea is that $\downarrow$-states $(\underline{t}, C, L, T)$ are queries about the head variable of (the head normal form of) t and $\uparrow$-states $(t, \underline{C}, L, T)$ are queries about the argument of an abstraction.

Simple Properties. The BIAM verifies several nice properties which can be proved just by examining the transition rules. If we consider it as an automaton it is bi-deterministic, i.e. it is deterministic and reversible. Moreover, the tape satisfies a FIFO stack policy.

Proposition 1 (Reversibility and Pumping). *If $q_1 \to_{\text{BIAM}} q_2$, then:*
1. *$q_2^{\perp} \to_{\text{BIAM}} q_1^{\perp}$, where $q^{\perp}$ is q with inverted direction.*
2. *$q_1 + T \to_{\text{BIAM}} q_2 + T$, where if q has tape T, then $q + T'$ has tape $T \cdot T'$.*

If we restrict ourselves to reachable states, we are able to establish some properties about them, by induction on the initial run that leads to them.

Proposition 2 (Invariants). *If (t, C_n, L, T, d) is a reachable BIAM state, then:*

1. *T satisfies the regular expression $(l \cdot (\bullet \mid \circ))^* \cdot l$.*
2. *$|L| = n$.*

These two invariants together guarantee that the BIAM is never blocked on the transition $\to_{\mathsf{var}}$. A much stronger (and way more involved) invariant that we are going to present in the next section shall guarantee that the BIAM can never be blocked but on states of the form $(\underline{\lambda x.t}, C, L, l)$, which therefore is the only possible shape of *final* states.

4 Jumping is not Boring

This section contains the main technical result of this paper, namely that adding *jumps* (in the sense of Danos and Regnier [15]) to the BIAM turns the resulting machine into a device which is bisimilar to the KAM. While this result is only sketched in [15], here we give a formal proof that jumps are actually shortcuts over the BIAM path, and thus that the KAM can be simulated by the BIAM.

Jumps. We have already stressed that each transition of the BIAM is *local*. This means that at each step the token moves to positions which are adjacent to the previous ones in the syntax tree of the λ-term under evaluation. This is not the case for the KAM. Indeed, the transition $\to_{\mathsf{sub}}$ moves the focus of the evaluation to an arbitrary position of the term. We call these non-local transitions *jumps*. Of course, jumps are possible because during the already carried out computation some positions have been saved. In particular, one could see a similar pattern between the BIAM transition $\to_{\bullet 1}$ (or $\to_{\bullet 2}$) and the KAM transition $\to_{\mathsf{sea}}$ (or

Closures	Environments	Stacks	States
$c ::= (t, C, e)$	$e_0 ::= \epsilon \quad e_{n+1} ::= l{\cdot}e_n$	$\pi ::= \epsilon \mid \bullet{\cdot}\pi \mid l{\cdot}\pi$	$s ::= (t, C, e, \pi)$
Logged Contexts		Logged Closures	
$E ::= \langle\cdot\rangle \mid (\lambda x.C_n, e_n{\cdot}E)$		$l ::= E\langle c\rangle$	

Trm	Ctx	Env	Stack		T	Ctx	Env	Stack
tu	C	e	π	$\to_{\mathsf{sea}}$	t	$C\langle\langle\cdot\rangle u\rangle$	e	$(u, C\langle t\langle\cdot\rangle\rangle, e){\cdot}\bullet{\cdot}\pi$
$\lambda x.t$	C	e	$l{\cdot}\bullet{\cdot}\pi$	$\to_{\beta}$	t	$C\langle\lambda x.\langle\cdot\rangle\rangle$	$l{\cdot}e$	π
x	$C\langle\lambda x.D_n\rangle$	$e_n{\cdot}E\langle t, C', e'\rangle{\cdot}e$	$l{\cdot}\pi$	$\to_{\mathsf{sub}}$	t	C'	e'	$(\lambda x.D_n, e_n{\cdot}l){\cdot}\pi$

Fig. 4: Data structures and transitions of the Boring Jumping Abstract Machine.

$\to_\beta$). Ignoring the constant $\bullet$ (which is not needed in the KAM, although it could be straightforwardly added), their difference lies in the fact that in transition $\to_{\mathsf{sea}}$ the dereliction constant $\diamond$ (and in transition $\to_\beta$ a generic logged position l) have been substituted by a closure c, that saves the encountered arguments (together with their respective environments). These closures are then used in the jump of transition $\to_{\mathsf{sub}}$.

The Boring Jumping Abstract Machine. We formalize the intuition above by defining the Boring Jumping Abstract Machine (BJAM), in Fig. 4. One could already notice the (bi-)similarity with the KAM, this is why we have kept the same transition names. Consistently with the BIAM, initial BJAM states have the form $(t, \langle\cdot\rangle, \epsilon, c_0)$, where c_0 is a dummy closure, and final states have the shape $(\lambda x.t, C, e, l)$. We denote by S_J the set of BJAM states. This machine is obtained from the BIAM applying just two small adjustments. (i) Saving a closure in transition $\to_{\bullet 1}$ instead of putting simply a $\diamond$; and (ii) jumping in transition $\to_{\mathsf{var}}$ to the argument stored in the *right* closure (as in the KAM). With these two modifications, one immediately notices that $\uparrow$-transitions are no more needed. Indeed, since the initial states are of $\downarrow$-direction and without any $\circ$, $\uparrow$-states are no more reachable (and the $\to_{\mathsf{bt2}}$ transition is never fired). Please notice that closures here, and differently from the KAM, can be surrounded by arbitrary *logged contexts*. Indeed closures can be pushed deep inside at arbitrary depth by the *BJAM* transition $\to_{\mathsf{sub}}$.

Organizing the Zoo. At this point we have defined three different machines: the KAM, the BIAM, and the BJAM. Then, we want to understand the relationship between them, and in particular to compare their efficiency. In particular, we want to prove that (i) the BJAM saves some steps over the BIAM; and (ii) the BJAM is strongly bisimilar to the KAM. But first, we briefly illustrate how the three machines compute with a small example.

Example 1. We consider the term $t := (\lambda x.x)(\lambda y.y)$. The BJAM, the BIAM, and the KAM complete runs are as follows (where $C := (\lambda x.x)\langle\cdot\rangle$ and $I := \lambda y.y$):

$$(\underline{t}, \langle\cdot\rangle, \epsilon, c_0) \to^2_{\text{BJAM}} (\underline{x}, (\lambda x.\langle\cdot\rangle)(\lambda y.y), (I, C, \epsilon), c_0) \to_{\text{BJAM}} (\underline{I}, C, \epsilon, (\lambda x.\langle\cdot\rangle, c_0))$$
$$(\underline{t}, \langle\cdot\rangle, \epsilon, \diamond) \to^2_{\text{BIAM}} (\underline{x}, (\lambda x.\langle\cdot\rangle)(\lambda y.y), \diamond, \diamond) \to^2_{\text{BIAM}} (\underline{I}, C, \epsilon, (\lambda x.\langle\cdot\rangle, \diamond))$$
$$(t, \langle\cdot\rangle, \epsilon, \epsilon) \to^2_{\text{KAM}} (x, (\lambda x.\langle\cdot\rangle)(\lambda y.y), (I, C, \epsilon), \epsilon) \to_{\text{KAM}} (I, C, \epsilon, \epsilon)$$

The reader can immediately notice that the BJAM somehow subsumes both the BIAM and the KAM. From BJAM states one obtains BIAM states by turning all closures into $\diamond$, and, in a dual way, one obtains KAM states by removing all the logged contexts surrounding closures.

Improvements, Abstractly. We formalize the intuition above through a bisimulation-like argument. In particular, we need to prove that BJAM jumps can be simulated by (possibly many) BIAM transitions. We use the abstract framework of improvements already setup in [3].

Definition 1 (Improvements). *Given two abstract machines with set of states Q and S, a relation $\mathcal{R} \subseteq Q \times S$ is an* improvement *if given $(q, s) \in \mathcal{R}$ the following conditions hold:*

1. Final state right*: if $s \in S$ is a final state, then $q \to^n q'$, for some final state $q' \in Q$ and $n \geq 0$.*
2. Transition left*: if $q \in Q$ and $q \to q'$, then there exists $q'' \in Q$ and $s' \in S$ such that $q' \to^m q''$, $s \to^n s'$, $q''\mathcal{R}s'$ and $n \leq m + 1$.*
3. Transition right*: if $s \in S$ and $s \to s'$, then there exists $q' \in Q$ and $s'' \in S$ such that $q \to^m q'$, $s' \to^n s''$, $q'\mathcal{R}s''$ and $m \geq n + 1$.*

What improves along an improvement is the number of transitions required to reach a final state, if any.

Proposition 3 ([3]). *Let $\mathcal{R} \subseteq Q \times S$ be an improvement, and $q\mathcal{R}s$. Then:*

1. (Non) Termination: *$q \to^* q'$ with q' final if and only if $s \to^* s'$ with s' final.*
2. Improvement: *$|q| \geq |s|$, where $|a|$ is the length of the run from a to a final state, if any, and $+\infty$ otherwise.*

Relating the BIAM and the BJAM. The main difficulty in relating BIAM and BJAM runs lies in proving that BJAM jumps can be simulated by the BIAM. To do so, we first define a translation $(\cdot)^I : S_J \to Q$, from BJAM states to BIAM states (and extended to all of their components) as follows:

$$(t, C, e, \pi)^I := (\underline{t}, C, e^I, \pi^I) \qquad (l \cdot e)^I := l^I \cdot e^I \qquad (\epsilon)^I := \epsilon$$
$$(l \cdot \pi)^I := l^I \cdot \pi^I \qquad (\bullet \cdot \pi)^I := \bullet \cdot \pi^I \qquad \langle\cdot\rangle^I := \langle\cdot\rangle$$
$$(\lambda x.C_n, e_n \cdot E)^I := (\lambda x.C_n, (e_n)^I \cdot E^I) \qquad E\langle c\rangle^I := E^I\langle\diamond\rangle$$

Then, we define the relation $\succ \subseteq Q \times S_J$ that we want to prove being an improvement as:

$$q \succ s \text{ if and only if } q \text{ and } s \text{ are reachable and } q = s^I$$

It is immediate to show that initial states (on the same term) are related: indeed we have that $(t, \langle\cdot\rangle, \epsilon, (t, \langle\cdot\rangle, \epsilon))^I = (\underline{t}, \langle\cdot\rangle, \epsilon, \diamond)$. At this point one has to prove that the relation $\succ$ satisfies Definition 1. One would like to proceed examining all the transitions of the BJAM, and while transitions $\rightarrow_{\mathsf{sea}}$, and $\rightarrow_{\beta}$ are easy to reason about, there is no easy way to handle transition $\rightarrow_{\mathsf{sub}}$. Indeed, while the BJAM performs a jump, the BIAM performs a (possibly very long) series of transitions. To solve this issue, we resort to proving a very strong invariant that somehow relates BJAM and BIAM states.

The Exhaustible Invariant. We lift the technique introduced by Accattoli et al. in [3], dubbed exhaustibility, to our use case. The technique being sophisticated some preliminary technical definitions are needed. The idea is that every closure c in a reachable BJAM state s can be tested. The test is passed if the BIAM state corresponding to c^I can be reached from s^I. Then, in order for the induction to work the exhaustibility predicate has to be strengthened, in a logical relation like way, requiring also the target state to be exhaustible. We first define test states for the stack and the environment.

Definition 2 (Stack tests). *Let $s = (t, C, e, \pi)$ be an BJAM state. Then, the BIAM state $q_c := (t, \underline{C}, e^I, (\pi')^I \cdot E^I\langle\diamond\rangle \cdot \circ)$ is a stack test for s of focus c for each decomposition $\pi = \pi' \cdot E\langle c\rangle \cdot \pi'' \cdot l'$ of the stack.*

Definition 3 (Environment tests). *Let $s = (t, C\langle\lambda x.D_n\rangle, e_n{\cdot}E\langle c\rangle{\cdot}e, \pi)$ be an BJAM state. Then, the BIAM state $q_c := (\lambda x.D_n\langle t\rangle, \underline{C}, e^I, E^I\langle\diamond\rangle \cdot \circ)$ is an environment test for s of focus c.*

Each closure naturally defines a BJAM state.

Definition 4 (State induced by a closure). *Given a closure $c := (t, C, e)$, the BJAM state s_c induced by c is defined as $s_c := (t, C, e, \epsilon)$.*

Then we are able to give a formal definition of exhaustible states, based on the previously defined notions.

Definition 5 (Exhaustible states). *$\mathcal{E}$ is the smallest set of those BJAM states s such that for any stack or environment test q_c of s of focus c, there exists a BIAM run $\rho : q_c \rightarrow^+_{\lambda\mathrm{IAM}} (s_c)^I$ such that $s_c \in \mathcal{E}$.*

Now, by induction on the length of the run we can prove (in the Appendix) that all reachable BJAM states are actually exhaustible.

Lemma 1. *Let s be a reachable BJAM state. Then s is exhaustible.*

Example 2. Being the invariant and the related definitions hard to grasp, we provide an example of application. We consider the reachable BJAM state $(\underline{x}, (\lambda x.\langle\cdot\rangle)(\lambda y.y), (\lambda y.y, (\lambda x.x)\langle\cdot\rangle, \epsilon), c_0)$, seen in Example 1. It has one environment test, namely $(\lambda x.x, \underline{\langle\cdot\rangle(\lambda y.y)}, \epsilon, \diamond \cdot \circ)$, of focus $c := (\lambda y.y, (\lambda x.x)\langle\cdot\rangle, \epsilon)$. The BJAM state induced by c is $s_c := (\lambda y.y, (\lambda x.x)\langle\cdot\rangle, \epsilon, \epsilon)$, and indeed we have that:

$$(\lambda x.x, \underline{\langle\cdot\rangle(\lambda y.y)}, \epsilon, \diamond \cdot \circ) \rightarrow_{\mathsf{arg}} (\underline{\lambda y.y}, (\lambda x.x)\langle\cdot\rangle, \epsilon, \epsilon) = (s_c)^I$$

We use Lemma 1 to prove the following key proposition, that establishes the simulation of the BJAM by the BIAM.

Proposition 4. $\succ$ *is an improvement.*

Proof. We need to verify all points of Definition 1. Point 1 (final state right) is immediate. Indeed if $q \succ s$ and s is final, i.e. in the form $s := (\lambda x.t, C, e, l)$, then also $q := (\underline{\lambda x.t}, C, e^I, l^I)$ is a final state.

We prove point 2 and 3 together. We proceed by analyzing the shape of s. Cases $\to_{\mathsf{sea}}$ and $\to_\beta$ are in the Appendix, while here we report the interesting one, i.e. the case $\to_{\mathsf{sub}}$. We need to close the following diagram:

$$\begin{array}{ccc} (\underline{x}, C\langle\lambda x.D_n\rangle, (e_n)^I \cdot E^I\langle\diamond\rangle \cdot e^I, l^I \cdot \pi^I) & & \succ (x, C\langle\lambda x.D_n\rangle, e_n \cdot E\langle t, C', e'\rangle \cdot e, l \cdot \pi) \\ \downarrow^{\mathsf{var}} & & \downarrow^{\mathsf{sub}} \\ (\lambda x.D_n\langle x\rangle, \underline{C}, e^I, E^I\langle\diamond\rangle \cdot \circ \cdot (\lambda x.D_n, (e_n)^I \cdot l^I) \cdot \pi^I) & & (t, C', e', (\lambda x.D_n, e_n \cdot l) \cdot \pi) \end{array}$$

Since s is reachable, and hence exhaustible by Lemma 1, we apply the definition of exhaustible state to the environment closure (t, C', e'). This means that we have the sequence of reductions:

$$q_c := (\lambda x.D_n\langle t\rangle, \underline{C}, e^I, E^I\langle\diamond\rangle \cdot \circ) \to^+_{\text{BIAM}} (t, C', (e')^I, \epsilon)$$

which, pumping $(\lambda x.D_n, (e_n)^I \cdot l^I) \cdot \pi^I$, allows us to close the diagram. □

Relating the BJAM and the KAM. We have related the BIAM and the BJAM. What is left is then to relate the BJAM and the KAM. Intuitively, BJAM states are just KAM states decorated with logged contexts. Moreover, the two machines morally behave the same. Indeed, logged contexts are never used by the BJAM, because in transition $\to_{\mathsf{sub}}$ they are just thrown away from the environment. This is why the proof of strong bisimulation is easy. As before, we first define a translation $(\cdot)^K : S_J \to S_K$, from BJAM states to KAM states (and extended to all of their components) that just ignores logged contexts, the rightmost dummy closure in the stack, and the $\bullet$ symbol, as follows:

$$\begin{array}{ccc} (t, C, e, \pi \cdot l)^K := (t, C, e^K, \pi^K) & (l \cdot e)^K := l^K \cdot e^K & (\epsilon)^K := \epsilon \\ (l \cdot \pi)^K := l^K \cdot \pi^K & (\bullet \cdot \pi)^K := \pi^K & E\langle c\rangle^K := c \end{array}$$

Then, we define the relation $\simeq \subseteq S_J \times S_K$ that we want to prove being an improvement as:

$$s \simeq r \text{ if and only if } s \text{ and } r \text{ are reachable and } r = s^K$$

We immediately observe that $\simeq$ is an improvement (actually a strong bisimulation).

Proposition 5. *The BJAM and the KAM are strongly bisimilar.*

Then, we can state our main result, namely that the BIAM can simulate the KAM.

Theorem 2. *There is an improvement between the BIAM and the KAM.*

Proof. Since the composition of two improvements is again an improvement, the desired improvement is $\succ; \simeq$.

Corollary 1. *The BIAM implements Closed Call-by-Name.*

Proof. Since improvements preserve (non)termination (Lemma 2).

5 Further Remarks

We have analyzed the relationship between the BIAM, the BJAM, and the KAM. We use this section to clarify some points that we have (intentionally) overlooked, and that we leave for further work.

Relationship with the IAM. The reader may wonder what is the relationship between the BIAM and the IAM obtained through the call-by-name translation of intuitionistic logic inside linear logic. A simple inspection of the transitions rules of the BIAM reveals that the two machines are likely to be strongly bisimilar. However, it is not clear whether and how it is possible to prove the correspondence directly, i.e. by exhibiting a concrete strong bisimulation between the states of the two machines, which are indeed very different. The length of the log is in fact equal to the depth of the code context, which is defined in a different way in the two translations. This question has to do with the intimate nature of the two translations, and is thus intriguing.

Call-by-Value Adequacy and Intersection Types. In [4], Accattoli and coauthors have established a tight correspondence (a sort of isomorphism) between the IAM runs on term t, and the (non-idempotent) intersection type derivation for t itself. It seems then natural to extend that correspondence to the "boring" case. Unfortunately, things are not straightforward. Indeed, the non-idempotent intersection type system obtained from the call-by-value translation is adequate for *call-by-value* evaluation (as one would expect), while here we have proved that the BIAM is adequate for *call-by-name*. We think that types could help in better understanding this mismatch, which definitely deserves to be better studied.

6 Conclusions

With this paper we have clarified some obscure points in the relationship between Krivine's abstract machine, the call-by-value (or "boring") Girard's translation of intuitionistic logic inside linear logic, and Girard's geometry of interaction. The connection between these three concepts was already considered by Danos and Regnier in [15], but without detailed proofs, and relying on the graphical syntax of linear logic proof-nets. Our exposition instead is fully formal and does not require any prerequisite but the familiarity with the λ-calculus.

References

1. Abramsky, S., Haghverdi, E., Scott, P.: Geometry of Interaction and linear combinatory algebras. Mathematical Structures in Computer Science **12**(5), 625–665 (2002)
2. Accattoli, B., Barras, B.: Environments and the complexity of abstract machines. In: Vanhoof, W., Pientka, B. (eds.) Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017. pp. 4–16. ACM (2017). https://doi.org/10.1145/3131851.3131855
3. Accattoli, B., Dal Lago, U., Vanoni, G.: The machinery of interaction. In: Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming. PPDP '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3414080.3414108
4. Accattoli, B., Dal Lago, U., Vanoni, G.: The (in)efficiency of interaction. Proc. ACM Program. Lang. **5**(POPL) (Jan 2021). https://doi.org/10.1145/3434332, https://doi.org/10.1145/3434332
5. Accattoli, B., Dal Lago, U., Vanoni, G.: The space of interaction. In: 2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). pp. 1–13 (2021). https://doi.org/10.1109/LICS52264.2021.9470726
6. Ager, M.S., Biernacki, D., Danvy, O., Midtgaard, J.: A functional correspondence between evaluators and abstract machines. In: Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden. pp. 8–19. ACM (2003). https://doi.org/10.1145/888251.888254
7. Curien, P., Herbelin, H.: Computing with abstract böhm trees. In: Sato, M., Toyama, Y. (eds.) Third Fuji International Symposium on Functional and Logic Programming, FLOPS 1998, Kyoto, Japan, Apil 2-4, 1998. pp. 20–39. World Scientific, Singapore (1998)
8. Curien, P., Herbelin, H.: Abstract machines for dialogue games (2007), http://arxiv.org/abs/0706.2544
9. Dal Lago, U., Faggian, C., Hasuo, I., Yoshimizu, A.: The geometry of synchronization. In: Henzinger, T.A., Miller, D. (eds.) Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014. pp. 35:1–35:10. ACM (2014). https://doi.org/10.1145/2603088.2603154
10. Dal Lago, U., Faggian, C., Valiron, B., Yoshimizu, A.: Parallelism and synchronization in an infinitary context. In: 30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015. pp. 559–572. IEEE Computer Society (2015). https://doi.org/10.1109/LICS.2015.58
11. Dal Lago, U., Faggian, C., Valiron, B., Yoshimizu, A.: The geometry of parallelism: classical, probabilistic, and quantum effects. In: Proceedings of the 44th POPL. pp. 833–845 (2017)
12. Dal Lago, U., Schöpp, U.: Computation by interaction for space-bounded functional programming. Information and Computation **248**, 150–194 (2016). https://doi.org/10.1016/j.ic.2015.04.006
13. Dal Lago, U., Tanaka, R., Yoshimizu, A.: The geometry of concurrent interaction: Handling multiple ports by way of multiple tokens. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017. pp. 1–12. IEEE Computer Society (2017). https://doi.org/10.1109/LICS.2017.8005112

14. Danos, V., Regnier, L.: Local and asynchronous beta-reduction (an analysis of Girard's execution formula). In: Proceedings of the Eighth Annual Symposium on Logic in Computer Science (LICS '93), Montreal, Canada, June 19-23, 1993. pp. 296–306. IEEE Computer Society (1993). https://doi.org/10.1109/LICS.1993.287578
15. Danos, V., Regnier, L.: Reversible, irreversible and optimal lambda-machines. Theoretical Computer Science **227**(1), 79–97 (1999). https://doi.org/10.1016/S0304-3975(99)00049-3
16. Fernández, M., Mackie, I.: Call-by-value lambda-graph rewriting without rewriting. In: Corradini, A., Ehrig, H., Kreowski, H., Rozenberg, G. (eds.) Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings. Lecture Notes in Computer Science, vol. 2505, pp. 75–89. Springer (2002). https://doi.org/10.1007/3-540-45832-8_8
17. Ghica, D.R.: Geometry of Synthesis: A Structured Approach to VLSI Design. In: Hofmann, M., Felleisen, M. (eds.) Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007. pp. 363–375. ACM (2007). https://doi.org/10.1145/1190216.1190269
18. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–101 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
19. Girard, J.Y.: Geometry of Interaction 1: Interpretation of System F. In: Ferro, R., Bonotto, C., Valentini, S., Zanardo, A. (eds.) Studies in Logic and the Foundations of Mathematics, vol. 127, pp. 221–260. Elsevier (1989)
20. Gonthier, G., Abadi, M., Lévy, J.J.: The Geometry of Optimal Lambda Reduction. In: Proceedings of the 19th POPL. pp. 15–26 (1992)
21. Hoshino, N., Muroya, K., Hasuo, I.: Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In: Henzinger, T.A., Miller, D. (eds.) Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014. pp. 52:1–52:10. ACM (2014). https://doi.org/10.1145/2603088.2603124
22. Hyland, J.M.E., Ong, C.L.: On full abstraction for PCF: i, ii, and III. Inf. Comput. **163**(2), 285–408 (2000). https://doi.org/10.1006/inco.2000.2917, https://doi.org/10.1006/inco.2000.2917
23. Joyal, A., Street, R., Verity, D.: Traced monoidal categories. Mathematical Proceedings of the Cambridge Philosophical Society **119**(3), 447–468 (1996)
24. Krivine, J.L.: A Call-by-name Lambda-calculus Machine. Higher Order Symbol. Comput. **20**(3), 199–207 (2007). https://doi.org/10.1007/s10990-007-9018-9
25. Landin, P.J.: The mechanical evaluation of expressions **6**(4), 308–320
26. Laurent, O.: A token machine for full geometry of interaction. In: Abramsky, S. (ed.) Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2044, pp. 283–297. Springer (2001). https://doi.org/10.1007/3-540-45413-6_23
27. Mackie, I.: The Geometry of Interaction Machine. In: Cytron, R.K., Lee, P. (eds.) Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995. pp. 198–208. ACM Press (1995). https://doi.org/10.1145/199448.199483

28. Mazza, D.: Simple parsimonious types and logarithmic space. In: Kreutzer, S. (ed.) 24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany. LIPIcs, vol. 41, pp. 24–40. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015). https://doi.org/10.4230/LIPIcs.CSL.2015.24
29. Schöpp, U.: On the relation of interaction semantics to continuations and defunctionalization. Logical Methods in Computer Science **10**(4) (2014). https://doi.org/10.2168/LMCS-10(4:10)2014
30. Schöpp, U.: From call-by-value to interaction by typed closure conversion. In: Feng, X., Park, S. (eds.) Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9458, pp. 251–270. Springer (2015). https://doi.org/10.1007/978-3-319-26529-2_14
31. Sestoft, P.: Deriving a lazy abstract machine. J. Funct. Program. **7**(3), 231–264 (1997)