The (In)Efficiency of Interaction

BENIAMINO ACCATTOLI, Inria & LIX, École Polytechnique, UMR 7161, France
UGO DAL LAGO, Università di Bologna, Italy and Inria Sophia Antipolis, France
GABRIELE VANONI, Università di Bologna, Italy and Inria Sophia Antipolis, France

Evaluating higher-order functional programs through abstract machines inspired by the geometry of the interaction is known to induce *space* efficiencies, the price being *time* performances often poorer than those obtainable with traditional, environment-based, abstract machines. Although families of lambda-terms for which the former is exponentially less efficient than the latter do exist, it is currently unknown how *general* this phenomenon is, and how far the inefficiencies can go, in the worst case. We answer these questions formulating four different well-known abstract machines inside a common definitional framework, this way being able to give sharp results about the relative time efficiencies. We also prove that non-idempotent intersection type theories are able to precisely reflect the time performances of the interactive abstract machine, this way showing that its time-inefficiency ultimately descends from the presence of higher-order types.

CCS Concepts: • **Theory of computation** → **Lambda calculus**; **Abstract machines**.

Additional Key Words and Phrases: lambda-calculus, abstract machines, geometry of interaction

ACM Reference Format:
Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2021. The (In)Efficiency of Interaction. *Proc. ACM Program. Lang.* 5, POPL, Article 51 (January 2021), 33 pages. https://doi.org/10.1145/3434332

1 INTRODUCTION

Sometimes, simple objects such as natural numbers can generate theories of marvelous richness, such as number theory. Something similar happens with the λ-calculus, the universally accepted model of purely functional programs. Its definition is simple: three constructors and just one rewriting rule, β-reduction, based on a natural notion of substitution. The theory of β-reduction, however, is surprisingly rich, and still the object of research, despite decades of deep investigations.

In the eighties, Barendregt's book [Barendregt 1984] presented a stable operational and denotational theory, Lévy had already developed his sophisticated optimality theory [Lévy 1978], and languages such as Haskell were using tricky sharing mechanisms in their implementations. In 1987, however, the linear logic [Girard 1987] earthquake came together with a completely new viewpoint on the λ-calculus, requiring to revisit the whole theory. For our story, two of its byproducts are relevant, namely the geometry of interaction [Girard 1989] (shortened to GoI) and game semantics.

GoI, Game Semantics and Abstract Machines. At the time, GoI was a radically new interpretation of proofs, arising from connections between linear logic and functional analysis, and based on an abstract notion of interactive execution for proofs. Game semantics was introduced to solve the *full abstraction problem for PCF* [Milner 1977], and along the years affirmed itself as the sharpest

Authors' addresses: Beniamino Accattoli, LIX, Inria & LIX, École Polytechnique, UMR 7161, France, beniamino.accattoli@inria.fr; Ugo Dal Lago, Università di Bologna, Italy, Inria Sophia Antipolis, France, ugo.dallago@unibo.it; Gabriele Vanoni, Università di Bologna, Italy, Inria Sophia Antipolis, France, gabriele.vanoni2@unibo.it.

2475-1421/2021/1-ART51
https://doi.org/10.1145/3434332

and most flexible form of semantics for higher-order languages. Roughly, the models known at the time were not able to capture fine computational behaviors—*i.e.* they were not *intensional* enough. Strategies from game semantics, instead, allow to faithfully model these behaviors of λ-terms: program composition is modeled as the interplay between the corresponding strategies—a concrete form of interaction—having the flavor of executions in some sort of abstract machine. In fact, there are two styles of game semantics. One, AJM games, is due to Abramsky, Jagadeesan, and Malacaria [Abramsky et al. 2000], and it is directly inspired by GoI. Another one, HO games, due to Hyland and Ong [Hyland and Ong 2000], models interaction in a different, pointer-based, way.

The computational content of GoI was first explored by Danos and Regnier [1995] and Mackie [1995], who proposed a new form of implementation schema called *interaction abstract machine* (shortened to IAM). The IAM works in a fundamentally different way with respect to environment-based abstract machines, which are the standard and time-efficient way of modeling the implementations of functional languages. The link between game semantics and abstract machines was first explored by Danos et al. [1996]. They showed the IAM to be the machinery behind AJM games, and the new *pointer abstract machine* (PAM) the one behind HO games. They also established a correspondence between the two styles of games, providing an indirect relationship between the IAM and the PAM. Finally, from a technical study of the IAM, Danos and Regnier [1999] introduced an optimized machine, the *jumping abstract machine* (JAM), claiming it isomorphic to the PAM despite using different data structures. In the following, we refer collectively to the IAM, the JAM, and the PAM, as to *game machines* (*interaction machines* would be ambiguous, because of the IAM).

A Blind Spot. Despite the existence of a huge literature about GoI and game semantics, their related abstract machines remain—somewhat surprisingly—not well understood. Game machines are quite sophisticated and their presentations are hard to grasp, sometimes even far from being formally defined. For instance, the PAM has always been presented informally, as an algorithm described in natural language or pseudocode. Additionally, the relationship between these machines is not clear, especially at the level of the relative performances. One of the aims of this paper is taking the first steps towards a proper theory of the efficiency of game machines.

Space and Interactions. It is well known that environment machines can be space inefficient, because the environment (or closure) mechanism they rely on uses space proportional to the number of β-steps, *i.e.* the natural time cost model of the λ-calculus. Using as much space as time is in fact the worst one can do, from a space efficiency point of view. The IAM relies on a different mechanism, that—similarly to offline Turing machines [Dal Lago and Schöpp 2010]—sacrifices time in order to be space-efficient. This phenomenon was first pointed out by Mackie [1995], but it is the extensive work by Schöpp and coauthors [Dal Lago and Schöpp 2010; Schopp 2007; Schöpp 2014, 2015] that showed that the IAM allows for capturing sub-linear space computations[1], something impossible in environment machines. Along the same lines, one can mention the *Geometry of Synthesis* [Ghica 2007; Ghica and Smith 2010], in which the geometry of interaction is seen as a compilation scheme towards circuits, and computation space is *finite*, and of paramount importance.

Time and Interactions. About time, instead, not much is known for game machines. Since the early papers on the IAM [Danos and Regnier 1995; Mackie 1995], it is known that it can be exponentially slower than environment machines. As an example, the family of terms t_n defined as $t_1 := \mathsf{I}$ and $t_{n+1} := t_n\mathsf{I}$ (where I is the identity combinator) takes time exponential in n to be evaluated by

[1]Evaluating a λ-term without fully inspecting it is indeed possible if the term is accessed by way of pointers, in the spirit of *offline Turing machines*—themselves an essential ingredient in the definition of complexity classes such as LOGSPACE—and this is precisely the way the IAM works. See [Dal Lago and Schöpp 2016] for a thorough discussion about sub-linear space computations in the λ-calculus.

the IAM, but only linear time in any environment machine. Therefore, game and environment machines are fundamentally different devices, and game ones—at least the IAM—*can* be time-inefficient. There remain however various open questions. Is the inefficiency of the IAM a *general* phenomenon, that is, are *all* λ-terms concerned? What about the other game machines? How bad can the aforementioned phenomenon be, quantitatively speaking? On *which* λ-terms does the phenomenon show up? The main objective of this paper is to provide answers to these questions.

Jumping is Dizzying. The time inefficiency of the IAM is addressed by Danos and Regnier and Mackie via an optimized machine, the JAM [Danos and Regnier 1999; Mackie 1995]. In which relation the JAM is with other machines is unclear. Danos and Regnier present the JAM as an optimization of the IAM defined on top of proof nets. Then, they claim (without proving it) that if one considers the call-by-name translation of the λ-calculus into proof nets, the JAM is isomorphic to the PAM, while they prove that using the call-by-*value* translation one obtains the KAM. This is somehow puzzling, since the KAM is a call-by-*name* machine.

Time, Environments, and Types. Another natural question comes from the study of the relationship between intersection types and environment machines. The non-idempotent variant of intersection types—here shortened to *multi types*—provides a type theoretic understanding of time for environment machines, as shown by de Carvalho [2018], since the execution time of environment machines can be extracted from multi types derivations. It is natural to wonder whether similar connections exist between game machines and multi types, or some other form of type system. That would be particularly useful as a way of comparing the time behaviour of a given term when evaluated by distinct machines.

This Paper. The aim of this work is giving the first sharp results about the time (in)efficiency of the interaction mechanism at work in game machines. We adopt the simplest possible setting, *i.e.* weak call-by-name evaluation on closed terms, and we provide four main contributions.

Contributions. (1) *A Formal Common Framework* Inspired by a very recent reformulation of the IAM on λ-terms (rather than proof nets, as in the original papers) by Accattoli et al. [2020b], called λIAM, we provide new similar presentations of the JAM and the PAM, called λJAM and λPAM. These formulations are easily manageable and comparable, enabling neat formal results about them—in particular, ours is the first formal and manageable definition of the PAM.

(2) *Comparative Complexity.* We provide bisimulations between the λIAM, the λJAM, the λPAM, and additionally the KAM, taken as the reference for environment machines. This allows for a precise comparison of the time behavior of the four machines:

(a) *Hierarchy*: we show that the KAM is never slower than the λJAM which is never slower than the λIAM, establishing a sort of hierarchy.
(b) *λJAM is (slowly) reasonable*: a close inspection shows that the λJAM is at most quadratically slower than the KAM. Since the KAM is a *reasonable*[2] implementation scheme, we obtain that the λJAM is reasonable as well.
(c) *λJAM and λPAM isomorphism*: we confirm Danos and Regnier's claim that the λJAM and the λPAM are isomorphic (and have the same time behavior), working out the elegant and yet far from trivial isomorphism[3].

[2]*Reasonable* is a technical word meaning polynomially related to Turing machines. In our context, a machine for Closed CbN is reasonable if the number of transition it takes on t is polynomial in the number of weak head β-steps to reduce t and in the size $|t|$ of t. For more details about reasonable cost models for the λ-calculus, see the overview in [Accattoli 2017].
[3]In the note [Danos and Regnier 2004], the authors claim that the PAM "is faster than the KAM in many cases" referring to private communication with Herbelin. Our results falsify the claim, as the PAM—behaving as the λJAM—is never faster and at most quadratically slower than the KAM.

(3) *λIAM Time and Multi Types.* We show how to extract the length of the λIAM run on a term t—that is, the time cost—from multi type derivations for t. This study complements de Carvalho's, showing that multi types can measure also the time of the λIAM, not just the KAM. A key point is that, by comparing how multi types measure game and environment machines, we obtain a clear insight about the time gap between the two approaches: the time usage of the λIAM depends on the multi type derivation *and* the size of the involved multi types, while the time usage of the KAM depends only on the former. Therefore, the gap is bigger on terms whose KAM evaluation is much shorter than the involved types[4]. The connection with multi types is obtained in two steps. First, we introduce a new abstract machine, the SIAM, running over multi type derivations and inspired by a GoI machine from [Dal Lago and Zorzi 2014], that we prove to be strongly bisimilar to the λIAM on typable terms. Second, we show how to extract the time of the SIAM from a multi type derivation, which—because of the bisimulation—is also the time of the λIAM. The second step, despite being a result similar to de Carvalho's for environment machines, requires a radically different proof technique, which is a further contribution of the paper.

(4) *A Uniform Proof Technique.* The proofs of the three bisimulations—namely between the λIAM and the λJAM, the λJAM and the KAM, and the λIAM and the SIAM—are all proved using the same novel technique. To prove the correctness of the λIAM, Accattoli et al. [2020b] study a new invariant, the *exhaustible (state) invariant*, expressing a form of coherence of the data structures used by the λIAM. Our theorems are all proved by adapting the exhaustible invariant to each specific case, providing concise technical developments and conceptual unity. This point contrasts strikingly with the original papers on game machines, whose proof techniques are involved, indirect[5], and often informal. The exhaustible invariant—while certainly technical—is simpler and provides direct arguments. It seems to be the key tool to study game machines. As a slogan, *interacting is exhausting*.

Our Two Cents about Space. At the end of the paper we also briefly discuss space. Specifically, we provide examples showing that the λIAM can use more space than the λJAM, despite the former being considered space-efficient and the latter being as space-inefficient as possible. This fact does not contradict the space efficiency of the λIAM, as it concerns only specific terms. Our example however shows that space relationships between game machines—if they can be established at all—are subtler than the time ones, and of a less uniform nature.

Our Results, at a Distance. The body of the paper is quite technical and this is inevitable, because abstract machines—for as much as they can be abstract—are low-level tools. It is however easy to provide a high-level perspective. Comparing with the original papers on game machines, our presentations play the role of a *Rosetta stone*, connecting concepts and decoding many technical subtleties and invariants. Our exhaustible invariant, additionally, removes the need to resort to game semantics or legal paths when relating the machines. Our complexity study suggests that interaction as modeled by HO games (seen as the λJAM and the λPAM) is a time-reasonable process, while as modeled by the GoI and AJM games (seen as the λIAM) is a time-inefficient process[6]. Our multi type study, however, suggests that the gap between the two is big only on terms whose type derivations are much smaller than the involved types. Focusing on HO games, the quadratic overhead of the λJAM with respect to the KAM shows that interaction as modeled by HO games is time-reasonable but not time-efficient. Summing up, *interacting takes time, and is exhausting.*

[4]Note that even the smallest multi type derivation for the inefficient IAM family t_n given above uses types exponential in n (inside the derivation).

[5]The relationship between the IAM and the PAM in [Danos et al. 1996] is not direct as it goes through both AJM games and HO games. Similarly, the relationship between the IAM and the JAM in [Danos and Regnier 1999] is not self-contained, as it is based on the non-trivial equivalence between regular and legal paths proved in [Asperti et al. 1994].

[6]Whether the IAM is reasonable is unclear. The mentioned time inefficiency of the IAM is not a proof that it is unreasonable.

A Summarizing Diagram. The obtained relationships between machines can be summarized by the following diagram.

$$\begin{tikzcd}
\lambda\text{IAM} \arrow[r, "\text{potential exponential}", "\text{speedup}"'] \arrow[d, leftrightarrow] & \lambda\text{JAM} \arrow[r, "\text{potential quadratic}", "\text{speedup}"'] \arrow[d, leftrightarrow] & \text{KAM} \\
\text{SIAM} & \lambda\text{PAM} &
\end{tikzcd}$$

Vertical arrows denote isomorphisms: the λIAM and the SIAM are strongly bisimilar (on typable and therefore terminating terms), and similarly the λJAM and the λPAM are strongly bisimilar (on whatever term). The horizontal arrows denote bisimulations where the machine on the right is never slower of the one on the left, and sometimes up to exponentially/quadratically faster.

Evaluating without Duplicating. Let us provide a conclusive insight. β-reducing a λ-term (potentially) duplicates arguments, whose different copies may be used differently, typically being applied to different further arguments. The machines in this paper never duplicate parts of the code[7], but have nonetheless to distinguish different uses of a same piece of code during execution. Each one does it in a clever and sophisticated different way—multi types also fit this view, as they remove duplications altogether by taking all the needed copies in advance (see Sect. 14). This paper can then be seen as a systematic and thorough study of *the art of evaluating without duplicating*.

Related Work. Beyond the works already cited above, game machines are also studied by Curien and Herbelin [1998, 2007] who consider different machines as directly obtained by game semantics, Mackie [2017] who derives a proof net based token machine for System T, Pinto [2001] who develops a parallel implementation of the IAM, and Fernández and Mackie [2002] who extend token machines to call-by-value. A game machine accommodating the additive connectives of linear logic is in [Laurent 2001]. Space-efficiency of variants of the IAM is addressed by Mazza [2015]; Mazza and Terui [2015]. Game machines for languages beyond the λ-calculus, like λ-calculi with algebraic effects, quantum λ-calculi or concurrent calculi are in [Dal Lago et al. 2014, 2015, 2017; Hoshino et al. 2014]. A different kind of machine inspired by the GoI is in [Danos and Regnier 1993; Pedicini and Quaglia 2007]. Interaction and rewriting are mixed in recent work by Muroya and Ghica [2017, 2019]. Clairambault [2011, 2015] uses an abstraction of the PAM to bound evaluation lengths, and similar studies are also done by Aschieri [2017]. *Traversals* are another operational tool connected to HO games, introduced by Ong [2006] and further developed by Blum [2020], see also [Berezun and Jones 2017].

The space inefficiency of environment machines has already been observed by Krishnaswami, Benton, and Hoffman, who proposed some techniques to alleviate it in the context of functional-reactive programming and based on linear types [Krishnaswami et al. 2012].

The time efficiency of environment machines has been recently studied in depth. Before 2014, the topic had been mostly neglected—the only two counterexamples being Blelloch and Greiner [1995]; Sands et al. [2002]. Since 2014—motivated by advances on time cost models for the λ-calculus by Accattoli and Dal Lago [2016]—the topic has actively been studied [Accattoli et al. 2014; Accattoli and Barras 2017; Accattoli et al. 2019a; Accattoli and Guerrieri 2019].

Intersection types are a standard tool to study λ-calculi—see standard references such as [Coppo and Dezani-Ciancaglini 1978, 1980; Krivine 1993; Pottinger 1980]. Non-idempotent intersection types, *i.e.* multi types, make their first appearance as a technical tool to study principal intersection

[7]Note that not all machines in the literature avoid duplications: machines with a single global environment duplicate pieces of the code, see [Accattoli and Barras 2017]. Perhaps surprisingly, performing duplications is not as costly as one may expect. Global environment machines are indeed time-efficient and faster than the game machines studied in this paper.

types in [Coppo et al. 1980]. They were first considered by themselves by Gardner [1994], and then by de Carvalho [2007, 2018]; Kfoury [2000]; Neergaard and Mairson [2004]—a survey is [Bucciarelli et al. 2017]. De Carvalho's use of multi types to give bounds to evaluation lengths has also been used in [Accattoli et al. 2020c; Accattoli and Guerrieri 2018; Accattoli et al. 2019b; Bernadet and Graham-Lengrand 2013; Bucciarelli et al. 2020; de Carvalho et al. 2011; Kesner and Vial 2020].

2 PRELIMINARIES: CLOSED CALL-BY-NAME, AND ABSTRACT MACHINES

Let $\mathcal{V}$ be a countable set of variables. Terms of the *λ-calculus* Λ are defined as follows.

$$\lambda\text{-TERMS} \quad t, u, r \quad ::= \quad x \in \mathcal{V} \mid \lambda x.t \mid tu.$$

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 *leveled* contexts, whose index, *i.e.* the level, stands for the number of arguments (*i.e.* the number of !-boxes in linear logic terminology) the hole lies in.

$$\text{LEVELED CONTEXTS}$$
$$C_0 \quad ::= \quad \langle\cdot\rangle \mid \lambda x.C_0 \mid C_0 t; \qquad C_{n+1} \quad ::= \quad C_{n+1}t \mid \lambda x.C_{n+1} \mid tC_n.$$

We simply write C for a context whenever the level is not relevant. The operation replacing the hole $\langle\cdot\rangle$ with a term t in a context C is noted $C\langle t\rangle$ and called *plugging*.

The operational semantics that we adopt here is weak head evaluation $\rightarrow_{wh}$, defined as follows:

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

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 $\rightarrow_{wh}$ is deterministic.

Abstract Machines Glossary. In this paper, an *abstract machine* $\mathrm{M} = (s, \rightarrow)$ is a transition system $\rightarrow$ over a set of states, noted s. The machines considered in this paper move over the code without ever changing it. A *position* in a term t is represented as a pair (u, C) of a sub-term u 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 data structures.

A state is *initial*, and noted s_t, if it is positioned on $(t, \langle\cdot\rangle)$, t is closed, and all the data structures are empty. We may write s_t^{M} to stress the machine, and t is always implicitly considered closed, without further mention. A state is *final* if no transitions apply.

A *run* $\pi : s \rightarrow^* s'$ is a possibly empty sequence of transitions, whose length is noted $|\pi|$. If a and b are transitions labels (that is, $\rightarrow_a \subseteq \rightarrow$ and $\rightarrow_b \subseteq \rightarrow$) then $\rightarrow_{a,b} := \rightarrow_a \cup \rightarrow_b$, $|\pi|_a$ is the number of a transitions in π, and $|\pi|_{\neg a}$ is the size of transitions in π that are not $\rightarrow_a$. An *initial run* is a run from an initial state s_t, and it is also called *a run from t*. A state s 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 s_t^{M}, and $\mathrm{M}(t)\Uparrow$ otherwise. We say that M *implements Closed CbN* when $\mathrm{M}(t)\Downarrow$ if and only if $\rightarrow_{wh}$ terminates on t, for every closed term t.

3 THE INTERACTION ABSTRACT MACHINE, REVISITED

In this section we provide an overview of the Interaction Abstract Machine (IAM). We adopt the λ-calculus presentation of the IAM, rather called λIAM and recently developed by Accattoli et al. [2020b]—we refer to their work for an in-depth study of the λIAM. The literature usually studies the (λ)IAM with respect to head evaluation of potentially open terms. Here we only deal with Closed CbN, that is closer to the practice of functional programming and also the setting underlying

$$\begin{array}{rlrl}
\text{LOGGED POSITIONS} & l ::= (t, C_n, L_n) & \text{TAPES} & T ::= \epsilon \mid \bullet\cdot T \mid l\cdot T \\
\text{LOGS} & L_0 ::= \epsilon \qquad L_{n+1} ::= l\cdot L_n & \text{DIRECTION} & d ::= \downarrow \mid \uparrow \\
\text{STATES} & s ::= (t, C, L, T, d) & &
\end{array}$$

Sub-term	Context	Log	Tape		Sub-term	Context	Log	Tape
$\underline{tu}$	C	L	T	$\rightarrow_{\bullet 1}$	$\underline{t}$	$C\langle\langle\cdot\rangle u\rangle$	L	$\bullet\cdot T$
$\underline{\lambda x.t}$	C	L	$\bullet\cdot T$	$\rightarrow_{\bullet 2}$	$\underline{t}$	$C\langle\lambda x.\langle\cdot\rangle\rangle$	L	T
$\underline{x}$	$C\langle\lambda x.D_n\rangle$	$L_n\cdot L$	T	$\rightarrow_{\text{var}}$	$\lambda x.D_n\langle x\rangle$	$\underline{C}$	L	$(x, \lambda x.D_n, L_n)\cdot T$
$\underline{\lambda x.D_n\langle x\rangle}$	C	L	$(x, \lambda x.D_n, L_n)\cdot T$	$\rightarrow_{\text{bt2}}$	x	$\underline{C\langle\lambda x.D_n\rangle}$	$L_n\cdot L$	T
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	L	$\bullet\cdot T$	$\rightarrow_{\bullet 3}$	tu	$\underline{C}$	L	T
t	$\underline{C\langle\lambda x.\langle\cdot\rangle\rangle}$	L	T	$\rightarrow_{\bullet 4}$	$\lambda x.t$	$\underline{C}$	L	$\bullet\cdot T$
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	L	$l\cdot T$	$\rightarrow_{\text{arg}}$	$\underline{u}$	$C\langle t\langle\cdot\rangle\rangle$	$l\cdot L$	T
t	$\underline{C\langle u\langle\cdot\rangle\rangle}$	$l\cdot L$	T	$\rightarrow_{\text{bt1}}$	$\underline{u}$	$C\langle\langle\cdot\rangle t\rangle$	L	$l\cdot T$

Fig. 1. Data structures and transitions of the λ Interaction Abstract Machine (λIAM).

the KAM, studied in sections 5 and 8. Keep in mind that the λIAM is an unusual machine, and that finding it hard to grasp is normal—probably, the next sections about the λJAM and the KAM provide clarifying insights. Also, in [Accattoli et al. 2020b] there is an alternative explanation of the λIAM, that may be helpful, together with the relationship with proof nets, which is however not needed here.

Bird's Eye view of the λIAM. Intuitively, the behaviour of the λIAM 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 head variable then the λIAM 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 (which however has nothing to do with backtracking as modeled by classical logic and continuations), 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*. There are no easy intuitions about how logs handle different uses—this is both the magic and the mystery of the geometry of interaction.

λIAM States. The transitions of the λIAM and all the data structures are defined in Fig. 1. The λIAM travels on a λ-term t carrying data structures—representing the token—storing information about the computation and determining the next transition to apply. A key point is that navigation is done locally, moving only between adjacent positions[8]. The λIAM 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, and a logged position is a position plus a log, meant to trace the history that led to that position. Logs and logged positions are defined by mutual induction[9]. Note

[8]Note that also the transition from the variable occurrence to the binder in $\rightarrow_{\text{var}}$ and $\rightarrow_{\text{bt2}}$ 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 [Accattoli and Barras 2017].

[9]This is similar to the KAM, where closures and environments are defined by mutual induction, but logs and logged positions play a different role. Moreover, there also is a constraint about the length.

that in the definition of a logged position the log is required to have length n, where n is the level of the context of the position. 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 symbol $\bullet$, needed to record the crossing of abstractions and applications. A *state* of the machine is given by a position and a token (that is, a log L and a tape T), together with a *direction*. Initial states have the form $s_t := (\underline{t}, \langle\cdot\rangle, \epsilon, \epsilon)$. 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.

Transitions. Intuitively, the machine evaluates the term t until the head abstraction of the head normal form is found (more explanations below). The transitions of the λIAM are in Fig. 1. Their union is noted $\rightarrow_{\lambda\text{IAM}}$. 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.

Next, we explain how the transitions realize three entangled mechanisms. Let us anticipate that the λJAM shall be obtained by short-circuiting the third mechanism, backtracking, and the KAM by the further removal of the second one, that shall also require to modify the first one.

	Sub-term	Context	Log	Tape	Dir
	$\underline{(\lambda y.\lambda x.xy)\mathsf{II}}$	$\langle\cdot\rangle$	ϵ	ϵ	$\downarrow$
$\rightarrow_{\bullet 1}$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}}$	$\langle\cdot\rangle\mathsf{I}$	ϵ	$\bullet$	$\downarrow$
$\rightarrow_{\bullet 1}$	$\underline{\lambda y.\lambda x.xy}$	$\langle\cdot\rangle\mathsf{II}$	ϵ	$\bullet\cdot\bullet$	$\downarrow$
$\rightarrow_{\bullet 2}$	$\underline{\lambda x.xy}$	$(\lambda y.\langle\cdot\rangle)\mathsf{II}$	ϵ	$\bullet$	$\downarrow$
$\rightarrow_{\bullet 2}$	$\underline{xy}$	$(\lambda y.\lambda x.\langle\cdot\rangle)\mathsf{II}$	ϵ	ϵ	$\downarrow$

Mechanism 1: Search Up to β-Redexes. Note that $\rightarrow_{\bullet 1}$ skips the argument and adds a $\bullet$ on the tape. The idea is that $\bullet$ keeps track that an argument has been encountered—its identity is however forgotten. Then $\rightarrow_{\bullet 2}$ does the dual job: it skips an abstraction when the tape carries a $\bullet$, that is, the trace of a previously encountered argument. Note that, when the λIAM moves through a β-redex with the two steps one after the other, the token is left unchanged. This mechanism thus realizes search *up to β-redexes*, that is, without ever recording them. Note that $\rightarrow_{\bullet 3}$ and $\rightarrow_{\bullet 4}$ realize the same during the $\uparrow$ phase. Let us illustrate this mechanism with an example (on the right): the first steps of evaluation on the term $(\lambda y.\lambda x.xy)\mathsf{II}$, where I is the identity combinator. One can notice that the λIAM traverses two β-redexes without altering the token, that is empty both at the beginning and at the end.

Mechanism 2: Finding Variables and Arguments. As a first approximation, navigating in direction $\downarrow$ corresponds to looking for the head variable of the term code, while navigating with direction $\uparrow$ corresponds to looking for the sub-term to replace the previously found head variable, what we call *the argument*. More precisely, when the head variable x of the active subterm is found, transition $\rightarrow_{\text{var}}$ switches direction from $\downarrow$ to $\uparrow$, and the machine starts looking for potential substitutions for x. The λIAM then moves to the position of the binder λx of x, and starts exploring the context C, looking for the first argument up to β-redexes. The relative position of x w.r.t. its binder is recorded in a new logged position that is added to the tape. Since the machine moves out of a context of level n, namely D_n, the logged position contains the first n logged positions of the log. Roughly, this is an encoding of the run that led from the level of $\lambda x.D_n\langle x\rangle$ to the occurrence of x at hand, in case the machine would later need to backtrack.

When the argument t for the abstraction binding the variable x in l is found, transition $\rightarrow_{\text{arg}}$ switches direction from $\uparrow$ to $\downarrow$, making the machine looking for the head variable of t. Note that moving to t, the level increases, and that the logged position l is moved from the tape to the log. The idea is that l is now a completed argument query, and it becomes part of the history of how the machine got to the current position, to be potentially used for backtracking. We continue the example of the previous point: the machine finds the head variable x and looks for its argument in $\uparrow$ mode. When it has been found, the direction turns to $\downarrow$ again.

	Sub-term	Context	Log	Tape	Dir
	$\underline{xy}$	$(\lambda y.\lambda x.\langle\cdot\rangle)\mathsf{II}$	ϵ	ϵ	$\downarrow$
$\to_{\bullet 1}$	$\underline{x}$	$(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{II}$	ϵ	$\bullet$	$\downarrow$
$\to_{\text{var}}$	$\lambda x.xy$	$\underline{(\lambda y.\langle\cdot\rangle)\mathsf{II}}$	ϵ	$(x, \lambda x.\langle\cdot\rangle y, \epsilon)\cdot\bullet$	$\uparrow$
$\to_{\bullet 4}$	$\lambda y.\lambda x.xy$	$\underline{\langle\cdot\rangle\mathsf{II}}$	ϵ	$\bullet\cdot(x, \lambda x.\langle\cdot\rangle y, \epsilon)\cdot\bullet$	$\uparrow$
$\to_{\bullet 3}$	$(\lambda y.\lambda x.xy)\mathsf{I}$	$\underline{\langle\cdot\rangle\mathsf{I}}$	ϵ	$(x, \lambda x.\langle\cdot\rangle y, \epsilon)\cdot\bullet$	$\uparrow$
$\to_{\text{arg}}$	$\underline{\mathsf{I}}$	$(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle$	$(x, \lambda x.\langle\cdot\rangle y, \epsilon)$	$\bullet$	$\downarrow$

Mechanism 3: Backtracking. It is started by transition $\to_{\text{bt1}}$. The idea is that the search for an argument of the $\uparrow$-phase has to temporarily stop, because there are no arguments left at the current level. The search of the argument then has to be done among the arguments of the variable occurrence that triggered the search, encoded in l. Then the machine enters into backtracking mode, which is denoted by a $\downarrow$-phase with a logged position on the tape, to reach the position in l. Backtracking is over when $\to_{\text{bt2}}$ is fired.

The $\downarrow$-phase and the logged position on the tape mean that the λIAM is backtracking. In fact, in this configuration the machine is not looking for the head variable of the current subterm $\lambda x.t$, it is rather going back to the variable position in the tape, to find its argument. This is realized by moving to the position in the tape and changing direction. Moreover, the log L_n encapsulated in the logged position is put back on the global log. An invariant shall guarantee that the logged position on the tape always contains a position relative to the active abstraction. In our running example, a backtracking phase, noted with a BT label, starts when the IAM looks for the argument of z. Since $\lambda z.z$ has been virtually substituted for x, its argument its actually y. Backtracking is needed to recover the variable a term was virtually substituted for.

		Sub-term	Context	Log	Tape	Dir
		$\underline{\lambda z.z}$	$(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle$	$(x, \lambda x.\langle\cdot\rangle y, \epsilon)$	$\bullet$	$\downarrow$
	$\to_{\bullet 2}$	$\underline{z}$	$(\lambda y.\lambda x.xy)\mathsf{I}(\lambda z.\langle\cdot\rangle)$	$(x, \lambda x.\langle\cdot\rangle y, \epsilon)$	ϵ	$\downarrow$
	$\to_{\text{var}}$	$\lambda z.z$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle}$	$(x, \lambda x.\langle\cdot\rangle y, \epsilon)$	$(z, \lambda z.\langle\cdot\rangle, \epsilon)$	$\uparrow$
BT	$\to_{\text{bt1}}$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}}$	$\langle\cdot\rangle\mathsf{I}$	ϵ	$(x, \lambda x.\langle\cdot\rangle y, \epsilon)\cdot(z, \lambda z.\langle\cdot\rangle, \epsilon)$	$\downarrow$
BT	$\to_{\bullet 1}$	$\underline{\lambda y.\lambda x.xy}$	$\langle\cdot\rangle\mathsf{II}$	ϵ	$\bullet\cdot(x, \lambda x.\langle\cdot\rangle y, \epsilon)\cdot(z, \lambda z.\langle\cdot\rangle, \epsilon)$	$\downarrow$
BT	$\to_{\bullet 2}$	$\underline{\lambda x.xy}$	$(\lambda y.\langle\cdot\rangle)\mathsf{II}$	ϵ	$(x, \lambda x.\langle\cdot\rangle y, \epsilon)\cdot(z, \lambda z.\langle\cdot\rangle, \epsilon)$	$\downarrow$
	$\to_{\text{bt2}}$	x	$\underline{(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{II}}$	ϵ	$(z, \lambda z.\langle\cdot\rangle, \epsilon)$	$\uparrow$

For the sake of completeness, we conclude the example, which runs until the head abstraction of the weak head normal form of the term under evaluation, namely the first occurrence of I, is found.

	Sub-term	Context	Log	Tape	Dir
	x	$\underline{(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{II}}$	ϵ	$(z, \lambda z.\langle\cdot\rangle, \epsilon)$	$\uparrow$
$\to_{\text{arg}}$	$\underline{y}$	$(\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{II}$	$(z, \lambda z.\langle\cdot\rangle, \epsilon)$	ϵ	$\downarrow$
$\to_{\text{var}}$	$\lambda y.\lambda x.xy$	$\underline{\langle\cdot\rangle\mathsf{II}}$	ϵ	$(y, \lambda x.x\langle\cdot\rangle, (z, \lambda z.\langle\cdot\rangle, \epsilon))$	$\uparrow$
$\to_{\text{arg}}$	$\underline{\mathsf{I}}$	$(\lambda y.\lambda x.xy)\langle\cdot\rangle\mathsf{I}$	$(y, \lambda x.x\langle\cdot\rangle, (z, \lambda z.\langle\cdot\rangle, \epsilon))$	ϵ	$\downarrow$

Last, note that our example is a linear λ-term. The technical report associated to this paper [Accattoli et al. 2020a] contains an example showing how the λIAM uses logged positions to deal with duplications.

Basic invariants. Given a state (t, C, L, T, d), the log and the tape, *i.e.* the token, verify two easy invariants connecting them to the position (t, C) and the direction d. The log L, together with the position (t, C), forms a logged position, *i.e.* the length of L is exactly the level of the code context C[10]. This fact guarantees that the λIAM never gets stuck because the log is not long enough for transitions $\to_{\text{var}}$ and $\to_{\text{bt1}}$ to apply.

About the tape, note that every time the machine switches from a $\downarrow$-state to an $\uparrow$-state (or vice versa), a logged position is pushed (or popped) from the tape T. Thus, for reachable states, the

[10] Then, the length of L is exactly the number of (linear logic) *boxes* in which the code term is contained.

number of logged positions in T gives the direction of the state. These intuitions are formalized by the *tape and direction* invariant below. Given a direction d we use d^n for the direction obtained by switching d exactly n times (i.e., $\downarrow^0 = \downarrow$, $\uparrow^0 = \uparrow$, $\downarrow^{n+1} = \uparrow^n$ and $\uparrow^{n+1} = \downarrow^n$).

Lemma 3.1 (λIAM basic invariants). *Let $s = (t, C_n, L, T, d)$ be a reachable state and $|T|_l$ the number of logged positions in T. Then*

(1) Position and log*: (t, C_n, L) is a logged position, and*
(2) Tape and direction*: $d = \downarrow^{|T|_l}$.*

Final States. If the λIAM starts on the initial state s_t, the execution may either never stop or end in a state s of the shape $s = (\underline{\lambda x.u}, C, L, \epsilon)$. The fact that no other shapes are possible for s is proved in Accattoli et al. [2020b]. The *tape and direction* invariant guarantees that the machine never stops because the log or the tape have not enough logged positions to apply a $\rightarrow_{\text{var}}$, $\rightarrow_{\text{bt1}}$, or a $\rightarrow_{\text{arg}}$ transition. Additionally, on states such as $(\underline{\lambda x.D\langle x\rangle}, C, L, l{\cdot}T)$, the logged position l has shape $(x, \lambda x.D, L')$, so that transition $\rightarrow_{\text{bt2}}$ can always apply—this is a consequence of the *exhaustible state invariant* in Sect. 6, as shown in Accattoli et al. [2020b].

The exhaustible state invariant shall be the technical blueprint for the proof of the relationship between the λIAM and the λJAM, amounting to short-circuiting backtracking phases. Similarly, we shall use variants of it to relate the KAM and the λJAM, and the λIAM with multi type derivations.

Implementation. Usually, the λIAM is shown to implement (a micro-step variant of) head reduction. The details are quite different from those in the usual notion of implementation for environment machines, such as the KAM. Essentially, it is shown that the λIAM induces a semantics $[\![\cdot]\!]_{\lambda\text{IAM}}$ of terms that is a sound and adequate with respect to head reduction, rather than showing a bisimulation between the machine and head reduction—this is explained at length in [Accattoli et al. 2020b]. For the sake of simplicity, here we restrict to Closed CbN. The λIAM semantics then reduces to just observing termination: $[\![t]\!]_{\lambda\text{IAM}}$ is defined if and only if weak head reduction terminates on t. Therefore, we avoid discussing semantics and only study termination.

Theorem 3.2 ([Accattoli et al. 2020b]). *The λIAM implements Closed CbN.*

Cost of λIAM Transitions. For all the abstract machines in this paper we take random access machines (shortened to RAM) with the uniform cost model as the reference computational model. This is standard in the time analyses of abstract machines for functional languages. Roughly, it amounts to seeing variables and positions as objects (namely pointers) whose manipulations take constant time.

Every λIAM transition can then be implemented on RAM in constant time but for transition $\rightarrow_{\text{var}}$, whose cost is bounded by the integer n given by D_n (referring to the notation of the rules), as the rule needs to split the log after the first n entries. This is in accordance with the proof nets interpretation of the λIAM, because transitions $\rightarrow_{\text{var}}$ correspond to *sequences* of IAM transitions on proof nets—see [Accattoli et al. 2020b][11]. Note that n is bound by the size $|t|$ of the (immutable) initial code t. The cost of implementing on RAM a λIAM run π from t then is $|\pi|_{\neg\text{var}} + |\pi|_{\text{var}} \cdot |t|$.

Two Useful Properties of the λIAM. A key property is that the λIAM is *bi-deterministic*, that is, it is deterministic and also deterministically reversible. Another more technical property is that it verifies a sort of context-freeness with respect to the tape T. Namely, extending the tape preserves the shape of the run and of the final state (up to the extension).

[11] Actually, also transition $\rightarrow_{\text{bt2}}$ corresponds to n IAM transitions on proof nets. By implementing logs as bi-linked lists, however, $\rightarrow_{\text{bt2}}$ can be implemented in constant time. For $\rightarrow_{\text{var}}$ instead, there is no easy way out.

Sub-term	Context	Log	Tape		Sub-term	Context	Log	Tape
$\underline{tu}$	C	L	T	$\to_{\bullet 1}$	$\underline{t}$	$C\langle\langle\cdot\rangle u\rangle$	L	$\bullet\cdot T$
$\underline{\lambda x.t}$	C	L	$\bullet\cdot T$	$\to_{\bullet 2}$	$\underline{t}$	$C\langle\lambda x.\langle\cdot\rangle\rangle$	L	T
$\underline{x}$	$C\langle\lambda x.D_n\rangle$	$L_n\cdot L$	T	$\to_{\text{var}}$	$\lambda x.D_n\langle x\rangle$	$\underline{C}$	L	$(x,\, C\langle\lambda x.D_n\rangle,\, L_n\cdot L)\cdot T$
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	L	$\bullet\cdot T$	$\to_{\bullet 3}$	tu	$\underline{C}$	L	T
t	$\underline{C\langle\lambda x.\langle\cdot\rangle\rangle}$	L	T	$\to_{\bullet 4}$	$\lambda x.t$	$\underline{C}$	L	$\bullet\cdot T$
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	L	$l\cdot T$	$\to_{\text{arg}}$	$\underline{u}$	$C\langle t\langle\cdot\rangle\rangle$	$l\cdot L$	T
t	$\underline{C\langle u\langle\cdot\rangle\rangle}$	$(x,\, D,\, L')\cdot L$	T	$\to_{\text{jmp}}$	x	$\underline{D}$	L'	T

Fig. 2. Transitions of the λ Jumping Abstract Machine (λJAM).

Lemma 3.3 (λIAM tape lift). *Let T be a tape and $\pi : s = (t, C, L, T', d) \to^n_{\lambda IAM} (u, D, L', T'', d') = s'$ a run. Then there is a λIAM run $\pi^T : s^T = (t, C, L, T'\cdot T, d) \to^n_{\lambda IAM} (u, D, L', T''\cdot T, d') = s'^T$.*

4 THE JUMPING ABSTRACT MACHINE, REVISITED

The Jumping Abstract Machine (JAM) is introduced in [Danos and Regnier 1999] as an optimization of the IAM obtained via a sophisticated analysis of IAM runs. Here we present the λJAM, the recasting of the JAM in the same syntactic framework of the λIAM. In particular, the λIAM and the λJAM rest on the same grammars and data structures, they only differ on some transitions.

Jumping Around the Log. The difference between the λIAM and the λJAM is in how they create logged positions, and consequently on how they backtrack. The λIAM has a *local* approach to logs, and backtracks via potentially long sequences of transitions, while the λJAM follows a *global* approach to logs, and it backtracks in just one *jump*. The transition system is presented in Fig. 2. The details of the two variations over the λIAM are:

- *Global logged position*: logged positions created by rule $\to_{\text{var}}$ are now global, in that they record the global position of the variable, and not only the position relative to its binder. This way, also the log has to be entirely copied. Differently from the λIAM, there is some duplication of information.
- *Backtracking is short-circuited*: backtracking is a phase of a λIAM run which is contained between $\to_{\text{bt1}}$ and $\to_{\text{bt2}}$ transitions acting on the same logged position. It starts when the machine has to rebuild the history of a redex/substitution and ends when the substituted variable occurrence l is found. The optimization at the heart of the λJAM comes from the observation that the λIAM backtracks to the exact same state that created l. This way, one use l to jump directly to that state instead of doing the backtracking. Of course, this is possible only if positions are saved globally in logged positions: note that the λIAM saves in l only part of the log of the state creating l, while to jump back and avoid backtracking one needs to save the whole log.

The absence of the backtracking phase makes the λJAM easier to understand than the λIAM. In particular, the $\downarrow$ and $\uparrow$ phases have now a precise meaning: the former being the quest for the head variable of the current subterm, and the latter being the search of the argument of the *only* variable occurrence in the tape. This is the second point of the following lemma.

Lemma 4.1 (λJAM basic invariants). *Let $s = (t, C_n, L, T, d)$ be a reachable state. Then*

(1) Position and log: *(t, C_n, L) is a logged position, and*
(2) Tape and direction: *if $d = \downarrow$, then T does not contain any logged position, otherwise, if $d = \uparrow$, then T contains exactly one logged position.*

Since the λJAM is an optimization of the λIAM, its final states have the same shape, namely $(\underline{\lambda x.u}, C, L, \epsilon)$. The fact that the log is always long enough to apply transition $\rightarrow_{\text{var}}$ is given by the *position and log* invariant above. In the next section we shall prove that the λIAM and the λJAM are termination equivalent, obtaining as a corollary that the λJAM implements Closed CbN.

An Example. We present the λJAM execution trace of the same term considered for the λIAM. In particular, the first transitions are identical to the λIAM execution since no $\rightarrow_{\text{var}}$ and $\rightarrow_{\text{bt1}}$ rules are involved. Instead, we observe that full context and log are saved at the occurrence of $\rightarrow_{\text{var}}$ transitions. We set $l_x := (x, (\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{I}(\lambda z.z), \epsilon)$.

	Sub-term	Context	Log	Tape	Dir
	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}\mathsf{I}}$	$\langle\cdot\rangle$	ϵ	ϵ	$\downarrow$
$\rightarrow_{\bullet 1}$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}}$	$\langle\cdot\rangle\mathsf{I}$	ϵ	$\bullet$	$\downarrow$
$\rightarrow_{\bullet 1}$	$\underline{\lambda y.\lambda x.xy}$	$\langle\cdot\rangle\mathsf{I}\mathsf{I}$	ϵ	$\bullet\cdot\bullet$	$\downarrow$
$\rightarrow_{\bullet 2}$	$\underline{\lambda x.xy}$	$(\lambda y.\langle\cdot\rangle)\mathsf{I}\mathsf{I}$	ϵ	$\bullet$	$\downarrow$
$\rightarrow_{\bullet 2}$	$\underline{xy}$	$(\lambda y.\lambda x.\langle\cdot\rangle)\mathsf{I}\mathsf{I}$	ϵ	ϵ	$\downarrow$
$\rightarrow_{\bullet 1}$	$\underline{x}$	$(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{I}\mathsf{I}$	ϵ	$\bullet$	$\downarrow$

	Sub-term	Context	Log	Tape	Dir
	$\underline{x}$	$(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{I}(\lambda z.z)$	ϵ	$\bullet$	$\downarrow$
$\rightarrow_{\text{var}}$	$\lambda x.xy$	$\underline{(\lambda y.\langle\cdot\rangle)\mathsf{I}(\lambda z.z)}$	ϵ	$l_x\cdot\bullet$	$\uparrow$
$\rightarrow_{\bullet 4}$	$\lambda y.\lambda x.xy$	$\underline{\langle\cdot\rangle\mathsf{I}(\lambda z.z)}$	ϵ	$\bullet\cdot l_x\cdot\bullet$	$\uparrow$
$\rightarrow_{\bullet 3}$	$(\lambda y.\lambda x.xy)\mathsf{I}$	$\underline{\langle\cdot\rangle(\lambda z.z)}$	ϵ	$l_x\cdot\bullet$	$\uparrow$
$\rightarrow_{\text{arg}}$	$\underline{(\lambda z.z)}$	$(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle$	l_x	$\bullet$	$\downarrow$
$\rightarrow_{\bullet 2}$	$\underline{z}$	$(\lambda y.\lambda x.xy)\mathsf{I}(\lambda z.\langle\cdot\rangle)$	l_x	ϵ	$\downarrow$
$\rightarrow_{\text{var}}$	$\lambda z.z$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle}$	l_x	$(z, (\lambda y.\lambda x.xy)\mathsf{I}(\lambda z.\langle\cdot\rangle), l_x)$	$\uparrow$

Finally, as already explained, backtracking is avoided by jumping: the λJAM restores the previously encountered state, saved in the logged position l_x, when exiting from the right-hand side of an application. We set $l_z := (z, (\lambda y.\lambda x.xy)\mathsf{I}(\lambda z.\langle\cdot\rangle), l_x)$.

	Sub-term	Context	Log	Tape	Dir
	$\lambda z.z$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle}$	l_x	l_z	$\uparrow$
$\rightarrow_{\text{jmp}}$	x	$\underline{(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{I}(\lambda z.z)}$	ϵ	l_z	$\uparrow$
$\rightarrow_{\text{arg}}$	$\underline{y}$	$(\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I}$	l_z	ϵ	$\downarrow$
$\rightarrow_{\text{var}}$	$\lambda y.\lambda x.xy$	$\underline{\langle\cdot\rangle\mathsf{I}\mathsf{I}}$	ϵ	$(y, (\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I}, l_z)$	$\uparrow$
$\rightarrow_{\text{arg}}$	$\underline{\mathsf{I}}$	$(\lambda y.\lambda x.xy)\langle\cdot\rangle\mathsf{I}$	$(y, (\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I}, l_z)$	ϵ	$\downarrow$

As for the λIAM, [Accattoli et al. 2020a] contains an example showing how the λJAM deals with duplications.

Cost of λJAM Transitions. The cost of implementing λJAM transitions and runs on RAM is exactly the same as for the IAM: all transitions are atomic but for $\rightarrow_{\text{var}}$, whose cost is given by the level n of the involved context D_n, itself bound by the size of the initial code t. Note that this means that in $\rightarrow_{\text{var}}$ the duplication of the log L amounts to the duplication of the pointer to the concrete representation of L, and not of the whole of L (that would make the cost of $\rightarrow_{\text{var}}$ much higher, namely depending on the length of the whole run that led to the transition).

5 KRIVINE ABSTRACT MACHINE

The Krivine abstract machine [Krivine 2007] (KAM) is a standard environment machine for Closed CbN whose time behavior has been studied thoroughly—in Sect. 10 we recall the literature about it. In particular, it is a time reasonable implementation of Closed CbN, where *reasonably* means polynomially related to the time cost model of Turing machines. To be uniform with respect to the other machines, we present the KAM adding information about the context, which is not needed.

Hopping on Arguments. The KAM (in Fig. 3) differs from the λIAM and λJAM as it *does record* every β-redex that it encounters—thus explicitly entangling time and space consumption—using two data structures. Log and tape are replaced by the *(local) environment* E and the *(applicative)*

Closures	Environments	Stacks	States
$c ::= (t, C, E)$	$E ::= \epsilon \mid [x{\leftarrow}c] \cdot E$	$S ::= \epsilon \mid c \cdot S$	$s ::= (t, C, E, S)$

Term	Ctx	Env	Stack		Term	Ctx	Env	Stack
tu	C	E	S	$\rightarrow_{\text{app}}$	t	$C\langle\langle\cdot\rangle u\rangle$	E	$(u, C\langle t\langle\cdot\rangle\rangle, E)\cdot S$
$\lambda x.t$	C	E	$c\cdot S$	$\rightarrow_{\text{abs}}$	t	$C\langle\lambda x.\langle\cdot\rangle\rangle$	$[x{\leftarrow}c]\cdot E$	S
x	C	$E\cdot[x{\leftarrow}(u, D, E'')]\cdot E'$	S	$\rightarrow_{\text{var}}$	u	D	E''	S

Fig. 3. Data structures and transitions of the Krivine Abstract Machine (KAM).

stack S. The basic idea is that, by saving encountered β-redexes in the environment E, when the machine finds a variable occurrence x it simply looks up in E for the argument of the binder λx binding x, avoiding the $\uparrow$-phase of the λJAM—note that the KAM has no $\uparrow$ phase and no logs. Mimicking the *jump* terminology, one may say that KAM transition $\rightarrow_{\text{var}}$ *hops* directly on the argument, skipping the search for it. The stack S 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. The intuition is that the stack has an entry for every occurrences of $\bullet$ on the tape of the λJAM in the $\downarrow$-phase, but such entries are more informative, they actually record the encountered argument (and a copy of the environment, explained next), and not just acknowledge its presence via $\bullet$.

Closures, Stacks, and Environments. The mutually recursive grammars for *closures* and *environments*, plus the independent one for *stacks* are defined in Fig. 3, 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. Also, when the machine executes a closed term t, every closure (u, C, E) in a reachable state is such that for any free variable x of u there is an entry $[x{\leftarrow}c]$ in E, thus E "closes" u, whence the name *closures*.

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. 3—their union is noted $\rightarrow_{KAM}$. The idea is that the $\rightarrow_{\text{var}}$ transition looks in the environment for the argument of the variable under evaluation. As for the other machines, 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 mentioned and standard (but omitted) invariant ensuring that when the initial term is closed, then every variable appearing in the code has an associated closure in the environment, so that the KAM never gets stuck on a $\rightarrow_{\text{var}}$ transition. In the next section we shall prove that the λJAM and the KAM are termination equivalent.

We show the KAM execution trace of our running example. Initially, the KAM looks for the head variable keeping track of the encountered arguments. We set $\mathsf{Q} := \lambda y.\lambda x.xy$.

	Sub-term	Context	Env.	Stack
	$\underline{(\lambda y.\lambda x.xy)\mathsf{II}}$	$\langle\cdot\rangle$	ϵ	ϵ
$\rightarrow_{\text{app}}$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}}$	$\langle\cdot\rangle\mathsf{I}$	ϵ	$(\mathsf{I}, \mathsf{QI}\langle\cdot\rangle, \epsilon)$
$\rightarrow_{\text{app}}$	$\underline{\lambda y.\lambda x.xy}$	$\langle\cdot\rangle\mathsf{II}$	ϵ	$(\mathsf{I}, \mathsf{Q}\langle\cdot\rangle\mathsf{I}, \epsilon)\cdot(\mathsf{I}, \mathsf{QI}\langle\cdot\rangle, \epsilon)$
$\rightarrow_{\text{abs}}$	$\underline{\lambda x.xy}$	$(\lambda y.\langle\cdot\rangle)\mathsf{II}$	$[y{\leftarrow}(\mathsf{I}, \mathsf{Q}\langle\cdot\rangle\mathsf{I}, \epsilon)]$	$(\mathsf{I}, \mathsf{QI}\langle\cdot\rangle, \epsilon)$
$\rightarrow_{\text{abs}}$	$\underline{xy}$	$(\lambda y.\lambda x.\langle\cdot\rangle)\mathsf{II}$	$[x{\leftarrow}(\mathsf{I}, \mathsf{QI}\langle\cdot\rangle, \epsilon)]\cdot[y{\leftarrow}(\mathsf{I}, \mathsf{Q}\langle\cdot\rangle\mathsf{I}, \epsilon)] = E$	ϵ
$\rightarrow_{\text{app}}$	$\underline{x}$	$(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{II}$	E	$(y, (\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{II}, E)$

Thanks to the information saved in the environment, the KAM is able to directly hop to the argument of x, namely the second identity. Moreover, the environment is restored from the closure.

Sub-term	Context	Env.	Stack
$\underline{x}$	$(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{I}(\lambda z.z)$	E	$(y, (\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I}, E)$
$\rightarrow_{\mathsf{var}}$ $\underline{(\lambda z.z)}$	$(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle$	ϵ	$(y, (\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I}, E)$

Then, the computation continues. All application arguments are saved in the stack as closures, *i.e.* together with their environment, and then moved to the environment when a binder λx is encountered (they are also linked to x). Whenever a variable x is reached, its argument is retrieved, together with its environment from the closure linked to that variable x.

Sub-term	Context	Env.	Stack
$\underline{(\lambda z.z)}$	$(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle$	ϵ	$(y, (\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I}, E)$
$\rightarrow_{\mathsf{abs}}$ $\underline{z}$	$(\lambda y.\lambda x.xy)\mathsf{I}(\lambda z.\langle\cdot\rangle)$	$[z\leftarrow(y, (\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I}, E)]$	ϵ
$\rightarrow_{\mathsf{var}}$ $\underline{y}$	$(\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I}$	$[x\leftarrow(\mathsf{I}, \mathsf{Q}\mathsf{I}\langle\cdot\rangle, \epsilon)]\cdot[y\leftarrow(\mathsf{I}, \mathsf{Q}\langle\cdot\rangle\mathsf{I}, \epsilon)]$	ϵ
$\rightarrow_{\mathsf{var}}$ $\underline{\mathsf{I}}$	$(\lambda y.\lambda x.xy)\langle\cdot\rangle\mathsf{I}$	ϵ	ϵ

As for the other machines, [Accattoli et al. 2020a] contains an example showing how the KAM deals with duplications.

Cost of KAM Transitions. The idea is that environments are implemented as linked lists, so that the duplication and insertion operations in transitions $\rightarrow_{\mathsf{app}}$ and $\rightarrow_{\mathsf{abs}}$ can be implemented in constant time. Transition $\rightarrow_{\mathsf{var}}$ needs to access the environment, whose size is bounded by $|t|$, the size of the initial term t of the run. By adopting smarter implementations of envrionments, one $\rightarrow_{\mathsf{var}}$ transition costs $\log|t|$—see Accattoli and Barras [2017] for discussions about implementations of the KAM. Then implementing on RAM a KAM run π from t costs $|\pi|_{\neg\mathsf{var}} + |\pi|_{\mathsf{var}} \cdot \log|t|$.

6 THE EXHAUSTIBLE STATE INVARIANT

Here we present the *exhaustible (state) invariant.* In [Accattoli et al. 2020b], this is a key ingredient for the proof of the λIAM implementation theorem. In this paper, we give it in various forms to establish the relationships between the various machines. Here we present the basic concepts.

The intuition behind the invariant is that whenever a logged position l occurs in a reachable state, it is there *for a reason*: no logged position occurs in initial states, and transitions only add logged positions to which the machine may come back. In particular, if the state is set in the right way (to be explained), the λIAM can reach l, *exhausting* it.

Preliminaries. Exhaustible states rest on some *tests* for their logged positions. More specifically, each logged position l in a state s has an associated test state s_l that tunes the data structures of s as to test for the reachability of l. Actually, there shall be *two* classes of test states, one accounting for the logged positions in the tape of s, and one for the those in the log of s. The technical definition of log tests, however, is in the technical report [Accattoli et al. 2020a]. They are essential for the proof of the exhaustible invariant, but they are not needed for showing the main consequence of interest in this section, that is, that backtracking always succeeds (Lemma 6.5 below), which is why they are omitted.

Tape Tests. Tape tests are easy to define. They focus on one of the logged positions in the tape, discarding everything that follows that position on the tape.

Definition 6.1 (Tape tests). Let $s = (t, C_n, L_n, T'{\cdot}l{\cdot}T'', d)$ be a state. Then the *tape test of s of focus l* is the state $s_l = (t, C_n, L_n, T'{\cdot}l, \uparrow^{|T'\cdot l|_l})$.

Note that the direction of tape tests is reversed with respect to what stated by the *tape and direction* invariant (Lemma 3.1), and so, in general, they are not reachable states. Such a counter-intuitive fact is needed for the invariant to go through, no more no less. When we shall use the properties of tests to prove properties of the λIAM (Lemma 6.5 below), we shall extend their tape via the tape lifting property (Lemma 3.3) as to satisfy the invariant and be reachable. Exhausting a logged position l means backtracking to it. We then decorate the backtracking transition $\rightarrow_{\mathrm{bt1}}$ and $\rightarrow_{\mathrm{bt2}}$ as $\rightarrow_{\mathrm{bt1},l}$ and $\rightarrow_{\mathrm{bt2},l}$ to specify the involved logged position l. We also need a notion of state positioned in l and having an empty tape, which is meant to be the target state of $\rightarrow_{\mathrm{bt2},l}$ when exhausting l starting on s_l.

Definition 6.2 (State surrounding a position). Let $l = (t, D, L)$ be a logged position. A state s surrounds l if $s = (t, \underline{C_n\langle D\rangle}, L \cdot L_n, \epsilon)$ for some C_n and L_n.

The Exhaustibility Invariant. After having introduced all the necessary preliminaries, we can now formulate the property of states that we shall next prove to be an invariant.

Definition 6.3 (Exhaustible states). $\mathcal{E}$ is the smallest set of states s such that if s_l is a tape or a log test of s then there exists a run $\pi : s_l \rightarrow^*_{\lambda\mathrm{IAM}} \rightarrow_{\mathrm{bt2},l} s'$, where s' surrounds l and for the shortest of such runs π it holds that $s' \in \mathcal{E}$. States in $\mathcal{E}$ are called *exhaustible*.

Informally, exhaustible states are those for which every logged position can be successfully tested, that is, the λIAM can backtrack to (an exhaustible state surrounding) it, if properly initialized. Roughly, a state is exhaustible if the backtracking information encoded in its logged positions is coherent. The set $\mathcal{E}$ being the *smallest* set of such states implies that checking that a state is exhaustible can be finitely certified, *i.e.* there must be a finitary proof.

Proposition 6.4 (Exhaustible invariant [Accattoli et al. 2020b]). *Let s be a λIAM reachable state. Then s is exhaustible.*

A key consequence is the fact that backtracking always succeeds, as it amounts to exhausting the first logged position on the log.

Lemma 6.5 (Backtracking always succeeds). *Let s a λIAM reachable state. If $s \rightarrow_{\mathrm{bt1},l} s'$ then there exists s'' such that $s' \rightarrow^*_{\lambda IAM} \rightarrow_{\mathrm{bt2},l} s''$.*

Proof. Consider $s = (t, \underline{C\langle u\langle\cdot\rangle\rangle}, l \cdot L, T) \rightarrow_{\mathrm{bt1},l} (\underline{u}, C\langle\langle\cdot\rangle t\rangle, L, l \cdot T) = s'$. Since s' is reachable then it is exhaustible, and so its tape test $s'_l := (\underline{u}, C\langle\langle\cdot\rangle t\rangle, L, l)$ can be exhausted, that is, there is a λIAM run $\pi : s'_l \rightarrow^*_{\lambda\mathrm{IAM}} \rightarrow_{\mathrm{bt2},l} q$ for a state q surrounding l. Note that s'_l is s' where the tape contains only l. Now, we lift π to a run $\pi^T : s' \rightarrow^*_{\lambda\mathrm{IAM}} \rightarrow_{\mathrm{bt2},l} s''$ using the tape lifting lemma (Lemma 3.3). □

7 RELATING THE λ-IAM AND THE λ-JAM: JUMPING IS EXHAUSTING

In this section we prove that the λJAM is a time optimization of the λIAM via an adaptation of the exhaustible invariant. Our proof is based on the construction of a bisimulation which also provides, as a corollary, the implementation theorem for the λJAM. The basic idea is that the two machines are equivalent *modulo backtracking*. Indeed, the λJAM evaluates terms as the λIAM, but for the backtracking phase, which is short-circuited and done with just one *jump* transition. Then one has to show that the *jump* is actually simulated by the λIAM.

Log Tests. For simulating jumps we need log tests. The idea is the same underlying tape tests: they focus on a given logged position in the log. Their definition however requires more than simply stripping down the log, as the new log and the position still have to form a logged position—said differently, the *position and the log* invariant (Lemma 3.1) has to be preserved. Roughly, the log test s_{l_m} focussing on the m-th logged position l_m in the log of a state $(t, C_n, l_n \cdots l_2 \cdot l_1, T, d)$ is obtained by removing the prefix $l_n \cdots l_{m+1}$ (if any), and moving the current position up by $n - m$ levels. Moreover, the tape is emptied and the direction is set to $\uparrow$.

In the argument for the simulation of jumps given below, we need only log tests of a very simple form. Namely, given a state $s = (t, \underline{C\langle u\langle\cdot\rangle\rangle}, l \cdot L, T)$ from which the λJAM jumps, we shall consider the log test $s_l := (t, \underline{C\langle u\langle\cdot\rangle\rangle}, l \cdot L, \epsilon)$, that is, the tape is emptied and (in this case) the position does not change. The more general form of log tests needing the position change is technical and defined in the technical report [Accattoli et al. 2020a]—it is unavoidable for proving the invariant, but we fear that giving it here would obfuscate the use of the exhaustible technique, whose idea is instead quite simple.

I-Exhaustible Invariant. The λIAM exhaustible invariant proves that backtracking phases always succeed, and it is the key ingredient to relate the λIAM and the λJAM. While the underlying idea is clear, there is an important detail that has to be addressed: to establish the simulation, we have to prove that the λIAM can exhaust logged positions *of the λJAM*, rather than its own.

Since the two machines use logs differently, we have to use a function $I(\cdot)$ that maps the log-related notions of the λJAM to those of the λIAM (where Γ ranges over both logs and tapes):

$$\begin{array}{rlll} \text{LOGGED POSITIONS} & I(x, C\langle\lambda x.D_n\rangle, L_n{\cdot}L) := (x, \lambda x.D_n, I(L_n)) & & \\ \text{TAPES AND LOGS} & I(\epsilon) := \epsilon & I(l{\cdot}\Gamma) := I(l){\cdot}I(\Gamma) & I(\bullet{\cdot}T) := \bullet{\cdot}I(T) \\ \text{STATES} & I(t, C, L, T, d) := (t, C, I(L), I(T), d) & & \end{array}$$

Another point is that the state surrounding the exhausted position now is uniquely determined by the logged position. Given a logged position $l = (x, D, L)$, the *state induced by l* is $l^\circ := (x, \underline{D}, L, \epsilon)$.

Definition 7.1 (I-Exhaustible States). $\mathcal{E}_I$ is the smallest set of λJAM states s such that for any tape or log test s_l of s of focus l, there exists a run $\pi : I(s_l) \rightarrow^*_{\lambda\text{IAM}}\rightarrow_{\text{bt2},I(l)} I(l^\circ)$ such that $l^\circ \in \mathcal{E}_I$. States in $\mathcal{E}_I$ are called *I-exhaustible.*

LEMMA 7.2 (I-EXHAUSTIBLE INVARIANT). *Let s be a λJAM reachable state. Then s is I-exhaustible.*

Jumping is Exhausting. From the invariant and the tape lifting property of the λIAM, it follows easily that jumps can be simulated via backtracking, from which the relationship between the λIAM and the λJAM immediately follows. We write $\rightarrow_{\text{jmp},l}$ for a $\rightarrow_{\text{jmp}}$ transition jumping to l.

LEMMA 7.3 (JUMPS SIMULATION VIA BACKTRACKING). *Let s be a λJAM reachable state such that $s \rightarrow_{\text{jmp},l} s'$. Then $I(s) \rightarrow_{\text{bt1},I(l)}\rightarrow^*_{\lambda IAM}\rightarrow_{\text{bt2},I(l)} I(s')$.*

PROOF. Let $l := (x, D, L')$ and consider $s = (t, \underline{C\langle u\langle\cdot\rangle\rangle}, (x, D, L') \cdot L, T) \rightarrow_{\text{jmp},l} (x, \underline{D}, L', T) = s'$. Since s is reachable then it is I-exhaustible, so its log test $s_l := (t, \underline{C\langle u\langle\cdot\rangle\rangle}, l \cdot L, \epsilon)$ can be exhausted, that is, there is a λIAM run $\pi : I(s_l) \rightarrow^*_{\lambda\text{IAM}}\rightarrow_{\text{bt2},I(l)} I(x, \underline{D}, L', \epsilon) = s''$. Note that the first transition of π is necessarily $\rightarrow_{\text{bt1},I(l)}$. Moreover, $I(s_l)$ and s'' are exactly $I(s)$ and $I(s')$ with empty tape. We lift π to a run $\pi^{I(T)} : I(s_l)^{I(T)} \rightarrow_{\text{bt1},I(l)}\rightarrow^*_{\lambda\text{IAM}}\rightarrow_{\text{bt2},I(l)} s''^{I(T)}$ using Lemma 3.3. Now, $\pi^{I(T)}$ is exactly the λIAM simulation of the jump, because $I(s_l)^{I(T)} = I(s)$ and $s''^{I(T)} = I(s)$. □

From the lemma it easily follows a bisimulation between the λIAM and the λJAM, showing that the latter is faster. In the technical report [Accattoli et al. 2020a], a general theorem relating also potentially diverging runs can be found. Here we give only the more concise statement about complete runs.

THEOREM 7.4 (λIAM AND λJAM RELATIONSHIP). *There is a complete λJAM run π_J from t if and only if there is a complete λIAM run π_I from t. In particular, the λJAM implements Closed CbN. Moreover, $|\pi_J| \leq |\pi_I|$ and $|\pi_J|_{\text{var}} \leq |\pi_I|_{\text{var}}$.*

Exponential Gap. The time gap between the λIAM and the λJAM can be exponential, as it is shown by the family of terms t_n ($t_1 := \mathsf{I}$ and $t_{n+1} := t_n\mathsf{I}$) mentioned in the introduction. The results of this paper provide a nice high-level proof. Next section shows that the time of the λJAM is polynomial in the time of the KAM, that takes time polynomial in the number of β-steps to evaluate

LOGGED CLOSURES	$\hat{c} ::= (t, C_{n+1}, E)^{L_n}$	ENVIRONMENTS	$E ::= \epsilon \mid [x{\leftarrow}\hat{c}] \cdot E$
LOGS	$L_0 ::= \epsilon \qquad L_{n+1} ::= \hat{l} \cdot L_n$	CLOSED POSITIONS	$\hat{l} ::= (t, C_n, L_n)^E$
CLOSED TAPES	$T ::= \epsilon \mid \hat{c} \cdot T \mid \hat{l} \cdot T$	STATES	$s ::= (t, C, L, E, T, d)$

Term	Ctx	Log	Env	Cl. Tape		Term	Ctx	Log	Env	Cl. Tape
$\underline{tu}$	C	L	E	T	$\rightarrow_{\bullet 1/\mathsf{app}}$	$\underline{t}$	$C\langle\langle\cdot\rangle u\rangle$	L	E	$\hat{c}\cdot T$
$\underline{\lambda x.t}$	C	L	E	$\hat{c}\cdot T$	$\rightarrow_{\bullet 2/\mathsf{abs}}$	$\underline{t}$	$C\langle\lambda x.\langle\cdot\rangle\rangle$	L	$[x{\leftarrow}\hat{c}]\cdot E$	T
$\underline{x}$	$C\langle\lambda x.D_n\rangle$	$L_n\cdot L$	$E'\cdot[x{\leftarrow}\hat{c}]\cdot E$	T	$\rightarrow_{\mathsf{varJ}}$	$\lambda x.D_n\langle x\rangle$	$\underline{C}$	L	E	$\hat{l}\cdot T$
$\underline{x}$	C	L	E	T	$\rightarrow_{\mathsf{hop/varK}}$	$\underline{u}$	D	$\hat{l}\cdot L'$	F	T

where $\hat{c} := (u, C\langle t\langle\cdot\rangle\rangle, E)^L$ in $\rightarrow_{\bullet 1/\mathsf{app}}$
$\hat{l} := (x, C\langle\lambda x.D\rangle, L_n\cdot L)^{E'\cdot[x\leftarrow\hat{c}]\cdot E}$ in $\rightarrow_{\mathsf{varJ}}$
$E = E'\cdot[x{\leftarrow}(u, D, F)^{L'}]\cdot E''$ and $\hat{l} = (x, C, L)^E$ in $\rightarrow_{\mathsf{hop/varK}}$.

Term	Ctx	Log	Env	Cl. Tape		Term	Ctx	Log	Env	Cl. Tape
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	L	E	$\hat{c}\cdot T$	$\rightarrow_{\bullet 3}$	tu	$\underline{C}$	L	E	T
t	$\underline{C\langle\lambda x.\langle\cdot\rangle\rangle}$	L	$[x{\leftarrow}\hat{c}]\cdot E$	T	$\rightarrow_{\bullet 4}$	$\lambda x.t$	$\underline{C}$	L	E	$\hat{c}\cdot T$
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	L	E	$\hat{l}\cdot T$	$\rightarrow_{\mathsf{arg}}$	$\underline{u}$	$C\langle t\langle\cdot\rangle\rangle$	$\hat{l}\cdot L$	E	T
t	$\underline{C\langle u\langle\cdot\rangle\rangle}$	$\hat{l}\cdot L$	E	T	$\rightarrow_{\mathsf{jmp}}$	x	$\underline{D}$	L'	E'	T

where in the last transition $\hat{l} = (x, D, L')^{E'}$.

Fig. 4. Data structures and transitions of the Hopping Abstract Machine (HAM).

t_n, that is, n. The study of multi types in Sect. 12 instead shows that the time of the λIAM depends on the size of the smallest type A_n of t_n, which is easily seen to be exponential in n. In fact, using the notation of Sect. 12, $A_1 := \star$, and $A_{n+1} := [A_n] \rightarrow A_n$.

8 ENTANGLING THE λJAM AND THE KAM: THE HAM

Now we turn to the relationship between the λJAM and the KAM. We prove that KAM runs can be obtained from λJAM ones via *hops* that short-circuit the search for arguments realized by the blue transitions. It then follows that the KAM can be seen as a time improvement of the λJAM.

The HAM. To prove that the KAM is a time improvement of the λJAM, we introduce an intermediate machine, the *Hopping Abstract Machine* (HAM) in Fig. 4, that merges the two. The HAM is a technical tool addressing an inherent difficulty: the λJAM and the KAM use different data structures and it is impossible to turn a KAM state into a λJAM state without having to look at the whole run that led to that state, as it is instead possible for the λJAM and the λIAM.

The idea behind the HAM is to entangle the data structures of both machines so that their states get paired by construction, and to allow it to behave non-deterministically either as the λJAM or as the KAM. The HAM deals with two enriched objects, *logged closures* $\hat{c}$ and *closed (logged) positions* $\hat{l}$ (defined in Fig. 4, overloading some of the notations of the previous sections), obtained by adding a log to closures and an environment to logged positions. Of course, environments and logs have to be redefined as containing these enriched objects. There is also a *(closed) tape* T, that is, a data structure obtained by merging the roles of the stack and the tape and containing both logged closures and closed positions. In fact the closed tape is obtained from the λJAM tape by upgrading every $\bullet$ entry to a logged closure $\hat{c}$, and every logged position l to a closed one $\hat{l}$. Logged closures and closed positions contain the same information (a term, a context, a log, and an environment) but they play different roles.

The non-determinism of the machine amounts to the presence of *two* transitions $\rightarrow_{\mathsf{varJ}}$ and $\rightarrow_{\mathsf{hop/varK}}$ for the variable case, that are simply the var transitions of the λJAM and the KAM, lifted to the new data structures. In particular, transition $\rightarrow_{\mathsf{hop/varK}}$ short-circuits a whole $\uparrow$ phase of the λJAM *hopping* directly to the argument.

It is evident that by removing environments, turning every logged closure into $\bullet$, and removing $\rightarrow_{\mathsf{hop/varK}}$ we obtain the λJAM. Similarly, by removing logs, $\rightarrow_{\mathsf{varJ}}$, and the $\uparrow$ transitions, one obtains the KAM. We avoid spelling out these immediate projections. Instead, we see KAM runs inside the HAM as given by the transition $\rightarrow_{\mathsf{HAM_K}} := \rightarrow_{\bullet 1/\mathsf{app}} \cup \rightarrow_{\bullet 2/\mathsf{abs}} \cup \rightarrow_{\mathsf{hop/varK}}$. Similarly, the λJAM is seen as transition $\rightarrow_{\mathsf{HAM_J}}$, defined as the union of all HAM transitions but $\rightarrow_{\mathsf{hop/varK}}$.

The HAM verifies the same basic properties of the λJAM, simply lifted to the enriched data structures. Moreover, it verifies a tape lifting property.

Lemma 8.1 (HAM tape lift). *Let $\pi : s = (t, C, L, E, T', d) \rightarrow^n_{HAM} (u, D, L', E', T'', d') = s'$ be a run and T be a tape. Then there is a run $\pi^T : s^T = (t, C, L, E, T' \cdot T, d) \rightarrow^n_{HAM} (u, D, L', E', T'' \cdot T, d') = s'^T$.*

9 HOPPING IS ALSO EXHAUSTING

Since jumping and hopping amount to a similar idea, the proof technique that we use to relate the λJAM and the KAM is obtained by another variation on the exhaustible invariant.

Testing Logged Closures. The main difference is that now we exhaust *logged closures* instead of logged positions. Via the $\uparrow$-exhaustible invariant below we shall show that the HAM can exhaust a logged closure—that is it can recover the argument in the closure—by using only λJAM $\uparrow$ transitions. This capability shall then be used to show that the λJAM can simulate hops.

Since logged closures are both in the environment and in the tape, we have two kinds of test. The definition of tape tests is in the technical report [Accattoli et al. 2020a]. They are essential for the proof of the $\uparrow$-exhaustible invariant, but they are not needed for the argument at work in the simulation, spelled out below.

Environment Tests. Given a HAM state (t, C, L, E, T, d) consider an entry $[x \leftarrow \hat{c}]$ in E. The idea is that one wants to exhaust $\hat{c}$ to return to the state saved in $\hat{c}$. Remember that the λJAM looks for the argument starting from the binder of x. Then, the test associated to $\hat{c}$ is obtained by positioning the machine on the binder λx for x, and modifying the log and the environment accordingly. Moreover, the tape is emptied.

Definition 9.1 (HAM environment tests). Let $s = (t, C\langle \lambda x.D_n \rangle, L_n \cdot L, E' \cdot [x \leftarrow \hat{c}] \cdot E, T, d)$ be a state. Then, $s_{\hat{c}} := (\lambda x.D_n\langle t \rangle, \underline{C}, L, E, \epsilon)$ is an environment test for s of focus $\hat{c}$.

As in the previous section, we need a notion of state induced by a logged closure $\hat{c}$, that is the state reached by a run exhausting $\hat{c}$. The definition may seem wrong, an explanation follows.

Definition 9.2 (HAM state induced by a logged closure). Given a logged closure $\hat{c} = (u, D\langle t\langle \cdot \rangle \rangle, E)^L$, the state $\hat{c}^\circ$ induced by $\hat{c}$ is defined as $\hat{c}^\circ := (t, \underline{D\langle \langle \cdot \rangle u \rangle}, L, E, \epsilon)$.

The previous definition is counter-intuitive, as one would expect $\hat{c}^\circ$ to rather be the state $s' := (\underline{u}, D\langle t\langle \cdot \rangle \rangle, L, E, \epsilon)$, but for technical reasons this is not possible. In the simulations of hops below, however, $\hat{c}^\circ$ is tape lifted to a state that makes a $\rightarrow_{\mathsf{arg}}$ transition to (a tape lifting of) s', as one would expect. We set $\rightarrow_{\uparrow} := \rightarrow_{\bullet 3, \bullet 4, \mathsf{arg}, \mathsf{jmp}}$.

Definition 9.3 (HAM $\uparrow$-Exhaustible states). $\mathcal{E}_{\uparrow}$ is the smallest set of those states s such that for any tape or environment test $s_{\hat{c}}$ of s, there exists a run $\pi_{\uparrow} : s_{\hat{c}} \rightarrow^*_{\uparrow} \hat{c}^\circ$ and $\hat{c}^\circ \in \mathcal{E}_{\uparrow}$. States in $\mathcal{E}_{\uparrow}$ are called $\uparrow$*-exhaustible* (pronounced *up*-exhaustible).

Lemma 9.4. *Let s be a HAM reachable state. Then s is $\uparrow$-exhaustible.*

Simulating Hops. From the invariant and the tape lifting property of the HAM, it follows easily that hops can be simulated via $\rightarrow_{\uparrow}$, as the next lemma shows.

Lemma 9.5 (Hops simulation via $\uparrow$). *Let s be a HAM reachable state and $s \rightarrow_{\mathsf{hop/varK}} s'$. Then $s \rightarrow_{\mathsf{varJ}} \rightarrow_{\uparrow}^{*} \rightarrow_{\mathsf{arg}} s'$.*

Proof. The hypothesis is: $s = (\underline{x}, C, L, E, T) \rightarrow_{\mathsf{hop/varK}} (\underline{u}, D\langle t\langle\cdot\rangle\rangle, \hat{l}\cdot L', F, T) = s'$ where $E = E'\cdot[x{\leftarrow}\hat{c}]\cdot E''$ with $\hat{c} = (u, D\langle t\langle\cdot\rangle\rangle, F)^{L'}$ and $\hat{l} := (x, C, L)^{E}$. From s the HAM can also do a $\rightarrow_{\mathsf{varJ}}$ transition: $s = (\underline{x}, C'\langle \lambda x.D'_n\rangle, L_n\cdot L'', E'[x{\leftarrow}\hat{c}]E'', T) \rightarrow_{\mathsf{varJ}} (\lambda x.D'_n\langle x\rangle, \underline{C'}, L'', E'', \hat{l}\cdot T) =: s''$ where $L = L_n\cdot L''$ and $C = C'\langle\lambda x.D'_n\rangle$. Now consider the environment test $s'_{\hat{c}} = (\lambda x.D_n\langle x\rangle, \underline{C}, L'', E'', \epsilon)$. By $\uparrow$-exhaustibility we obtain $\pi : s'_{\hat{c}} \rightarrow_{\uparrow}^{*} \hat{c}^{\circ} = (t, \underline{D\langle\langle\cdot\rangle u\rangle}, L', F, \epsilon)$ Then, lifting $\hat{c}^{\circ}$ with the tape $\hat{l}\cdot T$, one has $\hat{c}^{\circ}_{\hat{l}\cdot T} = (t, \underline{D\langle\langle\cdot\rangle u\rangle}, L', F, \hat{l}\cdot T) \rightarrow_{\mathsf{arg}} (\underline{u}, D\langle t\langle\cdot\rangle\rangle, \hat{l}\cdot L', F, T)$ Thus, $s \rightarrow_{\mathsf{varJ}} s'' \rightarrow_{\uparrow}^{*} \hat{c}^{\circ}_{\hat{l}\cdot T} \rightarrow_{\mathsf{arg}} s'$. □

From the lemma it easily follows a bisimulation between the λJAM and the KAM, showing that the latter is faster. In the technical report [Accattoli et al. 2020a], there is a theorem relating their runs inside the HAM, considering also potentially diverging runs. Here we give only the more concise statement about complete runs.

Theorem 9.6 (λJAM and KAM relationship). *There is a complete λJAM run π_J from t if and only if there is a complete KAM run π_K from t. Moreover, $|\pi_J| = |\pi_K| + |\pi_J|_{\uparrow}$ and $|\pi_J|_{\mathsf{var}} = |\pi_K|_{\mathsf{var}}$.*

10 THE λ-JAM IS SLOWLY REASONABLE

In this section we provide bounds for the complexity of the λJAM. First, we show that it is quadratically slower than the KAM, and then, by using results from the literature about the KAM, we obtain bounds with respect to the two parameters for complexity analyses of abstract machines, namely, the size $|t|$ of the evaluated term and the number $\#\beta$ of $\rightarrow_{wh}$-steps to evaluate t.

Locating the λJAM. We have proved in the previous two sections that a run π_J of the λJAM from t is such that $|\pi_K| \leq |\pi_J| \leq |\pi_I|$, where π_K and π_I are the runs from t respectively of the KAM and of the λIAM. However, this tells nothing about the inherent complexity of evaluating a term with the λJAM. In fact, it is well known that $|\pi_K|$ is polynomial in $\#\beta$ and $|t|$ (namely quadratic in $\#\beta$ and linear in $|t|$), while $|\pi_I|$ can be exponential in both $\#\beta$ and $|t|$ (the typical example of exponential behavior being the family of terms t_n defined as $t_1 := \mathsf{I}$ and $t_{n+1} := t_n\mathsf{I}$). What about the λJAM? Is it polynomial or exponential? It turns out that the λJAM is polynomial, and precisely at most quadratically slower than the KAM.

Bounding $\uparrow$ Phases. Since the KAM is the λJAM less the (blue) $\uparrow$ phases, and the complexity of the KAM is known, we only have to study the length of $\uparrow$ phases. The length of a $\uparrow$ phase extending a run π from t is bound by $|\pi|_{\mathsf{var}} \cdot |t|$, and the length of all $\uparrow$ phases together is bound by $|\pi|^2_{\mathsf{var}} \cdot |t|$. The proof is in three steps. First, we show that in absence of jumps a $\uparrow$ phase cannot be longer than $|t|$. An immediate induction on $|C|$ proves the following lemma.

Lemma 10.1. *Let $\pi : (t, \underline{C}, L, T) \rightarrow_{\bullet 3, \bullet 4}^{*} s$. Then $|\pi| \leq |C| \leq |C\langle t\rangle|$.*

Second, we need an invariant. To estimate the number of jumps in a $\uparrow$ phase, we need to link the structure of logs with the number of $\rightarrow_{\mathsf{var}}$ transitions encountered so far. We introduce the notion of *depth* of a tape/log Γ, defined in the following way:

$$\begin{array}{rcl@{\qquad}rcl}
\mathsf{depth}(\epsilon) & := & 0 & \mathsf{depth}(\bullet\cdot T) & := & \mathsf{depth}(T) \\
\mathsf{depth}(l\cdot\Gamma) & := & \mathsf{depth}(l) & \mathsf{depth}((x, C, L)) & := & 1 + \mathsf{depth}(L) \\
\mathsf{depth}(t, C, L, T, \uparrow) & := & \mathsf{depth}(T) & \mathsf{depth}(t, C, L, T, \downarrow) & := & \mathsf{depth}(L)
\end{array}$$

Proposition 10.2 (Depth invariant). *Let $\pi : s_t \to^*_{\lambda JAM} s$ be an initial run of the λJAM. Then* $\mathsf{depth}(s) = |\pi|_{\mathsf{var}}$. *Moreover* $\mathsf{depth}(s) \geq \mathsf{depth}(l)$ *for every logged position l in s.*

Third, we bound $\uparrow$ phases. The number of jumps in a single phase $s \to^*_{\uparrow} s'$ of $\uparrow$ transitions is bound by $\mathsf{depth}(s)$, and pairing it up with Lemma 10.1 we obtain a bound on the phase. By the depth invariant the bound can be given relatively to $|\pi|_{\mathsf{var}}$, and a standard argument extends the bound to all $\uparrow$ phases in a run, adding a quadratic dependency. Let $|\pi|_{\uparrow}$ be the number of $\to_{\uparrow}$ transitions in π.

Lemma 10.3 (Bound on $\uparrow$ phases).

(1) One $\uparrow$ phase*: if $s = (t, \underline{C}, L, T)$ is a reachable state and $\pi : s \to^*_{\uparrow} s'$ then $|\pi| \leq$* $\mathsf{depth}(s) \cdot |C\langle t\rangle|$.
(2) All $\uparrow$ phases*: if $\pi : s_t \to^*_{\lambda JAM} s$ then $|\pi|_{\uparrow} \leq |\pi|^2_{\mathsf{var}} \cdot |t|$.*

The Complexity of the KAM. We need to recall the complexity analysis of the KAM from [Accattoli et al. 2014; Accattoli and Barras 2017; Accattoli and Dal Lago 2012]. The length of a complete KAM run π from t verifies $|\pi| = |\pi|_{\mathsf{var}} + 2 \cdot |\pi|_{\mathsf{abs}}$ and we have $|\pi|_{\mathsf{var}} = O(|\pi|^2_{\mathsf{abs}})$ (the bound is tight, as there are examples reaching it). Since $|\pi|_{\mathsf{abs}}$ is exactly the number $\#\beta$ of $\to_{wh}$ steps to evaluate t (the cost model of reference for Closed CbN), we have that $|\pi| = O(\#\beta^2)$. Now since the cost of implementing KAM single transitions on RAM is bound by $|t|$, the complexity of implementing the KAM is $O(\#\beta^2 \cdot |t|)$, that is, the KAM is a reasonable machine.

The Complexity of the λJAM. From the complexity of the KAM, the fact that the λJAM and the KAM do exactly the same number of $\to_{\mathsf{var}}$ transitions, and that the number of $\uparrow$ transition of the λJAM are bound by $|\pi|^2_{\mathsf{var}} \cdot |t|$, we easily obtain the following results.

Theorem 10.4 (λJAM complexity). *Let t be a closed term such that $t \to^n_{wh} u$, u be $\to_{wh}$ normal, and π_J and π_K be the complete λJAM and KAM runs from t. Then:*

(1) The λJAM is quadratically slower than the KAM*:* $|\pi_K| \leq |\pi_J| = O(|\pi_K|^2 \cdot |t|)$.
(2) The λJAM is (slowly) reasonable*: $|\pi_J| = O(n^4 \cdot |t|)$, and the cost of implementing π_J on a RAM is also $O(n^4 \cdot |t|)$.*

11 THE POINTER ABSTRACT MACHINE, REVISITED

The Pointer Abstract Machine (PAM), due to Danos and Regnier [Danos et al. 1996; Danos and Regnier 2004], gives an operational account of the interaction process at work in Hyland and Ong game semantics. The machine is always described rather informally via a pseudo-code algorithm. Here we define it according to our syntactic style, calling it λPAM, and provide its first formal and manageable presentation as an actual abstract machine.

Our result concerning the λPAM is that it is strongly bisimilar to the λJAM. Roughly, the two are the same machine, with exactly the same time behavior, they just use different data structures. This connection is mentioned in [Danos and Regnier 1999], but not proved. We find it instructive to spell it out, as the connection is elegant but far from being evident.

Fragmented vs Monolitic Run Traces. Both machines jump and need to store information about the run, to jump to the right place. They differ on how they represent this information. The λJAM uses logged positions, that is, positions coming with the information to be restored after the jump. The approach can be seen as *fragmented*, as the trace of the run is distributed among all the logged positions in the state. The λPAM adopts a *monolitic* approach, storing all the information in a unique *history* H, a new data structure encoding the whole run in a minimalistic and sophisticated way. Roughly, the history H saves all the variable positions p for which an argument as been found, each one with a pointer (under the form of an index i) to a previous variable position p' in H. The index i intuitively realizes a mechanism to retrieve the log associated to p by the λJAM. We first define the machine and then explain the relationship between the two approaches.

Data Structures. All the data structures of the PAM are defined in Fig. 5. Positions are no longer logged, and noted with p, p', etc. An *index* i is simply a natural number. *Indexed positions* are pairs (p, i). A history H is a sequence of indexed variable positions (accumulated from right to left). The idea is that indices are pointers to entries in the history, that is, if the i-th entry of H is (p, j) then j points to a previous entry in H, that is, $j < i$. The tape of the λPAM is a stack containing variable positions and occurrences of $\bullet$.

Transitions and Look-Up. Initial states have the form $s_t := (\underline{t}, \langle\cdot\rangle, \epsilon, 0, \epsilon)$, the transitions of the λPAM are in Fig. 5, they are labeled exactly as in the λJAM, and their union is noted $\rightarrow_{\lambda\mathsf{PAM}}$. Transitions $\rightarrow_{\mathsf{var}}$ and $\rightarrow_{\mathsf{jmp}}$ need to retrieve information from the history H, for which there are some dedicated notations. We use i^H_k, x^H_k, D^H_k to denote, respectively, the index, variable, and context of the k-th indexed position in H.

Transition $\rightarrow_{\mathsf{var}}$ moreover looks up into H in an unusual way. The idea is that it accesses H n times to retrieve an index. The first time it retrieves the indexed position (p_1, j_1) of index i, to then retrieve the position (p_2, j_2) of index j_1, and so on, until it retrieves j_n and makes it the new state index. This is formalized using the look-up function $\phi_H : \mathbb{N} \to \mathbb{N}$ defined as $\phi_H(k) := i^H_k$, and whose powers ϕ^n_H are defined as $\phi^n_H(k) := \phi_H(\phi^{(n-1)}_H(k))$, where $\phi^0_H(k) := k$. Note that implementing $\rightarrow_{\mathsf{var}}$ on RAM then costs n, that is bound by the size $|t|$ of the initial term, exactly as for the λJAM, while all other transitions have constant cost.

An Example. As for the other machines we have considered in this paper, we give the execution trace of the λPAM on the term $(\lambda y.\lambda x.xy)\mathsf{I}\mathsf{I}$. The reader can grasp some intuition considering that the PAM is strongly bisimilar to the λJAM. In particular, the λPAM considers explicit pointers. Indeed, as we have already pointed out, λJAM logs are not actually copied in the λJAM $\rightarrow_{\mathsf{var}}$ transition: what is duplicated is just a pointer to them. The λPAM handles this mechanism directly in its definition, and can thus be considered as a low-level implementation of the λJAM. In the following we will explain this in more detail. After having looked for the head variable through the spine of the term, the λPAM, now in $\uparrow$ mode, queries the argument of x, namely $\lambda z.z$, that then explores. The argument of its head variable z is y, that has to be found via *backtracking* or *jumping*. We set $p_x := (x, (\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{I}(\lambda z.z))$.

	Sub-term	Context	Hist.	Index	Tape	Dir
	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}\mathsf{I}}$	$\langle\cdot\rangle$	ϵ	0	ϵ	$\downarrow$
$\rightarrow_{\bullet 1}$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}}$	$\langle\cdot\rangle\mathsf{I}$	ϵ	0	$\bullet$	$\downarrow$
$\rightarrow_{\bullet 1}$	$\underline{\lambda y.\lambda x.xy}$	$\langle\cdot\rangle\mathsf{I}\mathsf{I}$	ϵ	0	$\bullet\cdot\bullet$	$\downarrow$
$\rightarrow_{\bullet 2}$	$\underline{\lambda x.xy}$	$(\lambda y.\langle\cdot\rangle)\mathsf{I}\mathsf{I}$	ϵ	0	$\bullet$	$\downarrow$
$\rightarrow_{\bullet 2}$	$\underline{xy}$	$(\lambda y.\lambda x.\langle\cdot\rangle)\mathsf{I}\mathsf{I}$	ϵ	0	ϵ	$\downarrow$
$\rightarrow_{\bullet 1}$	$\underline{x}$	$(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{I}\mathsf{I}$	ϵ	0	$\bullet$	$\downarrow$

	Sub-term	Context	Hist.	Index	Tape	Dir
	$\underline{x}$	$(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{I}(\lambda z.z)$	ϵ	0	$\bullet$	$\downarrow$
$\rightarrow_{\mathsf{var}}$	$\lambda x.xy$	$\underline{(\lambda y.\langle\cdot\rangle)\mathsf{I}(\lambda z.z)}$	ϵ	0	$p_x\cdot\bullet$	$\uparrow$
$\rightarrow_{\bullet 4}$	$\lambda y.\lambda x.xy$	$\underline{\langle\cdot\rangle\mathsf{I}(\lambda z.z)}$	ϵ	0	$\bullet\cdot p_x\cdot\bullet$	$\uparrow$
$\rightarrow_{\bullet 3}$	$(\lambda y.\lambda x.xy)\mathsf{I}$	$\underline{\langle\cdot\rangle(\lambda z.z)}$	ϵ	0	$p_x\cdot\bullet$	$\uparrow$
$\rightarrow_{\mathsf{arg}}$	$\underline{(\lambda z.z)}$	$(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle$	$(p_x, 0)$	1	$\bullet$	$\downarrow$
$\rightarrow_{\bullet 2}$	$\underline{z}$	$(\lambda y.\lambda x.xy)\mathsf{I}(\lambda z.\langle\cdot\rangle)$	$(p_x, 0)$	1	ϵ	$\downarrow$
$\rightarrow_{\mathsf{var}}$	$\lambda z.z$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle}$	$(p_x, 0)$	1	$(z, (\lambda y.\lambda x.xy)\mathsf{I}(\lambda z.\langle\cdot\rangle))$	$\uparrow$

The jump is simulated by the λPAM retrieving the position saved in the history at the current index, and then updating the index accordingly, *i.e.* diminishing it by one. Intuitively, this corresponds to the 'unpacking' made by the λJAM in the $\rightarrow_{\mathsf{jmp}}$ transition. We set $p_z := (z, (\lambda y.\lambda x.xy)\mathsf{I}(\lambda z.\langle\cdot\rangle))$ and $p_y = (y, (\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I})$.

POSITIONS	$p ::= (t, C)$	TAPES	$T ::= \epsilon \mid \bullet\cdot T \mid p\cdot T$
HISTORIES	$H ::= \epsilon \mid (p, i)\cdot H$	STATES	$s ::= (t, C, H, i, T, d)$ $\quad i \in \mathbb{N}$

Sub-term	Context	Hist.	Index	Tape		Sub-term	Context	Hist.	Index	Tape
$\underline{tu}$	C	H	i	T	$\to_{\bullet 1}$	$\underline{t}$	$C\langle\langle\cdot\rangle u\rangle$	H	i	$\bullet\cdot T$
$\underline{\lambda x.t}$	C	H	i	$\bullet\cdot T$	$\to_{\bullet 2}$	$\underline{t}$	$C\langle\lambda x.\langle\cdot\rangle\rangle$	H	i	T
$\underline{x}$	$C\langle\lambda x.D_n\rangle$	H	i	T	$\to_{\text{var}}$	$\lambda x.D_n\langle x\rangle$	$\underline{C}$	H	$\phi^n_H(i)$	$(x, C\langle\lambda x.D_n\rangle)\cdot T$
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	H	i	$\bullet\cdot T$	$\to_{\bullet 3}$	tu	$\underline{C}$	H	i	T
t	$\underline{C\langle\lambda x.\langle\cdot\rangle\rangle}$	H	i	T	$\to_{\bullet 4}$	$\lambda x.t$	$\underline{C}$	H	i	$\bullet\cdot T$
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	H	i	$p\cdot T$	$\to_{\text{arg}}$	$\underline{u}$	$C\langle t\langle\cdot\rangle\rangle$	$(p, i)\cdot H$	$\lvert H\rvert + 1$	T
t	$\underline{C\langle u\langle\cdot\rangle\rangle}$	H	i	T	$\to_{\text{jmp}}$	x^H_i	$\underline{D^H_i}$	H	$i - 1$	T

Fig. 5. Data structures and transitions of the λ Pointer Abstract Machine (λPAM).

	Sub-term	Context	Hist.	Index	Tape	Dir
	$\lambda z.z$	$\underline{(\lambda y.\lambda x.xy)\mathsf{I}\langle\cdot\rangle}$	$(p_x, 0)$	1	p_z	$\uparrow$
$\to_{\text{jmp}}$	x	$\underline{(\lambda y.\lambda x.\langle\cdot\rangle y)\mathsf{I}(\lambda z.z)}$	$(p_x, 0)$	0	p_z	$\uparrow$
$\to_{\text{arg}}$	$\underline{y}$	$(\lambda y.\lambda x.x\langle\cdot\rangle)\mathsf{I}\mathsf{I}$	$(p_z, 0)\cdot(p_x, 0)$	2	ϵ	$\downarrow$
$\to_{\text{var}}$	$\lambda y.\lambda x.xy$	$\underline{\langle\cdot\rangle\mathsf{I}\mathsf{I}}$	$(p_y, 0)\cdot(p_z, 0)\cdot(p_x, 0)$	0	p_y	$\uparrow$
$\to_{\text{arg}}$	$\underline{\mathsf{I}}$	$(\lambda y.\lambda x.xy)\langle\cdot\rangle\mathsf{I}$	$(p_y, 0)\cdot(p_z, 0)\cdot(p_x, 0)$	3	ϵ	$\downarrow$

As for the other machines, the technical report [Accattoli et al. 2020a] contains an example showing how the λPAM deals with duplication.

Final States and Invariants. Final states of the λPAM have, as expected, shape $(\underline{\lambda x.t}, C, H, i, \epsilon, \downarrow)$. This follows from the fact that the machine is never stuck on $\to_{\text{var}}$ steps because $\phi^n_H(i)$ is undefined. Note indeed a subtle point: $\phi_H(0)$ is undefined, so, potentially, $\phi^n_H(i)$ may be undefined. We then need an invariant ensuring that—in the source state of $\to_{\text{var}}$— $\phi^n_H(i)$ is always defined. The next statement collects also other minor invariants of the λPAM.

We say that H *has depth* (at least) $n \in \mathbb{N}$ at i if $n = 0$ or if $n > 0$ and $\phi^m_H(i) > 0$ for every $m < n$.

LEMMA 11.1 (λPAM INVARIANTS). *Let $s = (t, C_n, H, i, T, d)$ be a reachable PAM state. Then:*

(1) Depth*: H has depth n at i. Moreover, if $((u, D_m), j)$ is the k-th indexed position of H, with $k > 0$, then H has depth m at $k - 1$.*
(2) Tape, index, and direction*: if $d = \downarrow$, then $i = \lvert H\rvert$ and T does not contain any logged position, otherwise if $d = \uparrow$ then T contains exactly one position.*

History, Indices, and Logs. Let's now explain the relationship between the λJAM and the λPAM. The history H essentially stores the sequence of $\to_{\text{var}}$ queries, consisting of the position of a variable needing an argument, that the λPAM has completed, that is, for which it has found the argument. The key point is that it stores them with an index i. Indices are a low-level mechanism to retrieve logs, that are crumbled and shuffled all over H.

Let us explain how a log $(p_1, L_1)\cdot \ldots \cdot(p_n, L_n)$ of a reachable λJAM state is represented by the index i_1 and the history H of the corresponding λPAM state. There are two ideas:

- *The sequence of positions*: p_1 is in the i_1-th entry (p_1, i_2) of H, p_2 is in the i_2-th entry (p_2, i_3), and so on.
- *The log of each position*: the log L_1 of p_1 is represented in H (recursively following the same principle) starting from index $i_1 - 1$, the log L_2 starting from index $i_2 - 1$, and so on.

The Bisimulation. The given explanation underlies the following definition of relations $\simeq_T$,$\simeq_{LH}$ and $\simeq$ between data structures and states of the λJAM and the λPAM, that induce a strong bisimulation. The intended meaning of the relation $L \simeq_{LH} (H, i)$ is that the log L is represented in the history H starting from index i.

Definition 11.2. The relations $\simeq_T$, $\simeq_{LH}$ and $\simeq$ are defined as follows.

$$\textsc{Tapes} \quad \frac{}{\epsilon \simeq_T \epsilon} \quad \frac{T_J \simeq_T T_P}{\bullet{\cdot}T_J \simeq_T \bullet{\cdot}T_P} \quad \frac{T_J \simeq_T T_P}{(x, C, L){\cdot}T_J \simeq_T (x, C){\cdot}T_P}$$

$$\textsc{Log-Histories} \quad \frac{}{\epsilon \simeq_{LH} (H, 0)} \quad \frac{(x, C) = (x^H_i, D^H_i) \qquad L \simeq_{LH} (H, \phi_H(i)) \qquad L' \simeq_{LH} (H, i-1)}{(x, C, L'){\cdot}L \simeq_{LH} (H, i)}$$

$$\textsc{States} \quad \frac{T_J \simeq_T T_P \qquad L \simeq_{LH} (H, i)}{(t, C, L, T_J, d) \simeq (t, C, H, i, T_P, d)}$$

Note that in the second rule for $\simeq_{LH}$ the index i is ≥ 1, and that $\simeq$ contains all pairs of initial states. Note also that the (logged) positions case of $\simeq_T$ (rightmost rule for $\simeq_T$) the log L has no matching construct on the λPAM side. This is why the next theorem is stated together with an invariant (the *moreover* part), allowing to retrieve that log from the history.

Theorem 11.3 ($\simeq$ is a strong bisimulation).

(1) for every run $\pi_J : s^{\lambda JAM}_t \to^*_{\lambda JAM} s_J$ there exists a run $\pi_P : s^{\lambda PAM}_t \to^*_{\lambda PAM} s_P$ such that $s_J \simeq s_P$ and $|\pi_J| = |\pi_P|$ and performing exactly the same transitions;

(2) for every run $\pi_P : s^{\lambda PAM}_t \to^*_{\lambda PAM} s_P$ there exists a run $\pi_J : s^{\lambda JAM}_t \to^*_{\lambda JAM} s_J$ such that $s_J \simeq s_P$ and $|\pi_J| = |\pi_P|$ and performing exactly the same transitions.

Moreover, if $s_J = (t, \underline{C}, L, T_J, \uparrow) \simeq (t, \underline{C}, H, i, T_P, \uparrow) = s_P$ and (x, D, L') is the unique logged position in T_J then $L' \simeq_{LH} (H, |H|)$.

Strong bisimulations trivially preserve termination and the length of runs.

Corollary 11.4 (Termination and λPAM implementation). *$\lambda JAM(t)\Downarrow$ if and only if $\lambda PAM(t)\Downarrow$, and the two runs use exactly the same transitions. Therefore, the λPAM implements Closed CbN.*

12 SEQUENCE TYPES

Here we introduce a type system that we shall use to measure the length of λIAM runs.

Intersections, Multi Sets, and Sequences. The framework that we adopt is the one of intersection types. As many recent works, we use the non-idempotent variant, where the intersection type $A \wedge A$ is not equivalent to A, and which has stronger ties to linear logic and time analyses, because it takes into account how many times a resource/type A is used, and not just whether A is used or not. Non-idempotent intersections are multi sets, which is why these types are sometimes called *multi types* and an intersection $A \wedge B \wedge A$ is rather noted $[A, B, A]$. Here we add a further change, we also consider *non-commutative* multi types. Removing commutativity turns multi sets into lists, or sequences—thus, we call them *sequence types*. Adopting sequences is an inessential tweak. Our study does not really depend on their sequential structure, we only need to use bijections between multi sets, to describe the SIAM, and these bijections are just more easily managed using sequences rather than multi sets. This *rigid* approach has been already used fruitfully by Tsukada et al. [2017] and Mazza et al. [2018].

Basic Definitions. As for multi types, there are two layers of types, *linear types* and *sequence types*, mutually defined as follows.

$$\frac{}{x : [A] \vdash x : A}\ \text{T-Var} \qquad \frac{\Gamma, x : S \vdash t : A}{\Gamma \vdash \lambda x.t : S \multimap A}\ \text{T-}\lambda \qquad \frac{}{\Gamma \vdash \lambda x.t : \star}\ \text{T-}\lambda_\star$$

$$\frac{\Gamma \vdash t : [A'_1, \ldots, A'_n] \multimap A \quad [\Delta_i \vdash u : A'_i]_{i \in [1,\ldots,n]}}{\Gamma \uplus \sum_{i \in [1,\ldots,n]} \Delta_i \vdash tu : A}\ \text{T-@}$$

Fig. 6. The sequence type system.

$$\text{Linear types} \quad A, A' \ ::= \ \star \mid S \multimap A \qquad \text{Sequence types} \quad S, S' \ ::= \ [A_1, \ldots, A_n]$$

Since commutativity is ruled out, we have, e.g., $[A, A'] \neq [A', A]$. We shall use $[\cdot]$ as a generic list constructor not limited to types, thus writing $[2, 1, 12, 4]$ for a list of natural numbers, and also use it for lists of judgments or type derivations. Note that there is a ground type $\star$, which can be thought as the type of normal forms, that in Closed CbN are precisely abstractions. Note also that arrow (linear) types $S \multimap A$ can have a sequence only on the left. The empty sequence is noted $[]$, and the concatenation of two sequences S and S' is noted $S \uplus S'$.

Type judgments have the form $\Gamma \vdash t : A$, where Γ is a type environment, defined below. The typing rules are in Fig. 6, type derivations are noted π and we write $\pi \triangleright \Gamma \vdash t : A$ for a type derivation π of ending judgment $\Gamma \vdash t : A$. Type environments, ranged over by Γ, Δ are total maps from variables to sequence types such that only finitely many variables are mapped to non-empty sequence types, and we write $\Gamma = x_1 : S_1, \ldots, x_n : S_n$ if $dom(\Gamma) = \{x_1, \ldots, x_n\}$—note that type environments are commutative, what is non-commutative is only the sequence constructor $[\cdot]$. Given two type environments Γ, Δ, the expression $\Gamma \uplus \Delta$ stands for the type environment assigning to every variable x the list $\Gamma(x) \uplus \Delta(x)$. A sequence $\Gamma_{i_1}, \ldots, \Gamma_{i_k}$ of type environments is also noted $\{\Gamma_i\}_{i \in [i_1, \ldots, i_k]}$, or $\{\Gamma_i\}_{i \in I}$ with $I = [i_1, \ldots, i_k]$. Moreover, we use $\sum_{i \in I} \Gamma_i$ for the type environment defined as $\sum_{i \in I} \Gamma_i := []$ if $I = []$, and $\sum_{i \in I} \Gamma_i := \Gamma_{i_1} \uplus \sum_{i \in I'} \Gamma_i$ if $I = [i_1] \uplus I'$.

In the following we use two basic properties of the type system, collected in the following straightforward lemma. One is the absence of weakening, and the other one is a correspondence between sequence types and axioms. We write $|S|$ for the length of S as a sequence.

Lemma 12.1 (Relevance and axiom sequences). *If $\pi \triangleright \Gamma \vdash t : A$ then $dom(\Gamma) \subseteq \mathsf{fv}(t)$, thus if t is closed then Γ is empty. Moreover, there are exactly $|\Gamma(x)|$ axioms typing x in π, which appear from left to right as leaves of π (seen as an ordered tree) in the order given by $\Gamma(x) = [A_1, \ldots, A_k]$ and that the i-th axiom types x with A_i.*

Characterization of Termination. It is well-known that intersection and multi types characterize Closed CbN termination, that is, they type *all* and only those λ-terms that terminate with respect to weak head reduction. If terms are closed, the same result smoothly holds for sequence types, as we now explain. The only point where non-commutativity is delicate for the characterization is in the proof of the typed substitution lemma for subject reduction (and the dual lemma for subject expansion), as substitution may change the order of concatenation in type environments. In our simple setting where terms are closed, however, the term to substitute is closed[12] and—by the relevance lemma—its type derivation comes with no type environment, so the order-of-concatenation problem disappears. Therefore, sequence types characterize termination in Closed CbN too. Thus from now on we essentially identify multi and sequence types.

Theorem 12.2. *A closed term t has weak head normal form if and only if $\vdash t : \star$.*

[12]It is well known that in Closed CbN the substitutions $t\{x \leftarrow u\}$ associated to reduced β-redexes are such that u is closed. The term t is of course (potentially) open, and its type derivation has a type environment Γ, but the important point here is that the type derivation of u has no type environment, so that the substitution does not concatenate sequence types.

KAM

$$\frac{}{\vdash^{1}}\text{ T-Var} \qquad \frac{\vdash^{w}}{\vdash^{w+1}}\text{ T-}\lambda \qquad \frac{}{\vdash^{0}}\text{ T-}\lambda_\star$$

$$\frac{\vdash^{w} \quad [\vdash^{v_i}]_{i\in[1,\ldots,n]}}{\vdash^{w+\sum v_i+1}}\text{ T-@}$$

λIAM

$$\frac{}{[A]\vdash^{\|A\|} A}\text{ T-Var} \qquad \frac{\Gamma, S\vdash^{w} A}{\Gamma\vdash^{w+\|S\to A\|} S\to A}\text{ T-}\lambda \qquad \frac{}{\Gamma\vdash^{0}\star}\text{ T-}\lambda_\star$$

$$\frac{\Gamma\vdash^{w}[A'_1,\ldots,A'_n]\to A \quad [\Delta_i\vdash^{v_i} A'_i]_{i\in[1,\ldots,n]}}{\Gamma\uplus\sum_{i\in[1,\ldots,n]}\Delta_i\vdash^{w+\sum v_i+\|A\|} A}\text{ T-@}$$

Fig. 7. The weight assignments $\mathbf{W}_{KAM}(\cdot)$, on the left, and $\mathbf{W}_{\lambda\mathrm{IAM}}(\cdot)$, on the right.

Sequence Types and KAM Time. Multi types have been successfully applied in quantitative analyses of normalization, starting with de Carvalho [2007, 2018] who used them to give a bound to the length of KAM runs. De Carvalho's technique can be re-phrased and distilled as a decoration of type derivations with *weights*, that is, cost annotations, following the scheme of Fig. 7. Please note that the weight assignment is blind to types, and thus relies only on the structure of the type derivation. De Carvalho's result can be formulated as follows.

Theorem 12.3 (De Carvalho). *There is a complete KAM run of length n from t if and only if exists π such that $\pi \triangleright \vdash t : \star$ and $\mathbf{W}_{KAM}(\pi) = n$.*

The KAM being deterministic, one has that all derivations $\vdash t : \star$ induce the same weights.

Sequence Types and λIAM Time. We use the same idea to capture the length of a λIAM run. We keep the same type system but we change the weight assignment to typing rules. First, we have to define a norm on types and sequence types, counts the number of occurrences of $\star$:

$$\|\star\| := 1 \qquad \|S\to A\| := \|S\| + \|A\| \qquad \big\|[A_1,\ldots,A_n]\big\| := \textstyle\sum_{1\le i\le n}\|A_i\|$$

Then we define the weight system $\mathbf{W}_{\lambda\mathrm{IAM}}(\cdot)$ in Fig. 7. Observe how this weight system is structurally very similar to $\mathbf{W}_{KAM}(\cdot)$, the only difference being the fact that whenever the latter adds 1 to the weight, the former adds the number of occurrences of $\star$ in the underlying type. The next section proves the following theorem, that is the λIAM analogous of de Carvalho's theorem.

Theorem 12.4. *There is a complete λIAM run of length n from t if and only if exists π such that $\pi \triangleright \vdash t : \star$ and $\mathbf{W}_{\lambda IAM}(\pi) = n$.*

13 THE SEQUENCE IAM

This section introduces yet another machine, the *Sequence IAM*, or *SIAM*, that mimics the λIAM directly on top of a type derivation π. It is the key tool used in the next section to show that the λIAM weights on type derivations do measure the time cost of λIAM runs.

The SIAM. The idea behind the SIAM is simple but a formal definition is a technical nightmare. Let us explain the idea. The machine moves over a fixed type derivation $\pi \triangleright \vdash t : \star$, to be thought as the code. The position of the machine is expressed by an occurrence of a type judgment[13] J of π. As the λIAM, the SIAM has two possible directions, noted $\downarrow$ and $\uparrow$[14]. In direction $\uparrow$ the machine looks at the rule above the focused judgment, in direction $\downarrow$ at the rule below. The only "data structure" is a type context $\mathbb{B}$ isolating an occurrence of $\star$ in the type A of the focused judgment (occurrence) $\Gamma \vdash u : A$, defined as follows (careful to not confuse type contexts $\mathbb{B}$ with type environments Γ):

Type ctxs $\quad \mathbb{B} ::= \langle\cdot\rangle \mid S \to \mathbb{B} \mid \mathbb{S} \to A \quad\Big|\quad$ Sequence ctxs $\quad \mathbb{S} ::= [A_1,..,A_k,\mathbb{B},A_{k+1}..,A_n]$

[13]A judgment may occur repeatedly in a derivation, which is why we talk about *occurrences* of judgments. To avoid too many technicalities, we usually just write the judgment, leaving implicit that we refer to an occurrence of that judgment.
[14]Type derivations are upside-down wrt to the term structure, then direction $\downarrow$ of the λIAM becomes here $\uparrow$, and $\uparrow$ is $\downarrow$.

$$\frac{\vdash t : S \to A \quad [\vdash]}{\vdash tu : \mathbb{B}\langle\star_\uparrow\rangle(=A)} \quad \to_{\bullet 1} \quad \frac{\vdash t : S \to \mathbb{B}\langle\star_\uparrow\rangle \quad [\vdash]}{\vdash tu : A} \qquad\Bigg\| \qquad \frac{\vdash t : A(=\mathbb{B}\langle\star\rangle)}{\vdash \lambda x.t : S \to \mathbb{B}\langle\star_\uparrow\rangle} \quad \to_{\bullet 2} \quad \frac{\vdash t : \mathbb{B}\langle\star_\uparrow\rangle}{\vdash \lambda x.t : S \to A}$$

$$\frac{\vdash t : S \to \mathbb{B}\langle\star_\downarrow\rangle \quad [\vdash]}{\vdash tu : A(=\mathbb{B}\langle\star\rangle)} \quad \to_{\bullet 3} \quad \frac{\vdash t : S \to A \quad [\vdash]}{\vdash tu : \mathbb{B}\langle\star_\downarrow\rangle} \qquad\Bigg\| \qquad \frac{\vdash t : \mathbb{B}\langle\star_\downarrow\rangle(=A)}{\vdash \lambda x.t : S \to A} \quad \to_{\bullet 4} \quad \frac{\vdash t : A}{\vdash \lambda x.t : S \to \mathbb{B}\langle\star_\downarrow\rangle}$$

$$\begin{array}{c} \overline{\vdash x : \mathbb{B}\langle\star_\uparrow\rangle_i(=A_i)}\ i \\ \vdots \\ \hline \vdash \lambda x.C\langle x\rangle : [\dots A_i \dots] \to A' \end{array} \quad \to_{\mathsf{var}} \quad \begin{array}{c} \overline{\vdash x : A_i}\ i \\ \vdots \\ \hline \vdash \lambda x.C\langle x\rangle : [\dots \mathbb{B}\langle\star_\downarrow\rangle_i \dots] \to A' \end{array}$$

$$\begin{array}{c} \overline{\vdash x : A_i(=\mathbb{B}\langle\star\rangle_i)}\ i \\ \vdots \\ \hline \vdash \lambda x.C\langle x\rangle : [\dots \mathbb{B}\langle\star_\uparrow\rangle_i \dots] \to A' \end{array} \quad \to_{\mathsf{bt2}} \quad \begin{array}{c} \overline{\vdash x : \mathbb{B}\langle\star_\downarrow\rangle_i}\ i \\ \vdots \\ \hline \vdash \lambda x.C\langle x\rangle : [\dots A_i \dots] \to A' \end{array}$$

$$\frac{\vdash t : [\dots \mathbb{B}\langle\star_\downarrow\rangle_i \dots] \to A' \quad \vdash_i u : A_i(=\mathbb{B}\langle\star\rangle_i)}{\vdash tu : A'} \quad \to_{\mathsf{arg}} \quad \frac{\vdash t : [\dots A_i \dots] \to A' \quad \vdash_i u : \mathbb{B}\langle\star_\uparrow\rangle_i}{\vdash tu : A'}$$

$$\frac{\vdash t : [\dots A_i \dots] \to A' \quad \vdash_i u : \mathbb{B}\langle\star_\downarrow\rangle_i(=A_i)}{\vdash tu : A'} \quad \to_{\mathsf{bt1}} \quad \frac{\vdash t : [\dots \mathbb{B}\langle\star_\uparrow\rangle_i \dots] \to A' \quad \vdash_i u : A_i}{\vdash tu : A'}$$

Fig. 8. The transitions of the Sequence IAM (SIAM).

Summing up, a state s is a quadruple $(\pi, J, \mathbb{B}, d)$. If J is in the form $\Gamma \vdash u : A$, we often write s as $\vdash u : \mathbb{B}\langle\star_d\rangle$, where $\mathbb{B}\langle\star\rangle = A$. In fact we shall see that type environments play no role.

Transitions. The SIAM starts on the final judgment of π, with empty type context $\mathbb{B} = \langle\cdot\rangle$ and direction $\uparrow$. It moves from judgment to judgment, following occurrences of $\star$ around π. The transitions are in Fig. 8, their union noted $\to_{\text{SIAM}}$, as we now explain them—the transitions have the labels of λIAM transitions, because they correspond to each other, as we shall show.

Let's start with the simplest, $\to_{\bullet 2}$. The state focusses on the conclusion judgment J of a T-λ rule with direction $\uparrow$. The eventual type environment Γ is omitted because the transition does not depend on it—none of the transitions does, so type environments are omitted from all transitions. The judgment assigns type $S \to A$ to $\lambda x.t$, and the type context is $S \to \mathbb{B}$, that is, it selects an occurrence of $\star$ in the target type $A = \mathbb{B}\langle\star\rangle$. The transition then simply moves to the judgment above, stripping down the type context to $\mathbb{B}$, and keeping the same direction. Transition $\bullet 4$ does the opposite move, in direction $\downarrow$, and transitions $\bullet 1$ and $\bullet 3$ behave similarly on T-@ rules: $[\vdash]$ simply denotes the right premise that is left unspecified since not relevant to the transition.

Transitions $\to_{\mathsf{arg}}$: the focus is on the left premise of a T-@ rule, of type $S \to A'$ isolating $\star$ inside the i-th type A_i in S. The transition then moves to the final judgment of the i-th derivation in the right premise, changing direction. Transition $\to_{\mathsf{bt1}}$ does the opposite move.

Transitions $\to_{\mathsf{var}}$ and $\to_{\mathsf{bt2}}$ are based on the axiom sequences property of Lemma 12.1. Consider a T-λ rule occurrence whose right-hand type of the conclusion is $S \to A'$. The premise has shape $\Gamma, x : S \vdash t : A'$, and by the lemma there is a bijection between the sequence of linear types in S and the axioms on x, respecting the order in S. The left side of $\to_{\mathsf{bt2}}$ focuses on the i-th type A_i in S

and the SIAM moves to the judgment of the axiom corresponding to that type, which is exactly the i-th from left to right seeing the derivation as a tree where the children of nodes are ordered as in the typing rules. Transition $\rightarrow_{\text{var}}$ does the opposite move, which can always happen because the code is the type derivation of a closed term.

The only typing rule not inducing a transition is T-$\lambda_\star$. Accordingly, when the SIAM reaches one of these rules it is in a final state. Exactly as the λIAM, the SIAM is bi-deterministic.

Proposition 13.1. *The SIAM is bi-deterministic for each type derivation $\pi \triangleright \vdash t : \star$.*

An example. We present below the very same example analyzed in Section 3. We have reported its type derivation, with the occurrences of $\star$ on the right of $\vdash$ annotated with increasing integers and a direction. The occurrence of $\star$ marked with 1 represents the first state, and so on.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : [[\star] \rightarrow \star] \vdash x : [\star_{\downarrow 16}] \rightarrow \star_{\uparrow 6}} \qquad \dfrac{}{y : [\star] \vdash y : \star_{\uparrow 17}}}{y : [\star], x : [[\star] \rightarrow \star] \vdash xy : \star_{\uparrow 5}}}{y : [\star] \vdash \lambda x.xy : [[\star_{\uparrow 15}] \rightarrow \star_{\downarrow 7}] \rightarrow \star_{\uparrow 4}}}{\vdash \lambda y.\lambda x.xy : [\star_{\downarrow 18}] \rightarrow [[\star_{\uparrow 14}] \rightarrow \star_{\downarrow 8}] \rightarrow \star_{\uparrow 3}} \qquad \dfrac{}{\vdash \mathsf{I} : \star_{\uparrow 19}}}{\vdash (\lambda y.\lambda x.xy)\mathsf{I} : [[\star_{\uparrow 13}] \rightarrow \star_{\downarrow 9}] \rightarrow \star_{\uparrow 2}} \qquad \dfrac{z : [\star] \vdash z : \star_{\uparrow 11}}{\vdash \lambda z.z : [\star_{\downarrow 12}] \rightarrow \star_{\uparrow 10}}}{\vdash (\lambda y.\lambda x.xy)\mathsf{I}(\lambda z.z) : \star_{\uparrow 1}}$$

One can immediately notice that every occurrence of $\star$ is visited exactly once. Moreover, the sequence of the visited subterms is the same as the one obtained in the example of Section 3.

14 λ-IAM TIME VIA EXHAUSTING SEQUENCE TYPES

The aim of this section is to explain the strong bisimulation between the SIAM and the λIAM, that, once again, is based on a variation on the exhaustible invariant. A striking point of the SIAM is that it does not have the log nor the tape. They are encoded in the judgment occurrence J and in the type context $\mathbb{B}$ of its states, as we shall show. But first, let's make a step back.

Handling Duplication. β-reducing a λ-term (potentially) duplicates arguments, whose different copies may be used differently, typically being applied to different further arguments. The machines in this paper never duplicate arguments, but have nonetheless to distinguish different uses of a same piece of code. This is why the λIAM uses *logged* positions instead of simple positions: the log is a trace of (part of) the previous run that allows to distinguishing different uses of the position—the closures of the KAM or the history mechanism of the λPAM are alternatives.

The key point of multi/sequence type derivations is that duplication is explicitly accounted for, somewhat *in advance*, by multi-set/sequences: all arguments come with as many type derivations as the times they are duplicated during evaluation. Note indeed that the type derivation may be way bigger than the term itself, while this is not possible with, say, simple types. Therefore, there is no need to resort to logs, closures, or histories to distinguish copies, because all copies are already there: simple positions in the type derivation (not in the term!) are informative enough.

In the technical report [Accattoli et al. 2020a] we provide the execution of the term $(\lambda x.xx)\mathsf{I}$, that actually duplicates the sub-term I, for all the machines presented in the paper.

Relating Logs and Tapes with Typed Positions. In the λIAM, the log $L = l_1 \cdot \ldots \cdot l_n$ has a logged position for every argument $u_1, \ldots, u_n$ in which the position of the current state is contained. The argument u_i is the answer to the query of an argument for the variable in the logged position l_i. The SIAM does not keep a trace of the variables for which it completed a query, but the answers to those (forgotten) queries are simply given by the sub-derivations for $u_1, \ldots, u_n$ in which the current judgment occurrence J is contained—the way in which l_k identifies a copy of u_k in the

λIAM corresponds on the type derivation π to the index i of the sub-derivation (in the sequence of sub-derivations) typing u_k in which J is located. Note that the λIAM manipulates the log only via transitions $\to_{\text{arg}}$ and $\to_{\text{bt1}}$, that on the SIAM correspond exactly to entering/exiting derivations for arguments. The tape, instead, contains logged positions for which the λIAM either has not yet found the associated argument, or it is backtracking to. Note that the λIAM puts logged positions on the tape via transitions $\to_{\text{var}}$ and $\to_{\text{bt1}}$, and removes them using $\to_{\text{arg}}$ and $\to_{\text{bt2}}$. By looking at Fig. 8, it is evident that there is a logged position on the λIAM tape for every type sequence S in which it lies the hole $\langle\cdot\rangle$ of the current type context $\mathbb{B}$ of the SIAM.

These ideas are used to extract from every SIAM state s a λIAM state $\text{ext}(s)$ in a quite technical way. A notable point is that the extraction procedure is formally defined by means of yet another reformulation on the SIAM of the exhaustible invariant, called *S-exhaustibility*, relying on tests induced by a SIAM state built following the explained correspondence. For lack of space the technical development is in [Accattoli et al. 2020a]. The extraction process induces a relation $s \simeq_{\text{ext}} \text{ext}(s)$ that is easily proved to be a strong bisimulation between the SIAM and the λIAM.

Proposition 14.1. *Let t a closed and $\to_{wh}$-normalizable term, and $\pi \triangleright \vdash t : \star$ a type derivation. Then $\simeq_{\text{ext}}$ is a strong bisimulation between SIAM states on π and λIAM states on t.*

Weights and the Length of SIAM Runs via Acyclicity. We now turn to the proof of the correctness of the weight assignment $\mathbf{W}_{\lambda\text{IAM}}(\pi)$, that is, the fact that it correctly measures the length of λIAM complete runs. While the weight assignment for the λIAM is similar to de Carvalho's one for the KAM, the proof of its correctness is completely different, and it must be, as we know explain.

The KAM performs an evaluation that essentially mimics cut-elimination and so the number of KAM transitions to normal form is obtained via a refined, quantitative form of subject reduction. One may say that it is obtained in a *step-by-step* manner. The λIAM, instead, does not mimic subject reduction. It walks over the type derivation *without ever changing it*, potentially passing many times over the same judgment (because of backtracking). Correctness of weights cannot then be obtained via a refined subject reduction property, because the reduced derivation gives rise to a different run, and not to a sub-run. It must instead follow from a *global* analysis of a fixed derivation, that we now develop. The proof technique is an original contribution of this paper.

Weights as in $\mathbf{W}_{\lambda\text{IAM}}(\pi)$ count the number of occurrences of $\star$ in π, and every such occurrence corresponds to a state of the SIAM. Proving the correctness of the weight system amounts to showing that every state of the SIAM is reachable, and reachable exactly once. In order to do so, we have to show that the SIAM never loops on typed derivations.

Note a subtlety: by the bisimulation with the λIAM (Prop. 14.1) we know that the run of the SIAM terminates, but we do not know whether it reaches all states. What we have to prove, then, is that there are no unreachable loops, that is, loops that are not reachable from an initial state. The next easy lemma guarantees that this is enough.

Lemma 14.2. *Let T be an acyclic bi-deterministic transition system on a finite set of states $\mathcal{S}$ and with only one initial state s_i. Then all states in $\mathcal{S}$ are reachable from s_i, and reachable only once.*

We show the absence of loops using a sort of subject reduction property. We first show that if the SIAM loops on $\pi \triangleright \vdash t : \star$ and $t \to_{wh} u$, then there is a type derivation $\pi' \triangleright \vdash u : \star$ on which the SIAM loops—that is, SIAM looping is preserved by reduction of the underlying term. This is done by defining a relation $\blacktriangleright$ between the SIAM states on π and on π'—see [Accattoli et al. 2020a].

Proposition 14.3. *$\blacktriangleright$ is a loop-preserving bisimulation between SIAM states.*

Other works dealing with the GoI also prove the absence of unreachable loops, for instance [Baillot 1999; Baillot et al. 2011]. Then, by the trivial fact that the SIAM does not loop on $\to_{wh}$-normal terms (as they are typed using just one rule, namely T-$\lambda_\star$), we obtain that it never loops.

Corollary 14.4. *Let $\pi \triangleright \vdash t : \star$ be a type derivation. Then the SIAM does not loop on π.*

The correctness of the weights for the length of SIAM runs immediately follows, and, via the strong bisimulation in Prop. 14.1, it transfers to the λIAM.

Theorem 14.5 (λIAM time via sequence types). *Let t be a closed term that is $\rightarrow_{wh}$-normalizable, σ the complete λIAM run from s_t, and $\pi \triangleright \vdash t : \star$ a type derivation for t. Then $|\sigma| = \mathbf{W}_{\lambda IAM}(\pi)$.*

15 OUR TWO CENTS ABOUT SPACE

Here we provide an interesting example about space usage, with the only purpose of stressing that the situation is subtler than for time. Among the machines we have presented, the λIAM is the only one tuned for space efficiency, as shown by the literature [Dal Lago and Schöpp 2010; Ghica 2007; Ghica and Smith 2010; Mazza 2015; Mazza and Terui 2015; Schopp 2007]. In fact, the space used by the λJAM (thus the λPAM) and the KAM is proportional to their time, *i.e.* their space usage is inflationary. Nonetheless, there are terms for which the λJAM outperforms the λIAM in space consumption, showing that the space relationship between the λIAM and the λJAM is less smooth than the time one.

Proposition 15.1. *Let r^h_k be defined as $r^h_k := (\lambda x_1 ... \lambda x_k.\lambda y.y(\lambda z_1 ... \lambda z_h.\lambda z.z))t_1 ... t_k(\lambda w.wu_1 ... u_h)$. The the λIAM space consumption for the evaluation of r^h_k is 2 logged positions plus $h + k$ occurrences of $\bullet$, while the λJAM needs 2 logged positions plus $\max\{h, k+1\}$ occurrences of $\bullet$.*

16 CONCLUSIONS

In this paper, we give alternative presentation of three game machines, namely the IAM the JAM and the PAM, formulated as ordinary abstract machines on λ-terms, and analyze their relative time performances. Our results can be summarized as follows:

$$\begin{tikzcd}
\lambda\text{IAM} \arrow[r, "\text{exponential speedup}", "\text{w/o backtracking}"'] \arrow[d, leftrightarrow, thick] & \lambda\text{JAM} \arrow[r, "\text{quadratic speedup}", "\text{w/o } \uparrow\text{-mode}"'] \arrow[d, leftrightarrow, thick] & \text{KAM} \\
\text{SIAM} & \lambda\text{PAM} &
\end{tikzcd}$$

Here, thicker arrows represent a stronger correspondence, i.e., the λPAM and λJAM are isomorphic, the λJAM improves the λIAM with a possibly exponential advantage, while the KAM improves the λJAM (thus the λPAM) with a potential quadratic speedup.

Besides settling the question about the relative efficiency of the main game machines, we also prove non-idempotent intersection types to be able to precisely characterize the time performance of the λIAM when run on the typed term, in analogy with the results on environment machines by de Carvalho [2018]. This way, the time behavior of two heterogeneous machines, namely the KAM and the λIAM, on a given normalizing term t can be captured by just comparing two different ways of weighting the same sequence type derivation, the former attributing weight 1 to any instance rule in the type derivation, the latter taking into account the size of the underlying type in an essential way. In other words, *the bigger the types, the more inefficient the λIAM.*

Among the topics for future work, we can certainly mention the extension of the results obtained here to *call-by-value* game machines, which seems within reach. A study on the relative *space* efficiency of game machines is more elusive, as the example in Section 15 shows.

ACKNOWLEDGMENTS

The second author is funded by the ERC CoG "DIAPASoN" (GA 818616). This work has been partially funded by the ANR JCJC grant "COCA HOLA" (ANR-16-CE40-004-01).

REFERENCES

Samson Abramsky, Radha Jagadeesan, and Pasquale Malacaria. 2000. Full Abstraction for PCF. *Inf. Comput.* 163, 2 (2000), 409–470. https://doi.org/10.1006/inco.2000.2930

Beniamino Accattoli. 2017. (In)Efficiency and Reasonable Cost Models. In *12th Workshop on Logical and Semantic Frameworks, with Applications, LSFA 2017, Brasília, Brazil, September 23-24, 2017 (Electronic Notes in Theoretical Computer Science, Vol. 338).* Elsevier, 23–43. https://doi.org/10.1016/j.entcs.2018.10.003

Beniamino Accattoli, Pablo Barenbaum, and Damiano Mazza. 2014. Distilling abstract machines. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional progranitarmming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 363–376. https://doi.org/10.1145/2628136.2628154

Beniamino Accattoli and Bruno Barras. 2017. Environments and the complexity of abstract machines. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, Namur, Belgium, October 09 - 11, 2017*, Wim Vanhoof and Brigitte Pientka (Eds.). ACM, 4–16. https://doi.org/10.1145/3131851.3131855

Beniamino Accattoli, Andrea Condoluci, Giulio Guerrieri, and Claudio Sacerdoti Coen. 2019a. Crumbling Abstract Machines. In *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, Ekaterina Komendantskaya (Ed.). ACM, 4:1–4:15. https://doi.org/10.1145/3354166.3354169

Beniamino Accattoli and Ugo Dal Lago. 2012. On the Invariance of the Unitary Cost Model for Head Reduction. In *23rd International Conference on Rewriting Techniques and Applications (RTA'12) , RTA 2012, May 28 - June 2, 2012, Nagoya, Japan (LIPIcs, Vol. 15)*, Ashish Tiwari (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 22–37. https://doi.org/10.4230/LIPIcs.RTA.2012.22

Beniamino Accattoli and Ugo Dal Lago. 2016. (Leftmost-Outermost) Beta Reduction is Invariant, Indeed. *Logical Methods in Computer Science* 12, 1 (2016). https://doi.org/10.2168/LMCS-12(1:4)2016

Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2020a. *The (In)Efficiency of Interaction.* Technical Report. https://arxiv.org/abs/2010.12988.

Beniamino Accattoli, Ugo Dal Lago, and Gabriele Vanoni. 2020b. The Machinery of Interaction. In *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020.* ACM, 4:1–4:15. https://doi.org/10.1145/3414080.3414108

Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. 2020c. Tight typings and split bounds, fully developed. *J. Funct. Program.* 30 (2020), e14. https://doi.org/10.1017/S095679682000012X

Beniamino Accattoli and Giulio Guerrieri. 2018. Types of Fireballs. In *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11275)*, Sukyoung Ryu (Ed.). Springer, 45–66. https://doi.org/10.1007/978-3-030-02768-1_3

Beniamino Accattoli and Giulio Guerrieri. 2019. Abstract machines for Open Call-by-Value. *Sci. Comput. Program.* 184 (2019). https://doi.org/10.1016/j.scico.2019.03.002

Beniamino Accattoli, Giulio Guerrieri, and Maico Leberle. 2019b. Types by Need. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423).* Springer, 410–439. https://doi.org/10.1007/978-3-030-17184-1_15

Federico Aschieri. 2017. Game Semantics and the Geometry of Backtracking: a New Complexity Analysis of Interaction. *J. Symb. Log.* 82, 2 (2017), 672–708. https://doi.org/10.1017/jsl.2016.48

Andrea Asperti, Vincent Danos, Cosimo Laneve, and Laurent Regnier. 1994. Paths in the lambda-calculus. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science (LICS '94), Paris, France, July 4-7, 1994.* IEEE Computer Society, 426–436. https://doi.org/10.1109/LICS.1994.316048

Patrick Baillot. 1999. *Approches dynamiques en sémantique de la logique lineaire: jeux et géométrie de l'interaction.* PhD Thesis. Universite Aix-Marseille 2.

Patrick Baillot, Paolo Coppola, and Ugo Dal Lago. 2011. Light logics and optimal reduction: Completeness and complexity. *Inf. Comput.* 209, 2 (2011), 118–142. https://doi.org/10.1016/j.ic.2010.10.002

Hendrik Pieter Barendregt. 1984. *The lambda calculus: its syntax and semantics.* North-Holland.

Daniil Berezun and Neil D. Jones. 2017. Compiling untyped lambda calculus to lower-level code by game semantics and partial evaluation (invited paper). In *Proceedings of the 2017 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2017, Paris, France, January 18-20, 2017*, Ulrik Pagh Schultz and Jeremy Yallop (Eds.). ACM, 1–11.

Alexis Bernadet and Stéphane Graham-Lengrand. 2013. Non-idempotent intersection types and strong normalisation. *Logical Methods in Computer Science* 9, 4 (2013). https://doi.org/10.2168/LMCS-9(4:3)2013

Guy E. Blelloch and John Greiner. 1995. Parallelism in Sequential Functional Languages. In *Proceedings of the seventh international conference on Functional programming languages and computer architecture, FPCA 1995, La Jolla, California, USA, June 25-28, 1995.* ACM, 226–237. https://doi.org/10.1145/224164.224210

William Blum. 2020. Evaluating lambda terms with traversals. *Theor. Comput. Sci.* 802 (2020), 77–104. https://doi.org/10.1016/j.tcs.2019.08.035

Antonio Bucciarelli, Delia Kesner, Alejandro Ríos, and Andrés Viso. 2020. The Bang Calculus Revisited. In *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12073)*, Keisuke Nakano and Konstantinos Sagonas (Eds.). Springer, 13–32. https://doi.org/10.1007/978-3-030-59025-3_2

Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. 2017. Non-idempotent intersection types for the Lambda-Calculus. *Logic Journal of the IGPL* 25, 4 (2017), 431–464. https://doi.org/10.1093/jigpal/jzx018

Pierre Clairambault. 2011. Estimation of the Length of Interactions in Arena Game Semantics. In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6604)*, Martin Hofmann (Ed.). Springer, 335–349. https://doi.org/10.1007/978-3-642-19805-2_23

Pierre Clairambault. 2015. Bounding linear head reduction and visible interaction through skeletons. *Log. Methods Comput. Sci.* 11, 2 (2015). https://doi.org/10.2168/LMCS-11(2:6)2015

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1978. A new type assignment for λ-terms. *Arch. Math. Log.* 19, 1 (1978), 139–156. https://doi.org/10.1007/BF02011875

Mario Coppo and Mariangiola Dezani-Ciancaglini. 1980. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic* 21, 4 (1980), 685–693. https://doi.org/10.1305/ndjfl/1093883253

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1980. Principal Type Schemes and Lambda-calculus Semantics. In *To H.B.Curry: Essays on Combinatory Logic, Lambda-calculus and Formalism.* Academic Press, 535–560. http://www.di.unito.it/~dezani/papers/CDV80.pdf

Pierre-Louis Curien and Hugo Herbelin. 1998. Computing with Abstract Böhm Trees. In *Third Fuji International Symposium on Functional and Logic Programming, FLOPS 1998, Kyoto, Japan, Apil 2-4, 1998*, Masahiko Sato and Yoshihito Toyama (Eds.). World Scientific, Singapore, 20–39.

Pierre-Louis Curien and Hugo Herbelin. 2007. Abstract machines for dialogue games. (2007). http://arxiv.org/abs/0706.2544

Ugo Dal Lago, Claudia Faggian, Ichiro Hasuo, and Akira Yoshimizu. 2014. The geometry of synchronization. In *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*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 35:1–35:10. https://doi.org/10.1145/2603088.2603154

Ugo Dal Lago, Claudia Faggian, Benoît Valiron, and Akira Yoshimizu. 2015. 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.* IEEE Computer Society, 559–572. https://doi.org/10.1109/LICS.2015.58

Ugo Dal Lago and Ulrich Schöpp. 2010. Functional Programming in Sublinear Space. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings (Lecture Notes in Computer Science, Vol. 6012)*, Andrew D. Gordon (Ed.). Springer, 205–225. https://doi.org/10.1007/978-3-642-11957-6_12

Ugo Dal Lago and Ulrich Schöpp. 2016. Computation by interaction for space-bounded functional programming. *Information and Computation* 248 (2016), 150–194. https://doi.org/10.1016/j.ic.2015.04.006

Ugo Dal Lago, Ryo Tanaka, and Akira Yoshimizu. 2017. 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.* IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005112

Ugo Dal Lago and Margherita Zorzi. 2014. Wave-Style Token Machines and Quantum Lambda Calculi. In *Proceedings Third International Workshop on Linearity, LINEARITY 2014, Vienna, Austria, 13th July, 2014 (EPTCS, Vol. 176)*, Sandra Alves and Iliano Cervesato (Eds.). 64–78. https://doi.org/10.4204/EPTCS.176.6

Vincent Danos, Hugo Herbelin, and Laurent Regnier. 1996. Game Semantics & Abstract Machines. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science, New Brunswick, New Jersey, USA, July 27-30, 1996.* IEEE Computer Society, 394–405. https://doi.org/10.1109/LICS.1996.561456

Vincent Danos and Laurent Regnier. 1993. 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.* IEEE Computer Society, 296–306. https://doi.org/10.1109/LICS.1993.287578

Vincent Danos and Laurent Regnier. 1995. Proof-Nets and the Hilbert Space. In *Proceedings of the Workshop on Advances in Linear Logic.* Cambridge University Press, USA, 307–328.

Vincent Danos and Laurent Regnier. 1999. Reversible, irreversible and optimal lambda-machines. *Theoretical Computer Science* 227, 1 (1999), 79–97. https://doi.org/10.1016/S0304-3975(99)00049-3

Vincent Danos and Laurent Regnier. 2004. *Head Linear Reduction.* Technical Report.

Daniel de Carvalho. 2007. *Sémantiques de la logique linéaire et temps de calcul.* Thèse de Doctorat. Université Aix-Marseille II.

Daniel de Carvalho. 2018. Execution time of λ-terms via denotational semantics and intersection types. *Math. Str. in Comput. Sci.* 28, 7 (2018), 1169–1203. https://doi.org/10.1017/S0960129516000396

Daniel de Carvalho, Michele Pagani, and Lorenzo Tortora de Falco. 2011. A semantic measure of the execution time in linear logic. *Theoretical Computer Science* 412, 20 (2011), 1884–1902. https://doi.org/10.1016/j.tcs.2010.12.017

Maribel Fernández and Ian Mackie. 2002. Call-by-Value lambda-Graph Rewriting Without Rewriting. In *Graph Transformation, First International Conference, ICGT 2002, Barcelona, Spain, October 7-12, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2505)*, Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg (Eds.). Springer, 75–89. https://doi.org/10.1007/3-540-45832-8_8

Philippa Gardner. 1994. Discovering Needed Reductions Using Type Theory. In *Theoretical Aspects of Computer Software (TACS '94) (Lecture Notes in Computer Science, Vol. 789)*. 555–574. https://doi.org/10.1007/3-540-57887-0_115

Dan R. Ghica. 2007. Geometry of Synthesis: A Structured Approach to VLSI Design. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 363–375. https://doi.org/10.1145/1190216.1190269

Dan R. Ghica and Alex I. Smith. 2010. Geometry of Synthesis II: From Games to Delay-Insensitive Circuits. In *Proceedings of the 26th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2010, Ottawa, Ontario, Canada, May 6-10, 2010 (Electronic Notes in Theoretical Computer Science, Vol. 265)*, Michael W. Mislove and Peter Selinger (Eds.). Elsevier, 301–324. https://doi.org/10.1016/j.entcs.2010.08.018

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

Jean-Yves Girard. 1989. Geometry of Interaction 1: Interpretation of System F. In *Studies in Logic and the Foundations of Mathematics*, R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo (Eds.). Vol. 127. Elsevier, 221–260.

Naohiko Hoshino, Koko Muroya, and Ichiro Hasuo. 2014. Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In *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*, Thomas A. Henzinger and Dale Miller (Eds.). ACM, 52:1–52:10. https://doi.org/10.1145/2603088.2603124

J. M. E. Hyland and C.-H. Luke Ong. 2000. On Full Abstraction for PCF: I, II, and III. *Inf. Comput.* 163, 2 (2000), 285–408. https://doi.org/10.1006/inco.2000.2917

Delia Kesner and Pierre Vial. 2020. Consuming and Persistent Types for Classical Logic. In *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller (Eds.). ACM, 619–632. https://doi.org/10.1145/3373718.3394774

Assaf J. Kfoury. 2000. A linearization of the Lambda-calculus and consequences. *Journal of Logic and Computation* 10, 3 (2000), 411–436. https://doi.org/10.1093/logcom/10.3.411

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-order functional reactive programming in bounded space. In *Proc. of POPL 2012*, John Field and Michael Hicks (Eds.). ACM, 45–58. https://doi.org/10.1145/2103656.2103665

Jean-Louis Krivine. 1993. *Lambda-calculus, types and models.* Masson.

Jean-Louis Krivine. 2007. A Call-by-name Lambda-calculus Machine. *Higher Order Symbol. Comput.* 20, 3 (2007), 199–207. https://doi.org/10.1007/s10990-007-9018-9

Olivier Laurent. 2001. A Token Machine for Full Geometry of Interaction. In *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2044)*, Samson Abramsky (Ed.). Springer, 283–297. https://doi.org/10.1007/3-540-45413-6_23

Jean-Jacques Lévy. 1978. *Réductions correctes et optimales dans le lambda-calcul.* PhD Thesis. Université Paris 7.

Ian Mackie. 1995. The Geometry of Interaction Machine. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, Ron K. Cytron and Peter Lee (Eds.). ACM Press, 198–208. https://doi.org/10.1145/199448.199483

Ian Mackie. 2017. A Geometry of Interaction Machine for Gödel's System T. In *Logic, Language, Information, and Computation - 24th International Workshop, WoLLIC 2017, London, UK, July 18-21, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10388)*, Juliette Kennedy and Ruy J. G. B. de Queiroz (Eds.). Springer, 229–241. https://doi.org/10.1007/978-3-662-55386-2_16

Damiano Mazza. 2015. Simple Parsimonious Types and Logarithmic Space. In *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany (LIPIcs, Vol. 41)*, Stephan Kreutzer (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 24–40. https://doi.org/10.4230/LIPIcs.CSL.2015.24

Damiano Mazza, Luc Pellissier, and Pierre Vial. 2018. Polyadic approximations, fibrations and intersection types. *Proc. ACM Program. Lang.* 2, POPL (2018), 6:1–6:28. https://doi.org/10.1145/3158094

Damiano Mazza and Kazushige Terui. 2015. Parsimonious Types and Non-uniform Computation. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9135)*, Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann

(Eds.). Springer, 350–361. https://doi.org/10.1007/978-3-662-47666-6_28

Robin Milner. 1977. Fully Abstract Models of Typed *lambda*-Calculi. *Theor. Comput. Sci.* 4, 1 (1977), 1–22. https://doi.org/10.1016/0304-3975(77)90053-6

Koko Muroya and Dan R. Ghica. 2017. The Dynamic Geometry of Interaction Machine: A Call-by-Need Graph Rewriter. In *26th EACSL Annual Conference on Computer Science Logic, CSL 2017, August 20-24, 2017, Stockholm, Sweden (LIPIcs, Vol. 82)*, Valentin Goranko and Mads Dam (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 32:1–32:15. https://doi.org/10.4230/LIPIcs.CSL.2017.32

Koko Muroya and Dan R. Ghica. 2019. The Dynamic Geometry of Interaction Machine: A Token-Guided Graph Rewriter. *Log. Methods Comput. Sci.* 15, 4 (2019). https://lmcs.episciences.org/5882

Peter Møller Neergaard and Harry G. Mairson. 2004. Types, potency, and idempotency: why nonlinearity and amnesia make a type system work. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP 2004, Snow Bird, UT, USA, September 19-21, 2004*, Chris Okasaki and Kathleen Fisher (Eds.). ACM, 138–149. https://doi.org/10.1145/1016850.1016871

C.-H. Luke Ong. 2006. On Model-Checking Trees Generated by Higher-Order Recursion Schemes. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. IEEE Computer Society, 81–90. https://doi.org/10.1109/LICS.2006.38

Marco Pedicini and Francesco Quaglia. 2007. PELCR: Parallel environment for optimal lambda-calculus reduction. *ACM Trans. Comput. Log.* 8, 3 (2007), 14. https://doi.org/10.1145/1243996.1243997

Jorge Sousa Pinto. 2001. Parallel Implementation Models for the lambda-Calculus Using the Geometry of Interaction. In *Proc. of TLCA 2001 (LNCS, Vol. 2044)*, Samson Abramsky (Ed.). Springer, 385–399. https://doi.org/10.1007/3-540-45413-6_30

Garrel Pottinger. 1980. A Type Assignment for The Strongly Normalizable λ-terms. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J.P. Seldin and J.R. Hindley (Eds.). Academic Press, 561–578.

David Sands, Jörgen Gustavsson, and Andrew Moran. 2002. Lambda Calculi and Linear Speedups. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday] (Lecture Notes in Computer Science, Vol. 2566)*, Torben Æ. Mogensen, David A. Schmidt, and Ivan Hal Sudborough (Eds.). Springer, 60–84. https://doi.org/10.1007/3-540-36377-7_4

Ulrich Schopp. 2007. Stratified Bounded Affine Logic for Logarithmic Space. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*. IEEE Computer Society, 411–420. https://doi.org/10.1109/LICS.2007.45

Ulrich Schöpp. 2014. 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

Ulrich Schöpp. 2015. From Call-by-Value to Interaction by Typed Closure Conversion. In *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)*, Xinyu Feng and Sungwoo Park (Eds.). Springer, 251–270. https://doi.org/10.1007/978-3-319-26529-2_14

Takeshi Tsukada, Kazuyuki Asada, and C.-H. Luke Ong. 2017. Generalised species of rigid resource terms. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, 1–12. https://doi.org/10.1109/LICS.2017.8005093