The Space of Interaction

The Space of Interaction

Beniamino Accattoli
Inria & École Polytechnique

Ugo Dal Lago
Università di Bologna & Inria

Gabriele Vanoni
Università di Bologna & Inria

***Abstract*—The space complexity of functional programs is not well understood. In particular, traditional implementation techniques are tailored to time efficiency, and space efficiency induces time *in*efficiencies, as it prefers re-computing to saving. Girard's geometry of interaction underlies an alternative approach based on the interaction abstract machine (IAM), claimed as space efficient in the literature. It has also been conjectured to provide a reasonable notion of space for the λ-calculus, but such an important result seems to be elusive.**

In this paper we introduce a new intersection type system precisely measuring the space consumption of the IAM on the typed term. Intersection types have been repeatedly used to measure time, which they achieve by dropping idempotency, turning intersections into *multisets*. Here we show that the space consumption of the IAM is connected to a further structural modification, turning multisets into *trees*. Tree intersection types lead to a finer understanding of some space complexity results from the literature. They also shed new light on the conjecture about reasonable space: we show that the usual way of encoding Turing machines into the λ-calculus *cannot* be used to prove that the space of the IAM is a reasonable cost model.

I. Introduction

Type systems are a form of compositional abstraction in which the behavior of programs, particularly higher-order programs, is described by *types*, that is, specifications of the kinds of objects programs expect in input and are supposed to produce as output. Typed programs usually *cannot go wrong*, as types guarantee the absence of run-time errors [14]. Some type systems ensure also other properties such as termination or (various forms of) security.

Intersection Types and Time Complexity: Among the many existing type systems one can define on top of the pure, untyped, λ-calculus [12], [31], *intersection types* have the peculiar feature of *characterizing* a property rather than merely *guaranteeing* it. This, in particular, is known to hold for various forms of termination properties since the pioneering work by Coppo and Dezani in 1978 [15]. More recently, a variant of intersection types has been proved to reflect *quantitative* properties of terms, such as the number of β-steps to normal form, or the number of steps of the Krivine abstract machine [32] (shortened to KAM), as discovered by de Carvalho [23]–[25]. The variant requires dropping idempotency of the intersection operator, therefore considering $A \wedge A$ as *not* equivalent to A, and ultimately making the type system strongly related to the modeling of resources as in linear logic. Such types are sometimes called *multi types*, as intersections become multisets. In the last few years, de Carvalho's results have been dissected and generalized in various ways [8]–[11], [13], [17], [30]. In particular, research on the topic received renewed attention after some recent progress in the study of reasonable cost models for the λ-calculus by Accattoli and Dal Lago [4] made evident that counting β-steps gives rise to a reasonable cost model for time.

Sharing, Time, and Space: The mainstream way of implementing the λ-calculus, at work also in the KAM, consists of mimicking β-steps while replacing meta-level substitutions with a finer approach based on environments. Environments are a way of realizing a form of *subterm sharing*, which is known to be mandatory in order to implement β-reduction in a time-efficient way. Traditional environment-based machines do not seem to be the right tool for space-efficiency. The reason is that these machines create a sharing annotation—counting as a unit of space—for *every* β-step—that is, for every abstract unit of time. They never garbage collect, because garbage collection is postponable, and ignoring it is safe for time, as its cost is negligible (it is polynomial, if not linear, in the number of β-steps). As a consequence, their use of space is linear in their time usage, which is the worst possible use of space. To study space efficiency, then, there are two possible approaches: either refining environment machines with an explicit treatment of garbage collection, or exploring alternative execution schemas. In this work, we follow the second approach.

Evaluating Without Sharing: Beyond the mainstream approach, there is an alternative execution schema for λ-terms, rooted in Girard's Geometry of Interaction [29] and Abramsky, Jagadeesan, and Malacaria game semantics [1], which does *not* rely on sharing. The *interaction abstract machine* (shortened to IAM), first studied by Mackie [33] and Danos & Regnier [20], [22], evaluates a λ-term without tracing every β-step, thus disentangling time and space. As it is the case for space-efficient Turing machines, the IAM sometimes repeats computations to retrieve unsaved intermediate results—thus treading time for space. Environments are replaced by a *token*, a data structure where the machine saves minimal information, and the repetition of computations is realized via a sophisticated backtracking mechanism (unrelated to backtracking as in control operators or classical logic). The minimal information in the token amounts to tracing the points along the execution history where the IAM may need to backtrack. The entries of the token are *trees* of pointers called *logged positions* in a recent presentation of the IAM by Accattoli, Dal Lago, and Vanoni [5] (called instead *exponential signatures* by Danos & Regnier [21]). Everything else is ignored, in particular the token does not record encountered β-redexes.

Further evidence of the relavance for space of the interaction paradigm, comes from Ghica's *Geometry of Synthesis* [27], [28],

in which the geometry of interaction is seen as a compilation scheme towards circuits, whose computation space is *finite*, and of paramount importance.

The Subtle Complexity of the IAM: These considerations suggest that the IAM is, roughly, bad for time and good for space. Interestingly, the situation is subtler. About time, there are indeed examples showing that the IAM is sometimes exponentially slower than environment machines. The slowdown however is not uniform, as in many cases the IAM exhibits good time performances.

The situation about space is not clear, either. The IAM has been used in the literature for obtaining sub-linear space bounds for functional programs (Dal Lago and Schöpp [19], [36], Mazza [34] and Ghica [27]), something hardly achievable with traditional environment machines. Having a close look at these results, however, one realizes that those bounds rely crucially on some tweaks (restricting to certain λ-terms or extending the language with ad-hoc constructs) and that they do not seem to be achievable on ordinary λ-terms.

A further evidence of the ambiguous space behavior of the IAM is that the folklore conjecture that the IAM space usage is a reasonable space cost model (that is, linearly related to the one of Turing machines) has been circulating among specialists for years, but has never found an answer.

Multi Types and the Time of the IAM: Very recently, there have been advances in the understanding of the subtle time behavior of the IAM. Accattoli, Dal Lago, and Vanoni have shown in [6] how to extract exact time bounds for the IAM from multi types derivations. Interestingly, they use the same types as de Carvalho's, despite the IAM and the KAM computing in very different ways. While the time of the KAM is given by the multiplicity of the multi sets in multi types, they show that the time of the IAM requires to take into account also the size of the involved types. The result also provides a high-level understanding for the time (in)efficiency of the IAM: the inefficiency is proportional to the size of types. Since higher-order types are bigger than first-order ones, the more a program uses higher-order types the more its execution with the IAM is slower than with, say, the KAM.

A. Contributions of the Paper

The aim of this paper is to provide advances in the understanding of the subtle *space* behavior of the IAM. Inspired by the mentioned recent results by Accattoli, Dal Lago, and Vanoni, we introduce a new variant of multi types from which we extract *exact space bounds* for the IAM. To our knowledge, it is the first use of intersection/multi types to measure space.

A key point is that multi types as used in [6]—as well as in the many recent papers to extract quantitative bounds on λ-terms cited above—cannot measure the space consumption of the IAM, as they do not have enough structure. Our work shows that at the type level, indeed, one needs to add a *tree* structure to multi sets, similarly to how measuring time requires switching from idempotent intersections to multisets.

Once the main result is proved, we show how to use it to understand the subtle space behavior of the IAM. Here, the main insight is a negative perspective about the elusive conjecture that the space of the IAM is a reasonable cost model. We also show that the new type system enlightens the key ingredients of the sub-linear space bounds in the literature.

Tree Types: Usually, multi types are structured in two mutually recursive and disjoint layers, linear types and multisets of linear types. Our types also have two layers, but they are no longer disjoint, as multisets can also contain multisets, not only linear types. This way multisets can naturally be seen as *trees*, whose internal nodes are multisets and whose leaves are linear types. Such tree types are very natural, and probably of independent interest. While flattening the tree structure recovers an ordinary multi type, the converse operation cannot be done in a canonical way. This fact shows that tree types add something new, they do not simply express a structure already present in multi types.

Actually, the tree structure reflects information about the token. The intuition is that while the number of leaves of a tree type counts the different uses of a variable/argument—as for multi types—internal nodes on the path to a leaf instead count how many backtracking pointers are stored in the token by the time the IAM gets to that copy.

The Complexity Analysis: Our space bounds are obtained by building on the technique developed by Accattoli, Dal Lago, and Vanoni in [6], which is inherently different from the one by de Carvalho and used in other recent works. The technique in [6] amounts to first defining an auxiliary machine evaluating multi type derivations and showing it bisimilar to the IAM. In this way, we obtain a representation of the states of the IAM run on the type derivation. Then, bounds are extracted from a global analysis based on *weighted* type derivations. We proceed similarly, replacing multi types with tree types, and introducing a new system of weights for space, based on the depth of tree types as trees.

Once the subtle bisimulation is established, our space complexity analysis is extremely simple, and also naturally provides exact bounds. This provides evidence that our types system is not ad-hoc. On the contrary, we believe it unveils a fundamental enrichment of multi types, deserving further studies.

Background on Reasonable Space: In order to discuss the mentioned conjecture about reasonable space, let us provide some background. First of all, studying interesting space complexity classes such as L, requires being able to measure *sub-linear* space. There is a recent result in the literature about reasonable space for the λ-calculus, by Forster, Kunze, and Roth [26], but their cost model—namely, the size of the term—can only measure linear and super-linear space, and thus it is not a solution for the general problem.

Now, showing that a cost model is reasonable requires studying the relationship with another known-to-be reasonable model, typically Turing machines. While for time the delicate direction is the simulation of the λ-calculus into Turing machines, for space the subtle one is the simulation of Turing machines in the λ-calculus, as it is hard to use as little space as a Turing machine.

The iteration of the transition function of Turing machines is a form of tail recursion, which in the λ-calculus is encoded via fixed-point operators. Such operators are also used to represent minimization in the encoding of partial recursive functions. Our insight about the reasonable space conjecture stems from an analysis of fixed-points operators in our type system.

The Elusive Reasonable Space Conjecture: Our contribution here is the fact that the IAM performs poorly when evaluating fixed-point operators, namely using an amount of space which is always at least *linear* in the number of performed recursive calls. This is done by deriving the type (schema) of a specific fixed-point operator in our system, then showing that its use of space is proportional to its use of time.

The specific operator we type is the one used in the encoding of Turing machines in the λ-calculus used by Accattoli and Dal Lago in their study of reasonable time [3], [16], as well as by Forster, Kunze, and Roth in [26]. Seeing it as the natural way of encoding tail recursion, it follows that the IAM space performance is poor with tail recursion, and, in turn, with the natural way of encoding Turing machines.

Summing up, our insight is that a positive answer to the conjecture cannot be done using the standard encoding of Turing machines, which explains the elusiveness of the conjecture.

Trees Density: A way of abstracting away from the issue with fixed-points is to look at how information is organized in the tree structure of tree types, itself reflecting the structure of (logged positions in) the token. When the tree is dense (its height is roughly the log of its number of nodes), then the IAM execution is space-efficient, while when the tree is sparse (height close to the number of nodes) it is inefficient—the type schema for fixed-points is indeed sparse.

Perspectives: The insight about the density of trees allows to re-understand some results in the literature, and opens new perspectives. As we detail in Section VIII, it sheds light on Dal Lago and Schöpp's space bounds for a functional language [19], [36], as well as on the encoding of sub-linear space computations in the λ-calculus by Mazza [34].

Proofs: Proofs are in the associated technical report [7].

II. 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.

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

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 (that is, the number of !-boxes in linear logic terminology) the hole lies in.

LEVELED CONTEXTS

$$\begin{array}{rcl} C_0 & ::= & \langle\cdot\rangle \;\big|\; \lambda x.C_0 \;\big|\; C_0 t; \\ C_{n+1} & ::= & C_{n+1}t \;\big|\; \lambda x.C_{n+1} \;\big|\; tC_n. \end{array}$$

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 $\to_{wh}$, defined as follows:

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

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

Abstract Machines Glossary: In this paper, an *abstract machine* $\mathrm{M} = (s, \to)$ is a transition system $\to$ over a set of states, noted s. The machine considered in this paper moves 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 $C\langle u\rangle = t$. States are composed by 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—t is always implicitly considered closed, without further mention. A state is *final* if no transitions apply.

A *run* $\rho : s \to^* s'$ is a possibly empty sequence of transitions, whose length is noted $|\rho|$. 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.

III. 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, Dal Lago, and Vanoni in [5]—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, which is closer to the practice of functional programming. Keep in mind that the λIAM is an unusual machine, and that finding it hard to grasp is normal. Also, in [5] 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 environment machines such as Krivine's, it looks for the head variable of a term. The peculiarity of the λIAM is that it does not store the encountered β-redexes in an environment. When it finds the head variable, 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

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

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	$\to_{\mathsf{var}}$	$\lambda x.D_n\langle x\rangle$	$\underline{C}$	L	$(x, \lambda x.D_n, L_n)\cdot T$
$\underline{\lambda x.D_n\langle x\rangle}$	C	L	$(x, \lambda x.D_n, L_n)\cdot T$	$\to_{\mathsf{bt2}}$	x	$\underline{C\langle\lambda x.D_n\rangle}$	$L_n\cdot L$	T
t	$\underline{C\langle\langle\cdot\rangle u\rangle}$	L	$\bullet\cdot T$	$\to_{\bullet 3}$	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	$l\cdot T$	$\to_{\mathsf{arg}}$	$\underline{u}$	$C\langle t\langle\cdot\rangle\rangle$	$l\cdot L$	T
t	$\underline{C\langle u\langle\cdot\rangle\rangle}$	$l\cdot L$	T	$\to_{\mathsf{bt1}}$	$\underline{u}$	$C\langle\langle\cdot\rangle t\rangle$	L	$l\cdot T$

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

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 data structures of the λIAM are 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. It travels according to 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. Note 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: The transitions of the λIAM are in Fig. 1. Their union is noted $\to_{\lambda\text{IAM}}$. 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. A key point is that navigation is done locally, moving only between adjacent positions[1]. Intuitively, the machine evaluates the term t until the head abstraction of the head normal form is found (more explanations below). The transitions realize three entangled mechanisms.

Mechanism 1: Search Up to β-Redexes: Note that $\to_{\bullet 1}$ 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 $\to_{\bullet 2}$ 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 $\to_{\bullet 3}$ and $\to_{\bullet 4}$ do the same during the $\uparrow$ phase.

Let us illustrate this mechanism with an example: the first steps of evaluation on the term $\mathsf{I}((\lambda x.xx)\mathsf{I})$, where I is the identity combinator. One can notice that the λIAM traverses one β-redexes without altering the token, that is empty both at the beginning and at the end.

Sub-term	Context	Log	Tape	Dir
$\underline{(\lambda z.z)((\lambda x.xx)\mathsf{I})}$	$\langle\cdot\rangle$	ϵ	ϵ	$\downarrow$
$\to_{\bullet 1}$ $\underline{\lambda z.z}$	$\langle\cdot\rangle((\lambda x.xx)\mathsf{I})$	ϵ	$\bullet$	$\downarrow$
$\to_{\bullet 1}$ $\underline{z}$	$(\lambda z.\langle\cdot\rangle)((\lambda x.xx)\mathsf{I})$	ϵ	ϵ	$\downarrow$

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 $\to_{\mathsf{var}}$ 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

[1] Transitions $\to_{\mathsf{var}}$ and $\to_{\mathsf{bt2}}$ might not look local, as they jump from a bound variable occurrence to its binder, and viceversa. If λ-terms are represented by implementing occurrences as pointers to their binders, as in the proof net representation of λ-terms—upon which some concrete implementation schemes are based, see [2]—then they are local.

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 $\to_{\mathsf{arg}}$ 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 z and looks for its argument in $\uparrow$ mode. When it has been found, the direction turns to $\downarrow$ again and the process continues as before: first the head variable is found and then the machine looks for its argument. Let us set $l_z := (z, (\lambda z.\langle\cdot\rangle)((\lambda x.xx)\mathsf{I}), \epsilon)$, $l_{\langle\cdot\rangle x} := (x, \lambda x.\langle\cdot\rangle x, \epsilon)$ and $l_y := (y, \lambda y.\langle\cdot\rangle, \epsilon)$.

	Sub-term	Context	Log	Tape	Dir
	$\underline{z}$	$(\lambda z.\langle\cdot\rangle)((\lambda x.xx)\mathsf{I})$	ϵ	ϵ	$\downarrow$
$\to_{\mathsf{var}}$	$\lambda z.z$	$\underline{\langle\cdot\rangle((\lambda x.xx)\mathsf{I})}$	ϵ	l_z	$\uparrow$
$\to_{\mathsf{arg}}$	$\underline{(\lambda x.xx)\mathsf{I}}$	$\mathsf{I}\langle\cdot\rangle$	l_z	ϵ	$\downarrow$
$\to_{\bullet 1}$	$\underline{\lambda x.xx}$	$\mathsf{I}(\langle\cdot\rangle\mathsf{I})$	l_z	$\bullet$	$\downarrow$
$\to_{\bullet 2}$	$\underline{xx}$	$\mathsf{I}((\lambda x.\langle\cdot\rangle)\mathsf{I})$	l_z	ϵ	$\downarrow$
$\to_{\bullet 1}$	$\underline{x}$	$\mathsf{I}((\lambda x.\langle\cdot\rangle x)\mathsf{I})$	l_z	$\bullet$	$\downarrow$
$\to_{\mathsf{var}}$	$\lambda x.xx$	$\underline{\mathsf{I}(\langle\cdot\rangle(\lambda y.y))}$	l_z	$l_{\langle\cdot\rangle x}\cdot\bullet$	$\uparrow$
$\to_{\mathsf{arg}}$	$\underline{\lambda y.y}$	$\mathsf{I}((\lambda x.xx)\langle\cdot\rangle)$	$l_{\langle\cdot\rangle x}\cdot l_z$	$\bullet$	$\downarrow$
$\to_{\bullet 2}$	$\underline{y}$	$\mathsf{I}((\lambda x.xx)(\lambda y.\langle\cdot\rangle))$	$l_{\langle\cdot\rangle x}\cdot l_z$	ϵ	$\downarrow$
$\to_{\mathsf{var}}$	$\lambda y.y$	$\underline{\mathsf{I}((\lambda x.xx)\langle\cdot\rangle)}$	$l_{\langle\cdot\rangle x}\cdot l_z$	l_y	$\uparrow$

Mechanism 3: Backtracking: It is started by transition $\to_{\mathsf{bt1}}$. 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 $\to_{\mathsf{bt2}}$ is fired.

The $\downarrow$-phase and the logged position on the tape mean that the λIAM is backtracking. During backtracking, the machine is not looking for the head variable of the current code $\lambda x.t$, 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 guarantees that the logged position on the tape always contains a position relative to the active abstraction.

In our example, a backtracking phase starts when the λIAM looks for the argument of y. Since $\lambda y.y$ has been virtually substituted for x, its argument is actually the second occurrence of x. Backtracking retrieves the variable which a term was virtually substituted for.

	Sub-term	Context	Log	Tape	Dir
	$\lambda y.y$	$\underline{\mathsf{I}((\lambda x.xx)\langle\cdot\rangle)}$	$l_{\langle\cdot\rangle x}\cdot l_z$	l_y	$\uparrow$
$\to_{\mathsf{bt1}}$	$\underline{\lambda x.xx}$	$\mathsf{I}(\langle\cdot\rangle\mathsf{I})$	l_z	$l_{\langle\cdot\rangle x}\cdot l_y$	$\downarrow$
$\to_{\mathsf{bt2}}$	x	$\underline{\mathsf{I}((\lambda x.\langle\cdot\rangle x)\mathsf{I})}$	l_z	l_y	$\uparrow$
$\to_{\mathsf{arg}}$	$\underline{x}$	$\mathsf{I}((\lambda x.x\langle\cdot\rangle)\mathsf{I})$	$l_y\cdot l_z$	ϵ	$\downarrow$

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 second occurrence of I, is found. We set $l_{x\langle\cdot\rangle} := (x, \lambda x.x\langle\cdot\rangle, l_y)$.

	Sub-term	Context	Log	Tape	Dir
	$\underline{x}$	$\mathsf{I}((\lambda x.x\langle\cdot\rangle)\mathsf{I})$	$l_y\cdot l_z$	ϵ	$\downarrow$
$\to_{\mathsf{var}}$	$\lambda x.xx$	$\underline{\mathsf{I}(\langle\cdot\rangle(\lambda y.y))}$	l_z	$l_{x\langle\cdot\rangle}$	$\uparrow$
$\to_{\mathsf{arg}}$	$\underline{\lambda y.y}$	$\mathsf{I}((\lambda x.xx)\langle\cdot\rangle)$	$l_{x\langle\cdot\rangle}\cdot l_z$	ϵ	$\downarrow$

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 and the current position (t, C) form a logged position, *i.e.* the length of L is exactly the level of the code context C[2]. This fact guarantees that the λIAM never gets stuck because the log is too short for transitions $\to_{\mathsf{var}}$ and $\to_{\mathsf{bt1}}$ 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 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 III.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 [5].

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 $[\![\cdot]\!]_{\lambda\mathrm{IAM}}$ 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 [5]. For the sake of simplicity, here we restrict to Closed CbN. The λIAM semantics then reduces to observing termination: $[\![t]\!]_{\lambda\mathrm{IAM}}$ is defined if and only if weak head reduction terminates on t. Therefore, we avoid discussing semantics and only study termination. We say that the *λIAM implements Closed CbN* when its execution from the initial state s_t reaches a final state if and only if $\to_{wh}$ terminates on t, for every closed term t.

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

$$\frac{}{x : [A] \vdash x : A}\ \text{T-VAR} \qquad \frac{\Gamma, x : T \vdash t : A}{\Gamma \vdash \lambda x.t : T \to A}\ \text{T-}\lambda \qquad \frac{\Gamma \vdash t : T \to A \quad \Delta \vdash u : T}{\Gamma \uplus \Delta \vdash tu : A}\ \text{T-@}$$

$$\frac{}{\Gamma \vdash \lambda x.t : \star}\ \text{T-}\lambda_\star \qquad \frac{\Gamma_i \vdash t : G_i \quad 1 \leq i \leq n}{[\uplus_{i=1}^n \Gamma_i] \vdash t : [G_1, .., G_n]}\ \text{T-MANY} \qquad \frac{}{\vdash t : [\cdot]}\ \text{T-NONE}$$

Figure 2: The tree type system.

Theorem III.2 ([5]). *The λIAM implements Closed CbN.*

λIAM Space Consumption: The space needed to represent a λIAM state is given by the following definition (the meta-variable Γ to denote either a tape T or a log L):

$$|(t, C, L, T, d)|_{\mathsf{sp}} := |L|_{\mathsf{sp}} + |T|_{\mathsf{sp}}$$

$$|(x, D, L')|_{\mathsf{sp}} := \mathsf{X} + |L'|_{\mathsf{sp}} \qquad |\epsilon|_{\mathsf{sp}} := 0$$

$$|l \cdot \Gamma|_{\mathsf{sp}} := |l|_{\mathsf{sp}} + |\Gamma|_{\mathsf{sp}} \qquad |\bullet \cdot T|_{\mathsf{sp}} := 1 + |T|_{\mathsf{sp}}$$

The value of the unknown X is simply the size of a pointer to a subterm of the term under evaluation, *i.e.* $\mathsf{X} = \log |C\langle t \rangle|$. Then, we are able to define the space of a λIAM run by taking the maximum size of the states reached during the run.

Definition III.3. *Let $\rho : s_0 \to^*_{\lambda\text{IAM}} s$ be a λIAM run. Then,*

$$|\rho|_{\mathsf{sp}} := \max_{s' \in \rho} |s'|_{\mathsf{sp}}$$

It is worth noticing what happens in the case of diverging computations. In principle, two cases could occur: either the space consumption is finite, or it is infinite. Actually, it is easy to prove that the first case is not possible.

Proposition III.4. *Let ρ be an infinite λIAM run. Then $|\rho|_{\mathsf{sp}} = \infty$.*

IV. Tree (Intersection) Types

Here we introduce a type system that we shall use to measure the space used by λIAM runs.

From Intersections Types to Tree Types: The framework that we adopt is the one of intersection types, with three tweaks:

1) *No idempotency*: we use the non-idempotent variant, where the intersection type $A \wedge A$ is not equivalent to A, and with strong 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 multisets, which is why these types are sometimes called *multi types* and an intersection $A \wedge B \wedge A$ is rather noted $[A, B, A]$.
2) *Nesting/tree shape*: multi types are usually defined by two mutually dependent layers, a linear one containing ground types and (linear) arrow types, and the multiset level containing linear types. Here we also have two layers, except that we allow multisets to also contain multisets, thus we can have $[A, [[B, B], A, [A]], A, B]$. A nested multiset is a *tree* whose leaves are linear types and whose internal nodes are nested multisets.
3) *No commutativity*: we also consider *non-commutative* tree types. Removing commutativity turns multisets into lists, or sequences, and trees into ordered trees. Removing commutativity is an inessential tweak. Our study does not depend on the ordered structure, we shall only need bijections between multisets, to describe the reformulation of the λIAM on type derivations, and these bijections are just more easily managed if commutativity is removed. This *rigid* approach is also used by Tsukada, Asada, and Ong [37] and Mazza, Pellissier, and Vial [35]. The inessential aspect is stressed by referring to our types as to *tree types*, rather than as to *ordered tree types*, despite adopting the ordered variant.

Basic Definitions: As for multi types, there are two mutually defined layers of types, *linear types* and *tree types*.

$$\begin{array}{rrcll} \text{LINEAR TYPES} & A, A' & ::= & \star \mid T \to A & \\ \text{TREE TYPES} & T, T' & ::= & [G_1, \ldots, G_n] & n \geq 0 \\ \text{(GENERIC) TYPES} & G, G' & ::= & A \mid T & \end{array}$$

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 $T \to A$ can have a tree type only on the left. About trees, since commutativity is ruled out, we have, for instance, that $[A, A'] \neq [A', A]$. Note that the empty tree type/sequence is a valid type, which is noted $[\cdot]$. The concatenation of two sequences T and T' is noted $T \uplus T'$. We use $|T|$ for the length of T as a sequence, that is, $|[G_1, .., G_n]| := n$.

Type judgments have the form $\Gamma \vdash t : G$, where Γ is a type environment, defined below. Type derivations are noted π and we write $\pi \triangleright \Gamma \vdash t : G$ for a type derivation π of ending judgment $\Gamma \vdash t : G$. Type environments, ranged over by Γ, Δ are total maps from variables to tree types such that only finitely many variables are mapped to non-empty tree types, and we write $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ if $dom(\Gamma) = \{x_1, \ldots, x_n\}$—note that type environments *are* commutative, what is non-commutative is the sequence constructor $[\cdot]$, only. Given two type environments Γ, Δ, we use $\Gamma \uplus \Delta$ for the type environment assigning to every variable x the list $\Gamma(x) \uplus \Delta(x)$.

The typing rules are in Fig. 2. With respect to the literature, the difference is in rule T-MANY. There are two differences. The first one is the already mentioned fact that premises may assign both linear types and tree types, while the literature usually only allows linear types. The second one is that the rule surrounds $\Gamma := \uplus_{i=1}^n \Gamma_i$ with an additional nesting level—the notation $[\Gamma]$ standing for the type environment $x_1 : [T_1], \ldots, x_n : [T_n]$ if $\Gamma = x_1 : T_1, \ldots, x_n : T_n$.

A Small Example: We show an instance of the rule T-MANY in the delicate case in which the premises contain

the same free variable x.

$$\frac{x : [A_1] \vdash t : G_1 \quad x : [A_2] \vdash t : G_2}{x : [[A_1, A_2]] \vdash t : [G_1, G_2]} \text{ T-MANY}$$

In particular, please note that first $[A_1]$ and $[A_2]$ are joined, and then they are surrounded by an additional nesting level. The other option would have been $x : [[A_1], [A_2]]$, but it is not what T-MANY does.

Leaves Extraction and Leaf Contexts: Every tree type T induces the sequence $\underline{T}$— equivalently, the flat tree type—of its leaves, defined by the following *leaves extraction* operation.

$$[\cdot]^\ell := [\cdot] \quad ([A] \uplus T)^\ell := [A] \uplus T^\ell \quad ([T'] \uplus T)^\ell := T'^\ell \uplus T^\ell$$

We shall describe the leaves of a tree type also via a notion of leaf context.

$$\text{LEAF CTXS} \quad \mathbb{L} ::= [G_1, .., \langle\cdot\rangle, .., G_n] \mid [G_1, .., \mathbb{L}, .., G_n]$$

If $T^\ell = [A_1, .., A_n]$ then for every A_i there is a leaf context $\mathbb{L}^i$ such that $T = \mathbb{L}^i\langle A_i\rangle$. Therefore, we shall use the notation $T = \mathbb{L}^i\langle A\rangle$, or even simply $T^i = A$, to say that the linear type A is the i-th leaf of T.

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.

Lemma IV.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 $|\Gamma(x)^\ell|$ 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 $\Gamma(x)^\ell = [A_1, \ldots, A_k]$ 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 Closed CbN. Moreover, every term that is Closed CbN normalizable can be typed with $\star$. The same characterization holds with tree types, following the standard recipe[3] for multi types, without surprises. See the technical report [7] for details.

Theorem IV.2 (Correctness and completeness of tree types for Closed CbN)**.** *A closed term t is Closed CbN normalizable if and only if there exists a tree type derivation $\pi \triangleright \vdash t : \star$.*

Relationship with Multi Types: The leaves extraction operation can easily be extended to a flattening function turning a tree type into a multi type. Flattening can also be extended to derivations, by collapsing trees of T-MANY rules into the more traditional rule for multi sets that does not *nodify* the type context. In this way, one obtains a forgetful transformation, easily defined by induction on derivations. A converse *lifting* transformation, however, cannot be defined by induction on derivations—it is unclear how to define it on applications. This fact is evidence that tree types are strictly richer than multi types, because the tree structure cannot be inferred from the multiset one.

[3]Namely, substitution lemma plus subject reduction for correctness, and anti-substitution lemma, subject expansion, and typability of all normal forms for completeness (here trivial, because all normal forms are typed by T-$\lambda_\star$).

V. THE TREE IAM

This section introduces a machine evaluating type derivations, the *Tree IAM*, or *TIAM*, that mimics the λIAM directly on top of a type derivation π. It is the key tool that we shall use to measure the space cost of λIAM runs. The TIAM is a very minor variation over the similar SIAM machine evaluating type derivations for sequence types by Accattoli, Dal Lago, and Vanoni in [6]. This and the next section mostly just recall and adapt notions and results from that paper.

Preamble about Duplications and (No) Logs: β-reducing a λ-term (potentially) duplicates arguments, whose different copies may be used differently, typically being applied to different further arguments. The λIAM never duplicate arguments, but has nonetheless to distinguish different uses of the same piece of code. This is why it 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 key point of multi/tree type derivations is that duplication is explicitly accounted for, *in advance*, by multisets/trees: all arguments come with as many type derivations as the times they are duplicated during evaluation. With tree types, the number of copies is the number of leaves of the tree. Note that a multi/tree type derivation may be way bigger than the term itself, while this is not possible with, say, simple types.

The intuition behind the TIAM is that the walk over the code done by the λIAM can be rephrased and simplified on multi/tree type derivations, because the mechanism of logs—needed to distinguish different copies of arguments—is no longer needed, since all copies are already there: simple positions in the type derivation (not in the term!) are informative enough.

The TIAM: On the one hand, the TIAM is simpler than the λIAM because it has no logs, on the other hand, it is more technical to define because tree type derivations are less easily manipulated than λ-terms. The underlying idea however is simple. The TIAM moves over a fixed type derivation $\pi \triangleright \vdash t : \star$, to be thought as the code, following the occurrence of $\star$ in the final judgment through π, according to the transitions in Fig. 3. We shall now explain every involved concept.

The position of the machine is given by an occurrence of a type judgment[4] J of π. As the λIAM, the TIAM has two possible directions, noted $\downarrow$ and $\uparrow$[5]. In direction $\uparrow$ the machine looks at the rule above the focused judgment, in direction $\downarrow$ at the rule below. The only "data structure"—encoding the tape of the λIAM, as we shall explain—is a type context $\mathbb{A}$ 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 $\mathbb{A}$ and $\mathbb{T}$ with type environments Γ):

[4]A judgment may occur repeatedly in a derivation, which is why we talk about *occurrences* of judgments. To avoid too many technicalities, however, we usually just write the judgment, leaving implicit that we refer to an occurrence of that judgment.

[5]Type derivations are upside-down wrt to the term structure, then direction $\downarrow$ of the λIAM becomes here $\uparrow$, and $\uparrow$ is $\downarrow$.

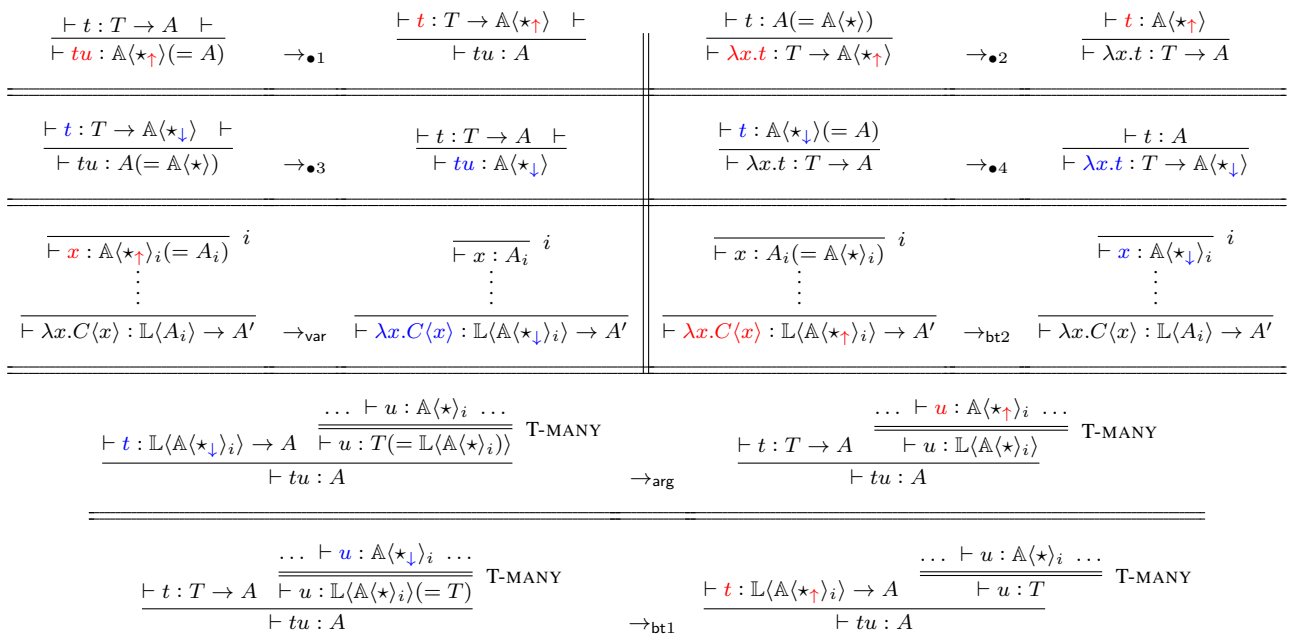

Figure 3: The transitions of the Tree IAM (TIAM).

$$\begin{array}{rl} \text{LINEAR CTXS} & \mathbb{A} ::= \langle\cdot\rangle \mid T \to \mathbb{A} \mid \mathbb{T} \to A \\ \text{TREE CTXS} & \mathbb{T} ::= [G_1, .., \mathbb{G}, .., G_n] \\ \text{TYPE CTXS} & \mathbb{G} ::= \mathbb{A} \mid \mathbb{T} \end{array}$$

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

Intuitions about Positions (and Logs): The intuition is that the active position (t, C_n) of a λIAM state corresponds to the judgment occurrence J in the TIAM, or, more precisely, to its position in the type derivation π. The sub-term t is exactly the term typed by J. The context C_n is exactly the context giving the term $C_n\langle t\rangle$ typed by the final judgment of π. The level n of the context C_n of the active position counts the number of arguments in which the hole of C_n is contained. On the type derivation, each such argument is associated to a T-@ rule having the current judgment J in its right sub-derivation. Last, note that in the λIAM the current log L_n has length equal to n. We shall see in the next section that one can recover the log L_n applying an extraction process to the T-@ rules found descending from J towards the final judgment.

Transitions: The TIAM starts on the final judgment of π, with empty type context $\mathbb{A} = \langle\cdot\rangle$ and direction $\uparrow$. It moves from judgment to judgment, following occurrences of $\star$ around π. To specify the transitions, we use the leaf contexts defined in the previous section.

The transitions are in Fig. 3, their union is noted $\to_{\text{TIAM}}$. We now explain them one by one—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 focuses 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 $T \to A$ to $\lambda x.t$, and the type context is $T \to \mathbb{A}$, that is, it selects an occurrence of $\star$ in the target type $A = \mathbb{A}\langle\star\rangle$. The transition then simply moves to the judgment above, stripping down the type context to $\mathbb{A}$, 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: the extra premise $\vdash$ simply denotes the right premise of the T-@ rule 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 $T \to A'$ isolating $\star$ inside the i-th leaf A_i of the tree type T. The transition then moves the state of the TIAM to the i-th leaf of the tree of T-MANY rules on 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 IV.1. Consider a T-λ rule occurrence whose right-hand type of the conclusion is $T \to A'$. The premise has shape $\Gamma, x : T \vdash t : A'$, and by the lemma there is a bijection between the leaves in T and the axioms on x, respecting the order in T^ℓ. The left side of $\to_{\mathsf{bt2}}$ focuses on the i-th leaf A_i of T and the TIAM 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 $\to_{\mathsf{var}}$ 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 TIAM reaches a T-$\lambda_\star$ rule, it is in a final state. Exactly as the λIAM, the TIAM is bi-deterministic.

Proposition V.1. *The TIAM is bi-deterministic for each type derivation $\pi \triangleright \vdash t : \star$.*

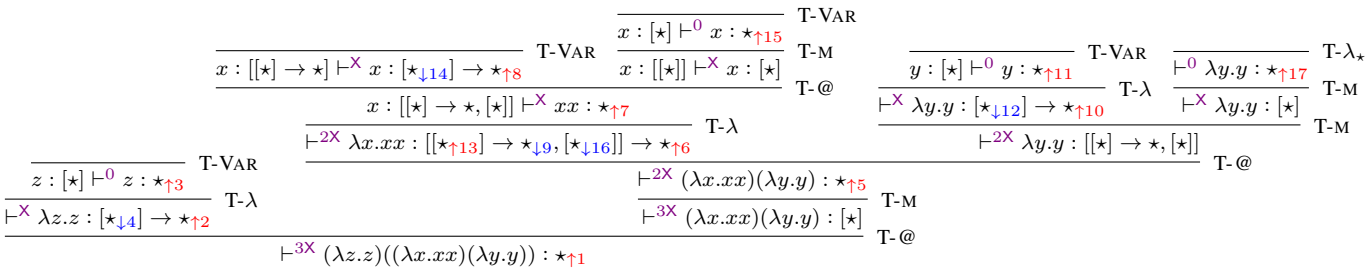

Figure 4: An example of TIAM execution.

An example: In Fig. 4 we present the very same example analyzed in Section III. 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. 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 III. Please note that judgments are decorated with weights (such as X), which shall be introduced only in Sect. VII, in order to later provide an example of decoration—they can be safely ignored for now.

VI. The λIAM and TIAM Bisimulation

The aim of this section is to explain the strong bisimulation between the TIAM and the λIAM, that is essentially the same between the SIAM and the λIAM studied in [6]. A striking point of the TIAM is that it does not have the log nor the tape. They are encoded in the position of the judgment occurrence J and in the type context $\mathbb{A}$ of its states, as we shall show.

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 TIAM 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 leaf (in the tree of sub-derivations) typing u_k in which J is located. Note that the λIAM manipulates the log only via transitions $\rightarrow_{\mathsf{arg}}$ and $\rightarrow_{\mathsf{bt1}}$, that on the TIAM 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 $\rightarrow_{\mathsf{var}}$ and $\rightarrow_{\mathsf{bt1}}$, and removes them using $\rightarrow_{\mathsf{arg}}$ and $\rightarrow_{\mathsf{bt2}}$. By looking at Fig. 3, it is clear that there is a logged position on the λIAM tape for every type sequence of the flattening of T in which it lies the hole $\langle\cdot\rangle$ of the current type context $\mathbb{A}$ of the TIAM.

Extracting λIAM States: These ideas are used to extract from every TIAM state s a λIAM state $\mathsf{ext}(s)$ in a quite technical way. In particular, the extraction process retrieves a log $L_{\mathsf{ext}}(s)$ from the judgement J of s and a tape $T_{\mathsf{ext}}(s)$ from the type context $\mathbb{A}$ of s, using a sophisticated *T-exhaustible invariant* of the TIAM to retrieve the exact shape of the logged positions in $L_{\mathsf{ext}}(s)$ and $T_{\mathsf{ext}}(s)$.

Let us give a high-level description of how extraction works. The invariant is based on the pairing of every TIAM state s with a set of *test states*, some coming from the judgment J of s, called *judgment tests*, and some coming from the type context $\mathbb{A}$, called *type (context) tests*. The invariant guarantees a certain recursive property of each test state. The extraction process uses this property to extract a logged position $l_{s'}$ from each test state s' of s.

Given a TIAM state $s = (\pi, J, \mathbb{A}, d)$, its judgment tests are associated to the T-@ rules having J in their right sub-derivation. Their extractions give logged positions $l_1 \cdot \ldots \cdot l_n$ forming the extracted log $L_{\mathsf{ext}}(s)$, following the correspondence described above.

Type tests are associated to the leaf contexts surrounding the hole of $\mathbb{A}$. The extraction of the tape $T_{\mathsf{ext}}(s)$ from $\mathbb{A}$ is done according to the following schema:

$$\begin{array}{lcl} T_{\mathsf{ext}}(\langle\cdot\rangle) & := & \epsilon \\ T_{\mathsf{ext}}(T \rightarrow \mathbb{A}) & := & \bullet \cdot T_{\mathsf{ext}}(\mathbb{A}) \\ T_{\mathsf{ext}}(\mathbb{L}\langle\mathbb{A}\rangle \rightarrow A') & := & l_{\mathsf{ext}}(s_{\mathbb{L}}) \cdot T_{\mathsf{ext}}(\mathbb{A}) \end{array}$$

where $s_{\mathbb{L}}$ is the state test associated to the leaf context $\mathbb{L}$.

For lack of space, and because this is essentially identical to what is done in [6] for the SIAM, the technical development is in the technical report [7]. The extraction process induces a relation $s \simeq_{\mathsf{ext}} \mathsf{ext}(s)$ that is easily proved to be a strong bisimulation between the TIAM and the λIAM.

Proposition VI.1 (TIAM and λIAM bisimulation)**.** *Let t a closed term and $\pi \triangleright \vdash t : \star$ a tree type derivation. Then $\simeq_{\mathsf{ext}}$ is a strong bisimulation between TIAM states on π and λIAM states on t. Moreover, if $s_\pi \simeq_{\mathsf{ext}} s_\lambda$ then s_π is TIAM reachable if and only if s_λ is λIAM reachable.*

The *moreover* part of the above statement hints at a bijection between *all* the states in π and reachable λIAM states. However, there still could be the possibility that some of the states in π are not reachable. This is actually *not* the case, and the technical development is in [7].

Proposition VI.2. *Let t a closed term and $\pi \triangleright \vdash t : \star$ a tree type derivation. Then every state of π is reached exactly once.*

$$\frac{}{x : [A] \overset{\|A\|}{\vdash} x : A}\ \text{T-VAR} \qquad \frac{\Gamma, x : T \overset{w}{\vdash} t : A}{\Gamma \overset{\max\{w, \|T \to A\|\}}{\vdash} \lambda x.t : T \to A}\ \text{T-}\lambda \qquad \frac{\Gamma \overset{w}{\vdash} t : T \to A \quad \Delta \overset{v}{\vdash} u : T}{\Gamma \uplus \Delta \overset{\max\{w,v\}}{\vdash} tu : A}\ \text{T-@}$$

$$\frac{}{\Gamma \overset{0}{\vdash} \lambda x.t : \star}\ \text{T-}\lambda_\star \qquad \frac{\Gamma_i \overset{w_i}{\vdash} t : G_i \quad 1 \le i \le n}{[\uplus_{i=1}^n \Gamma_i] \overset{\mathsf{X}+\max_i\{w_i\}}{\vdash} t : [G_1, .., G_n]}\ \text{T-MANY} \qquad \frac{}{\overset{0}{\vdash} t : [\cdot]}\ \text{T-NONE}$$

Figure 5: The tree type system with weights.

VII. Measuring the Space of Interactions

This section contains the main contribution of the article: it gives a way of measuring the space consumed by the complete λIAM run on the term t via a quantitative analysis of the tree type derivation for t. We proceed in two steps.

1) *The space of single extracted states*: given a TIAM state $s = (\pi, J, \mathbb{A}, d)$, we show how to measure the space of the extracted λIAM state $\mathsf{ext}(s)$ from π, J, and $\mathbb{A}$.
2) *The space of the whole execution*: we refine tree type derivations adding *weights* on judgments, and show that the weight of the final judgment coincides with the maximum space consumption over all extracted states, that is, along the whole λIAM execution.

The Undetermined Pointers Size X*:* A technical point common to both parts is that the quantitative study of tree types derivations is relative to an undetermined value X. The reason for using X is that our space analyses have both local and global components. Locally, we count how many occurrences of $\bullet$ and how many logged positions are involved in a state (for step 1) or in all states in and above that judgment (for step 2). The global component comes from the fact that all logged positions of the λIAM, independently of where they arise, are implemented via pointers to the *global* code. Essentially, X is meant to be replaced, at the very end of both analyses, by the size of pointers to the λIAM global code, that is, by $\log |t|$, where t is the term typed in the final judgment of the type derivation π. Therefore, locally our measures shall include X, which shall substituted at the end with $\log |t|$.

A. The Space of Single Extracted States

Trees and the Size of Extracted Logged Positions: Basically, given a TIAM state $s = (\pi, J, \mathbb{A}, d)$, the size of logged positions in $\mathsf{ext}(s)$ is obtained by counting X for

- *Extracted tape*: every sequence constructor $[\cdot]$ surrounding the hole $\langle\cdot\rangle$ in $\mathbb{A}$;
- *Extracted log*: every T-MANY rule on the path from J to the final judgment of π.

Clearly, it is the newly introduced tree structure that allows to measure the size of extracted logged positions, as expected.

First, we define a size of type contexts meant to measure the size of the extracted tapes.

Definition VII.1 (Branch size of type contexts/extracted tapes)**.** *Let $s = (\pi, J, \mathbb{A}, d)$ be a TIAM state. The branch size $|\cdot|_\mathsf{b}$ for type contexts is defined as follows:*

$$|\langle\cdot\rangle|_\mathsf{b} := 0 \qquad |T \to \mathbb{A}|_\mathsf{b} := 1 + |\mathbb{A}|_\mathsf{b}$$
$$|\mathbb{T} \to A|_\mathsf{b} := |\mathbb{T}|_\mathsf{b} \qquad |[G_1, .., \mathbb{G}, .., G_n]|_\mathsf{b} := \mathsf{X} + |\mathbb{G}|_\mathsf{b}$$

Let us interpret the branch size with respect to the tape extraction schema of the previous section. The $+1$ in the clause for $T \to \mathbb{A}$ is there to count $\bullet$. The clause for sequences instead gives $|\mathbb{L}| = n \cdot \mathsf{X}$ if the hole has height n in the leaf context $\mathbb{L}$ seen as a tree—whence the name *branch size*.

Then, we define a branch size for judgments, meant to measure the size of extracted logs, and a branch size for states.

Definition VII.2. *Let $s = (\pi, J, \mathbb{A}, d)$ be a TIAM state.*

- Branch size of judgments/extracted logs*: let n be the number of* T-MANY *rules encountered descending from J to the root of π. Then $|J|_\mathsf{b} := n \cdot \mathsf{X}$.*
- Branch size of TIAM states*: $|s|_\mathsf{b} := |\mathbb{A}|_\mathsf{b} + |J|_\mathsf{b}$.*

We prove that the defined branch sizes do correspond to their intended meanings, that is, the branch sizes of extracted logs and tapes, showing that the size of TIAM states captures the space size of the extracted λIAM state.

Proposition VII.3 (Space of Single Extracted States)**.** *Let $s = (\pi, J, \mathbb{A}, d)$ be a reachable TIAM state. Then $|\mathbb{A}|_\mathsf{b} = |T_{\mathsf{ext}}(s)|_{\mathsf{sp}}$ and $|J|_\mathsf{b} = |L_{\mathsf{ext}}(s)|_{\mathsf{sp}}$, and thus $|s|_\mathsf{b} = |\mathsf{ext}(s)|_{\mathsf{sp}}$. Moreover,*

1) *if $L_{\mathsf{ext}}(s) = l_1..l_n$, and let h_i be the number of* T-MANY *rules of the i^{th}* T-MANY *rule tree found descending from J to the root of π, then $|l_i|_{\mathsf{sp}} = h_i$;*
2) *for each extracted tape position l, i.e. for each $\mathbb{G}$ such that $\mathbb{A} = \mathbb{G}\langle\mathbb{L}\langle\mathbb{A}'\langle\star\rangle\rangle \to A\rangle$, then $|l|_{\mathsf{sp}} = |\mathbb{L}| \cdot \mathsf{X}$.*

The Need for Tree Types: A subtle point is that the tree structure of types is not needed in order to define the extraction process—indeed, λIAM states are extracted from *sequence* type derivations in [6]. Extraction is an indirect process—a sort of logical relation—whose functioning is guaranteed by an invariant (the T-exhaustible invariant in [7]). The process does not describe explicitly the shape of the extracted logged positions, it only guarantees that adequate logged positions exist. Without tree types, the structure of multi types derivations somehow encodes enough information to retrieve $\mathsf{ext}(s)$, but how many logged positions are involved can be discovered only by unfolding the whole extraction process, the information is not encoded into the types themselves.

There is a further subtlety. Tree types trace the *number* of pointers as precisely described by the *moreover* part of Prop. VII.3, but do not describe the internal structure of logged positions. Given a TIAM state s, we can easily know the length of $L_{\mathsf{ext}}(s)$ and $T_{\mathsf{ext}}(s)$, and know the number of pointers to implement each logged position l in them, which is enough to measure space. The internal structure of l, however, cannot be read from tree types. Again, it is determined only by unfolding the whole extraction process.

B. The Space of the Whole Execution

Type Weights: To obtain the space cost of the whole execution we endow tree types derivations with *weights*[6]. In turn, we first have to define a notion of weight for types. The intuition is that we are taking the max of the branch size for type contexts $\mathbb{G}$ used above, over all the ways of writing a type G as $\mathbb{G}\langle\star\rangle$, as confirmed by the associated lemma.

$$\|\star\| := 0 \qquad \|T \to A\| := \max\{\|T\|, \|A\| + 1\}$$
$$\left\|[G_1, .., G_n]\right\| := \mathsf{X} + \max_i\{\|G_i\|\}$$

Lemma VII.4. *Let G be a type. Then $\|G\| = \max_{\mathbb{G}\mid G=\mathbb{G}\langle\star\rangle} |\mathbb{G}|_{\mathsf{b}}$.*

Note that, via the space of single extracted states (Prop. VII.3), the previous lemma states that the size of G is the maximum space of all the tapes extracted from TIAM states over a same judgment $\Gamma \vdash t : G$.

Judgements and Derivations Weights: Weights are extended to judgments in Fig. 5, and the weight of a derivation is the weight of its final judgment. The idea is that the weight w of a weighted judgment $J = \Gamma \overset{w}{\vdash} t : G$ gives the maximum space of all the states over J and—crucially—above J.

Now, we prove that the weight of a judgment J is greater than the maximum size of the tape of the states in its derivation.

Lemma VII.5 (Judgment weights bound extracted tapes)**.** *Let $\pi : \Gamma \overset{w}{\vdash} t : G$ be a weighted derivation and $\mathcal{J}$ be the set of all the judgments occurring in π. Then $w \geq \max_{\Delta \vdash u : G' \in \mathcal{J}} \left\|G'\right\|$.*

Judgement weights actually take also logs into account.

Lemma VII.6 (Weights bound also extracted logs)**.** *Let $\pi \triangleright \Gamma \overset{w}{\vdash} t : G$ be a weighted derivation. Then $w \geq v + |J|_{\mathsf{b}}$ for every weighted judgment $J \vdash^{v}$ in π.*

We then obtain that the weight of a derivation π for t bounds the space used by the TIAM execution of π, and so by the λIAM execution of t.

Theorem VII.7 (λIAM space bounds)**.** *Let $\pi \triangleright \overset{w}{\vdash} t : \star$ be a weighted tree types derivation. Then $|\mathsf{ext}(s)|_{\mathsf{sp}} \leq w$ for every $s \in \mathsf{states}(\pi)$.*

[6]We introduce a different word for measuring space of the whole execution, because judgments are measured in two different ways: the *branch size* measures what is below the judgment, and corresponds to the size of the extracted log for a state, the *weight* measures what is above a judgment, and gives the maximum space over all states in the rooted sub-derivation.

Last, we show that weights provide *exact* bounds, as there always is a witness state using as much space as in the weight.

Proposition VII.8 (Weight witness)**.** *Let $\pi \triangleright \Gamma \overset{w}{\vdash} t : G$ be a weighted derivation and $G \neq [\cdot]$. Then there exists a TIAM state s over π such that $w = |s|_{\mathsf{b}}$.*

We can then conclude our complexity analysis.

Corollary VII.9 (λIAM exact bound via tree types derivations)**.** *Let $\pi \triangleright \overset{w}{\vdash} t : \star$ be a tree types derivation and ρ the complete λIAM run on t. Then $|\rho|_{\mathsf{s}} = w$.*

Now, the reader can fully understand and appreciate the weights in the derivation of Fig. 4. Please note that we have considered $\max\{1, \mathsf{X}\} = \mathsf{X}$ when assigning the weights.

VIII. On the λIAM Space (In)Efficiency

Tree Density: As we have proved in the last section, the space consumed by a λIAM run ultimately depends on the level of nesting of tree types. Let us be slightly more precise. When we are dealing with multi types, the argument u of an application tu has to be typed with a multiset, say $M := [\star, .., \star]$ where $|M| = n$ and n is the number of times that u shall be copied. In our type system, M can be represented as a tree T, capturing the space needed for evaluating the copies of u, in several ways. In the tree type derivation for tu only one of these representations is used, and depends on the type of t. The important point is that different representations lead to very different space consumptions. The ideal case is the flat representation $T_f := [\star, .., \star]$, for which $\left\|T_f\right\| = \mathsf{X}$. Another good case is given by a full binary tree T_b, or, more generally, by a tree T_d whose height is logarithmic in n, so that $\|T_d\| = \log n \cdot \mathsf{X}$. Let us deem this case *dense*. A bad case is given by linearly shaped trees such as $T_l := T_n$, where $T_0 := [\star]$ and $T_{n+1} := [\star, T_n]$, for which $\|T_s\| = (n+1) \cdot \mathsf{X} \geq \|T_d\|$. We call *sparse* a tree with n leaves and height n like T_l. Not that there can even be worse, as in general a tree can have an arbitrary number of internal nodes, for instance a multi set $[\star]$ can be represented as a tree also as $[[[[\star]]]]$—again, it depends on the type of t. Intuitively, if a term can be typed with flat or dense trees, the λIAM evaluates it space *efficiently*, while in the case of sparse trees (or worse) the evaluation is space *inefficient*.

Fixed-Points Are Sparse: As is well known, the λ-calculus is a universal (or Turing complete) model of computation. The fundamental ingredient that allows one to achieve universality is the presence of fixed-point combinators. These combinators allow for an encoding of general recursion, as needed when simulating, e.g., partial recursive functions or Turing machines. In particular, the fixed point combinator can capture unbounded iteration (i.e. `while` loops) or tail-recursion, as needed *e.g.* in the encoding of minimization from Kleene algebra.

As an example, consider how the encoding of a Turing machine M may look like, in the λ-calculus. Let t_M be the encoding of M's transition function (which is typically very simple if states and tapes are encoded using, e.g., Scott's numerals [38]). Then, the (recursively defined) function that

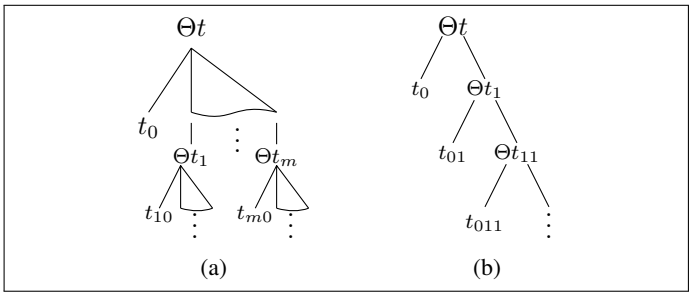

Figure 6: Different ways Θ can copy its argument.

iterates t_M, thus capturing the overall behavior of M, can be written as follows:

$$\mathsf{iter} = \underbrace{(\lambda f.\lambda s.\texttt{if } s \text{ is final } \texttt{then halt else } f(t_M s))}_{\mathsf{iteraux}}\,\mathsf{iter}$$

How can we build a solution to this equation in the form of a λ-term? Apart from an encoding of the conditional operator, itself very easy to write, we need a fixed-point combinator Θ, such as Turing's:

$$\Theta := \theta\theta \qquad \text{where} \qquad \theta := \lambda x.\lambda y.y(xxy)$$

We highlight that $\Theta t \to_{wh} (\lambda y.y(\Theta y))t \to_{wh} t(\Theta t)$. Then, we can set, as expected, $\mathsf{iter} := \Theta\mathsf{iteraux}$.

Let us analyze how Θ implements recursion, independently on what the argument of Θ is. Please note that during the reduction of Θt, the variable y is substituted for the term t, which after two β-steps appears twice, once in head position (call this occurrence t_0), and once applied to Θ. The latter copy of t, together with Θ, can be copied potentially many times, depending on how t_0 uses its argument. Some of these copies, say m, will eventually appear in head position, and the same process starts again. In other words, the copies of t that the combinator Θ will eventually create can be organized in a tree, see Figure 6a. This is a faithful description of how recursion unfolds, independently on how t uses its argument.

If $t = \mathsf{iteraux}$, however, the situation is much simpler: t uses its argument at most once, and the complicated tree in Figure 6a becomes the one in Figure 6b. Every copy t_{01^n} of t either brings $\Theta t_{1^{n+1}}$ in head position (without copying it), or discards it, depending on whether the current state is final or not. Saying it another way, the height of the tree in Figure 6b is nothing more than the number of reduction steps the Turing machine M performs.

In order to understand if the λIAM could be reasonable in space, *i.e.* if it can simulate Turing machines with a constant overhead in space, we apply our technique by giving Θ a suitable type (schema) $\mathbb{F}^{\overline{A}}_n$, depending on a list of types $\overline{A} := A_n, .., A_0$ and on n, which is the number of times the fixed point is unfolded. In particular, $\mathbb{F}^{\overline{A}}_n$ turns out to be *sparse*. The details of the technical development can be found in the technical report [7].

Proposition VIII.1. *For each $n \geq 0$, and for each list of types $\overline{A}$ such that $|\overline{A}| \geq n+1$, $\vdash^{\geq(2n+1)\mathsf{X}} \Theta : \mathbb{F}^{\overline{A}}_n$.*

In particular, $\Theta : \mathbb{F}^{\overline{A}}_n$ when Θ is unfolded n times and thus, inside the encoding of a Turing machine M which takes n steps to halt. This way the λIAM, independently of the space consumption of M, requires space at least linear in the number of reduction steps it performs, and there is thus no hope to stay within sub-linear space constraints.

(In)Efficiency in Related Works: How could we reconcile all this with the claims from the literature about the existence of sub-linear space bounds (Mazza [34] and Dal Lago and Schöpp [18], [19]) within the realm of geometry of interaction machines? The answer is relatively simple: the kind of machines considered in the cited works are *fundamentally different* than ours, being based on the idea that the information stored in logged positions can be taken to be a natural number *smaller or equal to* the cardinality of the underlying multi type. In Mazza [34], this is possible due to the peculiarities of the underlying type system. There, the use of non-linearity is much more restricted than in the λ-calculus, being based on parsimonious types. This allows for tail-recursive schemas *only*: in our terminology, all trees are linearly shaped, and then logged positions can be represented differently and with less space, namely by simply taking the height of the tree. The works by Dal Lago and Schöpp [18], [19] rely on the fact that the underlying type system is resource-aware, this way allowing for a different representation of logged positions as natural numbers rather than trees. Moreover, space-efficient simulations are done via an ad-hoc combinator for skewed iterations, thus circumventing the problem with fixed-point operators. This, unfortunately, is not available in the realm of the λ-calculus.

A question, however, remains. Is it possible *at all* that the λIAM consumes an amount of computation *space* significantly smaller than computation *time*? The answer is positive: there is a family of terms $\{t_i\}_{i\in\mathbb{N}}$ such that t_i reduces in Closed CbN to normal form in time exponential in i (namely, taking an exponential number of β-steps) but requires space linear in i, when reduced by the λIAM. Details are in the technical report [7].

IX. Conclusions

Space efficiency of higher-order languages accommodating sub-linear complexity is a topic about which almost nothing is known. The literature has suggested that the right tool to achieve it is the alternative paradigm of the IAM, a machine rooted in the geometry of interaction, without however clarifying whether its use of space can be used as a reasonable space cost model.

In this paper we develop a sharp tool—a type system—for the understanding of the space consumption of the (λ)IAM. Our new tree intersection types provide—for the first time—exact space bounds, via a simple system of weights measuring the depth of the tree structure in the types.

The tree structure seems to naturally complement the multiset one needed for measuring time, and it is of independent interest, given the relevance of intersection types in semantics. For instance, can such a structure be seen in game semantics? What

does it measure with respect to cut-elimination or environment machines?

Beyond the theoretical result, the type system has a direct application, as we show by studying the space usage of the IAM on the traditional encoding of Turing machines. Such a usage turns out to be very inefficient, providing negative insights on the elusive conjecture about the reasonable space usage of the IAM.

Acknowledgements. 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

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria, "Full abstraction for PCF," *Inf. Comput.*, vol. 163, no. 2, pp. 409–470, 2000.

[2] B. Accattoli and B. Barras, "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*, W. Vanhoof and B. Pientka, Eds. ACM, 2017, pp. 4–16.

[3] B. Accattoli and U. Dal Lago, "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*, ser. LIPIcs, A. Tiwari, Ed., vol. 15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012, pp. 22–37.

[4] ——, "(Leftmost-Outermost) Beta Reduction is Invariant, Indeed," *Logical Methods in Computer Science*, vol. 12, no. 1, 2016.

[5] B. Accattoli, U. Dal Lago, and G. Vanoni, "The machinery of interaction," in *PPDP '20: 22nd International Symposium on Principles and Practice of Declarative Programming, Bologna, Italy, 9-10 September, 2020*. ACM, 2020, pp. 4:1–4:15.

[6] ——, "The (in)efficiency of interaction," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–33, 2021.

[7] ——, "The space of interaction," Tech. Rep., 2021, https://arxiv.org/abs/2104.13795.

[8] B. Accattoli, S. Graham-Lengrand, and D. Kesner, "Tight typings and split bounds, fully developed," *J. Funct. Program.*, vol. 30, p. e14, 2020.

[9] B. Accattoli and G. Guerrieri, "Types of fireballs," in *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, ser. Lecture Notes in Computer Science, S. Ryu, Ed., vol. 11275. Springer, 2018, pp. 45–66.

[10] B. Accattoli, G. Guerrieri, and M. Leberle, "Types by need," in *28th European Symposium on Programming, ESOP 2019,Prague, Czech Republic, April 6-11, 2019, Proceedings*, ser. Lecture Notes in Computer Science, vol. 11423. Springer, 2019, pp. 410–439.

[11] S. Alves, D. Kesner, and D. Ventura, "A quantitative understanding of pattern matching," in *25th International Conference on Types for Proofs and Programs, TYPES 2019, June 11-14, 2019, Oslo, Norway*, 2019, pp. 3:1–3:36.

[12] H. P. Barendregt, *The lambda calculus: its syntax and semantics*. North-Holland, 1984.

[13] A. Bucciarelli, D. Kesner, A. Ríos, and A. Viso, "The bang calculus revisited," in *Functional and Logic Programming - 15th International Symposium, FLOPS 2020, Akita, Japan, September 14-16, 2020, Proceedings*, ser. Lecture Notes in Computer Science, K. Nakano and K. Sagonas, Eds., vol. 12073. Springer, 2020, pp. 13–32.

[14] L. Cardelli, "Type systems," in *The Computer Science and Engineering Handbook*, A. B. Tucker, Ed. CRC Press, 1997, pp. 2208–2236.

[15] M. Coppo and M. Dezani-Ciancaglini, "A new type assignment for λ-terms," *Arch. Math. Log.*, vol. 19, no. 1, pp. 139–156, 1978.

[16] U. Dal Lago and B. Accattoli, "Encoding turing machines into the deterministic lambda-calculus," *CoRR*, vol. abs/1711.10078, 2017.

[17] U. Dal Lago, C. Faggian, and S. Ronchi Della Rocca, "Intersection types and (positive) almost-sure termination," *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–32, 2021.

[18] U. Dal Lago and U. Schöpp, "Functional programming in sublinear space," in *19th European Symposium on Programming, ESOP 2010, Paphos, Cyprus, March 20-28, 2010, Proceedings.*, ser. Lecture Notes in Computer Science, A. D. Gordon, Ed., vol. 6012. Springer, 2010, pp. 205–225.

[19] U. Dal Lago and U. Schöpp, "Computation by interaction for space-bounded functional programming," *Information and Computation*, vol. 248, pp. 150–194, 2016.

[20] V. Danos, H. Herbelin, and L. Regnier, "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, 1996, pp. 394–405.

[21] V. Danos and L. Regnier, "Proof-nets and the hilbert space," in *Proceedings of the Workshop on Advances in Linear Logic*. USA: Cambridge University Press, 1995, p. 307–328.

[22] ——, "Reversible, irreversible and optimal lambda-machines," *Theoretical Computer Science*, vol. 227, no. 1, pp. 79–97, 1999.

[23] D. de Carvalho, "Sémantiques de la logique linéaire et temps de calcul," Thèse de Doctorat, Université Aix-Marseille II, 2007.

[24] ——, "Execution time of λ-terms via denotational semantics and intersection types," *Math. Str. in Comput. Sci.*, vol. 28, no. 7, pp. 1169–1203, 2018.

[25] D. de Carvalho, M. Pagani, and L. Tortora de Falco, "A semantic measure of the execution time in linear logic," *Theoretical Computer Science*, vol. 412, no. 20, pp. 1884–1902, 2011.

[26] Y. Forster, F. Kunze, and M. Roth, "The weak call-by-value λ-calculus is reasonable for both time and space," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 27:1–27:23, 2020.

[27] D. R. Ghica, "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*, M. Hofmann and M. Felleisen, Eds. ACM, 2007, pp. 363–375.

[28] D. R. Ghica and A. I. Smith, "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*, ser. Electronic Notes in Theoretical Computer Science, M. W. Mislove and P. Selinger, Eds., vol. 265. Elsevier, 2010, pp. 301–324.

[29] J.-Y. Girard, "Geometry of interaction 1: Interpretation of system f," in *Logic Colloquium '88*, ser. Studies in Logic and the Foundations of Mathematics, R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, Eds. Elsevier, 1989, vol. 127, pp. 221 – 260.

[30] D. Kesner and P. Vial, "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*, H. Hermanns, L. Zhang, N. Kobayashi, and D. Miller, Eds. ACM, 2020, pp. 619–632.

[31] J. Krivine, *Lambda-calculus, types and models*, ser. Ellis Horwood series in computers and their applications. Masson, 1993.

[32] J.-L. Krivine, "A Call-by-name Lambda-calculus Machine," *Higher Order Symbol. Comput.*, vol. 20, no. 3, pp. 199–207, 2007.

[33] I. Mackie, "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*, R. K. Cytron and P. Lee, Eds. ACM Press, 1995, pp. 198–208.

[34] D. Mazza, "Simple parsimonious types and logarithmic space," in *24th EACSL Annual Conference on Computer Science Logic, CSL 2015, September 7-10, 2015, Berlin, Germany*, ser. LIPIcs, S. Kreutzer, Ed., vol. 41. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 24–40.

[35] D. Mazza, L. Pellissier, and P. Vial, "Polyadic approximations, fibrations and intersection types," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 6:1–6:28, 2018.

[36] U. Schopp, "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, 2007, pp. 411–420.

[37] T. Tsukada, K. Asada, and C.-H. L. Ong, "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, 2017, pp. 1–12.

[38] C. P. Wadsworth, "Some unusual λ-calculus numeral systems," in *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, J. P. Seldin and J. R. Hindley, Eds., 1980, pp. 215–230.